Python Frequently Asked Questions

Release 3.12.7

Guido van Rossum and the Python development team

outubro 18, 2024

Python Software Foundation
Email: docs@python.org

Sumário

1 Python Geral **1**
1.1 Informações gerais . . . 1
1.1.1 O que é Python? . . . 1
1.1.2 O que é a Python Software Foundation? . . . 1
1.1.3 Existem restrições de direitos autorais sobre o uso de Python? . . . 1
1.1.4 Em primeiro lugar, por que o Python foi desenvolvido? . . . 2
1.1.5 Para o que Python é excelente? . . . 2
1.1.6 Como funciona o esquema de numeração de versões do Python? . . . 2
1.1.7 Como faço para obter uma cópia dos fonte do Python? . . . 3
1.1.8 Como faço para obter a documentação do Python? . . . 3
1.1.9 Eu nunca programei antes. Existe um tutorial básico do Python? . . . 3
1.1.10 Existe um grupo de discussão ou lista de discussão dedicada ao Python? . . . 3
1.1.11 Como faço para obter uma versão de teste beta do Python? . . . 4
1.1.12 Como eu envio um relatório de erros e correções para o Python? . . . 4
1.1.13 Existem alguns artigos publicados sobre o Python para que eu possa fazer referência? . . . 4
1.1.14 Existem alguns livros sobre o Python? . . . 4
1.1.15 Onde está armazenado o site www.python.org? . . . 4
1.1.16 Por que o nome Python? . . . 4
1.1.17 Eu tenho que gostar de "Monty Python's Flying Circus"? . . . 4
1.2 Python no mundo real . . . 4
1.2.1 Quão estável é o Python? . . . 4
1.2.2 Quantas pessoas usam o Python? . . . 5
1.2.3 Existe algum projeto significativo feito em Python? . . . 5
1.2.4 Quais são os novos desenvolvimentos esperados para o Python no futuro? . . . 5
1.2.5 É razoável propor mudanças incompatíveis com o Python? . . . 5
1.2.6 O Python é uma boa linguagem para quem está começando na programação agora? . . . 5

2 FAQ referente a Programação **7**
2.1 Questões Gerais . . . 7
2.1.1 Existe um depurador a nível de código-fonte que possuo pontos de interrupção (breakpoints), single-stepping etc.? . . . 7
2.1.2 Existem ferramentas para ajudar a encontrar bugs ou fazer análise estática de desempenho? 8
2.1.3 Como posso criar um binário independente a partir de um script Python? . . . 8
2.1.4 Existem padrões para a codificação ou um guia de estilo utilizado pela comunidade Python? 8
2.2 Núcleo da Linguagem . . . 8
2.2.1 Porque estou recebo o erro UnboundLocalError quando a variável possui um valor associado? 8
2.2.2 Quais são as regras para variáveis locais e globais em Python? . . . 9
2.2.3 Por que os lambdas definidos em um loop com valores diferentes retornam o mesmo resultado? . . . 10
2.2.4 Como definir variáveis globais dentro de módulos? . . . 10
2.2.5 Quais são as "melhores práticas" quando fazemos uso da importação de módulos? . . . 11

2.2.6 Por que os valores padrão são compartilhados entre objetos? . . . 11
2.2.7 Como passar parâmetros opcionais ou parâmetros na forma de keyword de uma função a outra? . . . 12
2.2.8 Qual a diferença entre argumentos e parâmetros? . . . 13
2.2.9 Por que ao alterar a lista 'y' também altera a lista 'x'? . . . 13
2.2.10 Como escrever uma função com parâmetros de saída (invocada por referência)? . . . 14
2.2.11 Como fazer uma função de ordem superior em Python? . . . 15
2.2.12 Como faço para copiar um objeto no Python? . . . 16
2.2.13 Como posso encontrar os métodos ou atributos de um objeto? . . . 16
2.2.14 Como que o meu código pode descobrir o nome de um objeto? . . . 16
2.2.15 O que há com a precedência do operador vírgula? . . . 17
2.2.16 Existe um equivalente ao operador "?:" ternário do C? . . . 17
2.2.17 É possível escrever instruções de uma só linha ofuscadas em Python? . . . 17
2.2.18 O que a barra(/) na lista de parâmetros de uma função significa? . . . 18
2.3 Números e Strings . . . 18
2.3.1 Como faço para especificar números inteiros hexadecimais e octal? . . . 18
2.3.2 Por que -22 // 10 retorna -3? . . . 19
2.3.3 How do I get int literal attribute instead of SyntaxError? . . . 19
2.3.4 Como faço para converter uma String em um número? . . . 19
2.3.5 Como faço para converter um número numa string? . . . 19
2.3.6 Como faço para modificar uma string no lugar? . . . 20
2.3.7 Como faço para invocar funções/métodos através de uma String? . . . 20
2.3.8 Existe um equivalente em Perl `chomp()` para remover linhas novas de uma String? . . . 21
2.3.9 Existe uma função scanf() ou sscanf() ou algo equivalente? . . . 21
2.3.10 O que significa o erro 'UnicodeDecodeError' ou 'UnicodeEncodeError'? . . . 21
2.3.11 Can I end a raw string with an odd number of backslashes? . . . 21
2.4 Desempenho . . . 22
2.4.1 Meu programa está muito lento. Como faço para melhorar o desempenho? . . . 22
2.4.2 Qual é a maneira mais eficiente de concatenar muitas Strings? . . . 23
2.5 Sequencias (Tuples/Lists) . . . 23
2.5.1 Como faço para converter tuplas em listas? . . . 23
2.5.2 O que é um índice negativo? . . . 23
2.5.3 Como que eu itero uma sequência na ordem inversa? . . . 24
2.5.4 Como que remove itens duplicados de uma lista? . . . 24
2.5.5 Como remover múltiplos itens de uma lista . . . 24
2.5.6 Como fazer um vetor em Python? . . . 24
2.5.7 Como faço para criar uma lista multidimensional? . . . 25
2.5.8 How do I apply a method or function to a sequence of objects? . . . 25
2.5.9 Porque a_tuple[i] += ['item'] levanta uma exceção quando a adição funciona? . . . 26
2.5.10 I want to do a complicated sort: can you do a Schwartzian Transform in Python? . . . 27
2.5.11 Como eu posso ordenar uma lista pelos valores de outra lista? . . . 27
2.6 Objetos . . . 27
2.6.1 O que é uma classe? . . . 27
2.6.2 O que é um método? . . . 27
2.6.3 O que é o self? . . . 28
2.6.4 Como eu verifico se um objeto é uma instância de uma dada classe ou de uma subclasse dela? 28
2.6.5 O que é delegation? . . . 29
2.6.6 Como eu chamo um método definido numa classe base derivada de uma classe que estende ela? . . . 30
2.6.7 Como eu posso organizar meu código para facilitar a troca da classe base? . . . 30
2.6.8 How do I create static class data and static class methods? . . . 30
2.6.9 Como eu posso sobrecarregar construtores (ou métodos) em Python? . . . 31
2.6.10 Eu tentei usar __spam e recebi um erro sobre _SomeClassName__spam. . . . 31
2.6.11 Minha classe define __del__, mas o mesmo não é chamado quando eu excluo o objeto. . . . 32
2.6.12 Como eu consigo pegar uma lista de todas as instâncias de uma dada classe? . . . 32
2.6.13 Por que o resultado de `id()` aparenta não ser único? . . . 32
2.6.14 Quando eu posso depender dos testes de identidade com o operador *is*? . . . 33
2.6.15 How can a subclass control what data is stored in an immutable instance? . . . 34

2.6.16 How do I cache method calls? . . . 34
2.7 Módulos . . . 36
2.7.1 Como faço para criar um arquivo .pyc? . . . 36
2.7.2 Como encontro o nome do módulo atual? . . . 36
2.7.3 How can I have modules that mutually import each other? . . . 36
2.7.4 __import__('x.y.z') returns <module 'x'>; how do I get z? . . . 37
2.7.5 Quando eu edito um módulo importado e o reimporto, as mudanças não aparecem. Por que isso acontece? . . . 37

3 Design e Histórico FAQ **39**
3.1 Por que o Python usa indentação para agrupamento de instruções? . . . 39
3.2 Por que eu estou recebendo resultados estranhos com simples operações aritméticas? . . . 39
3.3 Por que o cálculo de pontos flutuantes são tão imprecisos? . . . 40
3.4 Por que strings do Python são imutáveis? . . . 40
3.5 Por que o 'self' deve ser usado explicitamente em definições de método e chamadas? . . . 40
3.6 Por que não posso usar uma atribuição em uma expressão? . . . 41
3.7 Por que o Python usa métodos para algumas funcionalidades (ex: lista.index()) mas funções para outras (ex: len(lista))? . . . 41
3.8 Por que o join() é um método de string em vez de ser um método de lista ou tupla? . . . 41
3.9 Quão rápidas são as exceções? . . . 42
3.10 Por que não existe uma instrução de switch ou case no Python? . . . 42
3.11 Você não pode emular threads no interpretador em vez de confiar em uma implementação de thread específica do sistema operacional? . . . 43
3.12 Por que expressões lambda não podem conter instruções? . . . 43
3.13 O Python pode ser compilado para linguagem de máquina, C ou alguma outra linguagem? . . . 43
3.14 Como o Python gerencia memória? . . . 44
3.15 Por que o CPython não usa uma forma mais tradicional de esquema de coleta de lixo? . . . 44
3.16 Por que toda memória não é liberada quando o CPython fecha? . . . 44
3.17 Por que existem tipos de dados separados para tuplas e listas? . . . 45
3.18 Como as listas são implementadas no CPython? . . . 45
3.19 Como são os dicionários implementados no CPython? . . . 45
3.20 Por que chaves de dicionário devem ser imutáveis? . . . 45
3.21 Por que lista.sort() não retorna a lista ordenada? . . . 46
3.22 Como você especifica e aplica uma especificação de interface no Python? . . . 47
3.23 Por que não há goto? . . . 47
3.24 Por que strings brutas (r-strings) não podem terminar com uma contrabarra? . . . 48
3.25 Por que o Python não tem uma instrução "with" para atribuição de atributos? . . . 48
3.26 Por que os geradores não suportam a instrução with? . . . 49
3.27 Por que dois pontos são necessários para as instruções de if/while/def/class? . . . 49
3.28 Por que o Python permite vírgulas ao final de listas e tuplas? . . . 49

4 FAQ de Bibliotecas e Extensões **51**
4.1 Questões gerais sobre bibliotecas . . . 51
4.1.1 Como encontrar um módulo ou aplicação para realizar uma tarefa X? . . . 51
4.1.2 Onde está o código-fonte do math.py (socket.py, regex.py, etc.)? . . . 51
4.1.3 Como tornar um script Python executável no Unix? . . . 51
4.1.4 Existe um pacote de curses/termcap para Python? . . . 52
4.1.5 Existe a função onexit() equivalente ao C no Python? . . . 52
4.1.6 Por que o meu manipulador de sinal não funciona? . . . 52
4.2 Tarefas comuns . . . 52
4.2.1 Como testar um programa ou componente Python? . . . 52
4.2.2 Como faço para criar uma documentação de doc strings? . . . 53
4.2.3 Como faço para pressionar uma tecla de cada vez? . . . 53
4.3 Threads . . . 53
4.3.1 Como faço para programar usando threads? . . . 53
4.3.2 Nenhuma de minhas threads parece funcionar, por que? . . . 53
4.3.3 Como distribuo o trabalho entre várias threads de trabalho? . . . 54
4.3.4 Que tipos de variáveis globais mutáveis são seguras para thread? . . . 55

4.3.5 Não podemos remover a Trava Global do interpretador? . . . 56
4.4 Entrada e Saída . . . 56
4.4.1 Como faço para excluir um arquivo? (E outras perguntas sobre arquivos) . . . 56
4.4.2 Como eu copio um arquivo? . . . 57
4.4.3 Como leio (ou escrevo) dados binários? . . . 57
4.4.4 Por que não consigo usar os.read() em um encadeamento com os.popen()? . . . 57
4.4.5 Como acesso a porta serial (RS232)? . . . 57
4.4.6 Por que o sys.stdout (stdin, stderr) não fecha? . . . 58
4.5 Programação Rede / Internet . . . 58
4.5.1 Quais ferramentas para WWW existem no Python? . . . 58
4.5.2 Como faço para imitar a submissão de formulário CGI (METHOD=POST)? . . . 58
4.5.3 Qual módulo devo usar para ajudar na geração do HTML? . . . 59
4.5.4 Como envio um e-mail de um script Python? . . . 59
4.5.5 Como evito um bloqueio no método connect() de um soquete? . . . 59
4.6 Base de Dados . . . 60
4.6.1 Existem interfaces para banco de dados em Python? . . . 60
4.6.2 Como você implementa objetos persistentes no Python? . . . 60
4.7 Matemáticos e Numéricos . . . 60
4.7.1 Como gero número aleatórios no Python? . . . 60

5 FAQ sobre Extensão/Incorporação **61**
5.1 Posso criar minhas próprias funções em C? . . . 61
5.2 Posso criar minhas próprias funções em C++? . . . 61
5.3 A escrita em C é difícil, Há algumas alternativas? . . . 61
5.4 Como posso executar instruções arbitrárias de Python a partir de C? . . . 61
5.5 How can I evaluate an arbitrary Python expression from C? . . . 62
5.6 Como extraio valores em C a partir de um objeto Python? . . . 62
5.7 Como posso utilizar Py_BuildValue() para criar uma tupla de comprimento arbitrário? . . . 62
5.8 How do I call an object's method from C? . . . 62
5.9 How do I catch the output from PyErr_Print() (or anything that prints to stdout/stderr)? . . . 63
5.10 How do I access a module written in Python from C? . . . 63
5.11 How do I interface to C++ objects from Python? . . . 63
5.12 I added a module using the Setup file and the make fails; why? . . . 64
5.13 How do I debug an extension? . . . 64
5.14 I want to compile a Python module on my Linux system, but some files are missing. Why? . . . 64
5.15 How do I tell "incomplete input" from "invalid input"? . . . 64
5.16 How do I find undefined g++ symbols __builtin_new or __pure_virtual? . . . 64
5.17 Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)? . . . 65

6 Python no Windows **67**
6.1 Como faço para executar um programa Python no Windows? . . . 67
6.2 Como eu faço para criar programas Python executáveis? . . . 68
6.3 Por que Python às vezes demora tanto para iniciar? . . . 68
6.4 Como eu faço para criar um executável a partir de um código Python? . . . 68
6.5 Um arquivo `*.pyd` é o mesmo que um DLL? . . . 69
6.6 Como eu posso embutir Python dentro de uma aplicação do Windows? . . . 69
6.7 Como eu impeço editores de adicionarem tabulações na minha source do Python? . . . 70
6.8 Como faço para verificar uma tecla pressionada sem bloquear? . . . 70
6.9 Como resolvo o erro da api-ms-win-crt-runtime-l1-1-0.dll ausente? . . . 70

7 FAQ da Interface Gráfica do Usuário **71**
7.1 Perguntas Gerais sobre a GUI . . . 71
7.2 Quais toolkits de GUI existem para o Python? . . . 71
7.3 Perguntas do Tkinter . . . 71
7.3.1 Como eu congelo as aplicações Tkinter? . . . 71
7.3.2 Posso ter eventos Tk manipulados enquanto aguardo pelo E/S? . . . 71
7.3.3 Não consigo fazer as ligações de tecla funcionarem no Tkinter: por que? . . . 72

8 FAD de "Por que o Python está instalado em meu computador?" 73
8.1 O que é Python? . . . 73
8.2 Porque Python está instalado em minha máquina? . . . 73
8.3 Eu posso apagar o Python? . . . 73

A Glossário 75

B Sobre esses documentos 93
B.1 Contribuidores da Documentação Python . . . 93

C História e Licença 95
C.1 História do software . . . 95
C.2 Termos e condições para acessar ou usar Python . . . 96
C.2.1 ACORDO DE LICENCIAMENTO DA PSF PARA PYTHON 3.12.7 . . . 96
C.2.2 ACORDO DE LICENCIAMENTO DA BEOPEN.COM PARA PYTHON 2.0 . . . 97
C.2.3 CONTRATO DE LICENÇA DA CNRI PARA O PYTHON 1.6.1 . . . 97
C.2.4 ACORDO DE LICENÇA DA CWI PARA PYTHON 0.9.0 A 1.2 . . . 99
C.2.5 LICENÇA BSD DE ZERO CLÁUSULA PARA CÓDIGO NA DOCUMENTAÇÃO DO PYTHON 3.12.7 . . . 99
C.3 Licenças e Reconhecimentos para Software Incorporado . . . 99
C.3.1 Mersenne Twister . . . 99
C.3.2 Soquetes . . . 100
C.3.3 Serviços de soquete assíncrono . . . 101
C.3.4 Gerenciamento de cookies . . . 101
C.3.5 Rastreamento de execução . . . 102
C.3.6 Funções UUencode e UUdecode . . . 102
C.3.7 Chamadas de procedimento remoto XML . . . 103
C.3.8 test_epoll . . . 103
C.3.9 kqueue de seleção . . . 104
C.3.10 SipHash24 . . . 104
C.3.11 strtod e dtoa . . . 105
C.3.12 OpenSSL . . . 105
C.3.13 expat . . . 108
C.3.14 libffi . . . 109
C.3.15 zlib . . . 109
C.3.16 cfuhash . . . 110
C.3.17 libmpdec . . . 111
C.3.18 Conjunto de testes C14N do W3C . . . 111
C.3.19 Audioop . . . 112
C.3.20 asyncio . . . 112

D Direitos autorais 113

Índice 115

CAPÍTULO 1

Python Geral

1.1 Informações gerais

1.1.1 O que é Python?

O Python é uma linguagem de programação interpretada, interativa e orientada a objetos. O mesmo incorporou módulos, exceções, tipagem dinâmica, tipos de dados dinâmicos de alto nível e classes. Há suporte a vários paradigmas de programação além da programação orientada a objetos, tal como programação procedural e funcional. O Python fornece ao desenvolvedor um poder notável aliado a uma sintaxe simples de clara. Possui interfaces para muitas chamadas e bibliotecas do sistema, bem como para vários sistemas de janelas, e é extensível através de linguagem como o C ou C++. Também é utilizado como linguagem de extensão para aplicativos que precisam de uma interface programável. Finalmente, o Python é portátil: o mesmo pode ser executado em várias variantes do Unix, incluindo Linux e Mac, e no Windows.

Para saber mais, inicie pelo nosso tutorial tutorial-index. Os links do Beginner's Guide to Python para outros tutoriais introdutórios e recursos da linguagem Python.

1.1.2 O que é a Python Software Foundation?

O Python Software Foundation é uma organização independente e sem fins lucrativos que detém os direitos autorais sobre as versões 2.1 do Python e as mais recentes. A missão do PSF é avançar a tecnologia de código aberto relacionada à linguagem de programação Python e divulgar a utilização do Python. A página inicial do PSF pode ser acessada pelo link a seguir https://www.python.org/psf/.

Doações para o PSF estão isentas de impostos nos EUA. Se utilizares o Python e achares útil, contribua através da página de doação da PSF.

1.1.3 Existem restrições de direitos autorais sobre o uso de Python?

Podemos fazer tudo o que quisermos com os fontes, desde que deixemos os direitos autorais e exibamos esses direitos em qualquer documentação sobre o Python que produzirmos. Se honrarmos as regras dos direitos autorais, não há quaisquer problema em utilizar o Python em versões comerciais, vendê-lo, copiá-lo na forma de código-fonte ou o seu binária (modificado ou não modificado), ou para vender produtos que incorporem o Python de alguma forma. Ainda gostaríamos de saber sobre todo o uso comercial de Python, é claro.

Veja a página da licença para encontrar mais explicações e um link para o texto completo da licença.

O logotipo do Python é marca registrada e, em certos casos, é necessária permissão para usá-la. Consulte a Política de Uso da Marca comercial para obter mais informações.

1.1.4 Em primeiro lugar, por que o Python foi desenvolvido?

Aqui está um resumo *muito* breve de como que tudo começou, escrito por Guido van Rossum:

> Eu tive vasta experiência na implementação de linguagens interpretada no grupo ABC da CWI e, ao trabalhar com esse grupo, aprendi muito sobre o design de linguagens. Esta é a origem de muitos recursos do Python, incluindo o uso do recuo para o agrupamento de instruções e a inclusão de tipos de dados de alto nível (embora existam diversos detalhes diferentes em Python).
>
> Eu tinha uma série de queixas sobre a linguagem ABC, mas também havia gostado de muitos das suas características. Era impossível estender ABC (ou melhorar a implementação) para remediar minhas queixas – na verdade, a falta de extensibilidade era um dos maiores problemas. Eu tinha alguma experiência com o uso de Modula-2+ e conversei com os designers do Modula-3 e li o relatório do Modula-3. Modula-3 foi a origem da sintaxe e semântica usada nas exceções, e alguns outros recursos do Python.
>
> Eu estava trabalhando no grupo de sistema operacional distribuído da Amoeba na CWI. Precisávamos de uma maneira melhor de administrar o sistema do que escrevendo programas em C ou scripts para a shell Bourne, uma vez que o Amoeba tinha a sua própria interface de chamada do sistema, que não era facilmente acessível a partir do shell Bourne. Minha experiência com o tratamento de erros em Amoeba me conscientizou da importância das exceções como um recurso das linguagens de programação.
>
> Percebi que uma linguagem de script com uma sintaxe semelhante a do ABC, mas com acesso às chamadas do sistema Amoeba, preencheria a necessidade. Percebi também que seria uma boa escrever uma linguagem específica para o Amoeba, então, decidi que precisava de uma linguagem realmente extensível.
>
> Durante as férias do Natal de 1989, tive bastante tempo disponível e então decidi tentar a construção de algo. Durante o ano seguinte, continuei trabalhando em minhas horas vagas, e o Python foi usado no projeto Amoeba com crescente sucesso, e o feedback dos colegas me fez implementar muitas melhorias.
>
> Em fevereiro de 1991, depois de mais de um ano de desenvolvimento, decidi publicar na USENET. O resto está no arquivo `Misc/HISTORY`.

1.1.5 Para o que Python é excelente?

Python é uma linguagem de programação de propósito geral, de alto nível e que pode ser aplicada em muitos tipos diferentes de problemas.

A linguagem vem com uma grande biblioteca padrão que abrange áreas como processamento de strings (expressões regulares, Unicode, cálculo de diferenças entre arquivos), protocolos de internet (HTTP, FTP, SMTP, XML-RPC, POP, IMAP), engenharia de software (testes de unidade , registro, criação de perfil, análise de código Python) e interfaces de sistema operacional (chamadas de sistema, sistemas de arquivos, soquetes TCP/IP). Veja o sumário de library-index para ter uma ideia do que está disponível. Uma ampla variedade de extensões de terceiros também está disponível. Consulte o Python Package Index para encontrar pacotes de seu interesse.

1.1.6 Como funciona o esquema de numeração de versões do Python?

As versões de Python são enumeradas como "A.B.C" ou "A.B":

- *A* é o número da versão principal - sendo incrementada apenas em grandes mudanças na linguagem.
- *B* é o número da versão menor - sendo incrementada apenas para mudanças menos estruturais.
- *C* é o número para micro versão – sendo incrementada apenas para lançamento com correção de bugs.

Nem todos as versões são lançamentos de correções de erros. Na corrida por um novo lançamento de funcionalidade, uma série de versões de desenvolvimento são feitas, denotadas como alfa, beta ou candidata. As versões alfa são lançamentos iniciais (early releases) em que as interfaces ainda não estão finalizadas; não é inesperado ver uma mudança de interface entre duas versões alfa. As betas são mais estáveis, preservando as interfaces existentes, mas possivelmente adicionando novos módulos, e as candidatas a lançamento são congeladas, sem alterações, exceto quando necessário para corrigir erros críticos.

As versões alpha, beta e candidata a lançamento possuem um sufixo adicional:

- O sufixo para uma versão alfa é "aN" para algum número pequeno *N*.

- O sufixo para uma versão beta é "bN" para algum número pequeno *N*.
- O sufixo para um lançamento em versão candidata é "rcN" para algum número pequeno *N*.

Em outras palavras, todas as versões rotuladas como *2.0aN* precedem as versões rotuladas como *2.0bN*, que por sua vez precedem versões rotuladas como *2.0rcN*, e *estas* precedem 2.0.

Também podemos encontrar números de versão com um sufixo "+", por exemplo, "2.2+". Estas são versões não lançadas, construídas diretamente do repositório de desenvolvimento do CPython. Na prática, após uma última versão menor, a versão é incrementada para a próxima versão secundária, que se torna a versão "a0", por exemplo, "2.4a0".

Veja o Developer's Guide para mais informações sobre o ciclo de desenvolvimento, e a **PEP 387** para aprender mais sobre a política de compatibilidade com versões anteriores do Python. Veja também a documentação para `sys.version`, `sys.hexversion`, e `sys.version_info`.

1.1.7 Como faço para obter uma cópia dos fonte do Python?

A última distribuição fonte do Python sempre está disponível no python.org, em https://www.python.org/downloads/. As últimas fontes de desenvolvimento podem ser obtidas em https://github.com/python/cpython/.

A distribuição fonte é um arquivo .tar com .gzip contendo o código-fonte C completo, a documentação formatada com o Sphinx, módulos de biblioteca Python, programas de exemplo e várias peças úteis de software livremente distribuível. A fonte compilará e executará sem a necessidade de configurações extras na maioria das plataformas UNIX.

Consulte a seção Introdução do Guia do Desenvolvedor Python para obter mais informações sobre como obter o código-fonte e compilá-lo.

1.1.8 Como faço para obter a documentação do Python?

A documentação padrão para a versão atualmente estável do Python está disponível em https://docs.python.org/3/. Em PDF, texto simples e versões HTML para download também estão disponíveis em https://docs.python.org/3/download.html.

A documentação é escrita em reStructuredText e processada pela ferramenta de documentação Sphinx. Os fonte do reStructuredText para documentação fazem parte da distribuição fonte do Python.

1.1.9 Eu nunca programei antes. Existe um tutorial básico do Python?

Existem inúmeros tutoriais e livros disponíveis. A documentação padrão inclui tutorial-index.

Consulte o Guia do Iniciante para encontrar informações para quem está começando agora na programação Python, incluindo uma lista com tutoriais.

1.1.10 Existe um grupo de discussão ou lista de discussão dedicada ao Python?

Existe um grupo de notícias `comp.lang.python`, e uma lista de discussão, python-list. O grupo notícias e a lista de discussão são conectados um ou outro – se poderes ler as notícias, não será necessário se inscrever na lista de discussão. `comp.lang.python` possui bastante postagem, recebendo centenas de postagens todos os dias, e os leitores do Usenet geralmente são mais capazes de lidar com esse volume.

Os anúncios de novas versões do software e eventos podem ser encontrados em comp.lang.python.announce, uma lista moderada de baixo tráfego que recebe cerca de cinco postagens por dia. Está disponível como a lista de discussão python-announce.

Mais informações sobre outras listas de discussão e grupos de notícias podem ser encontradas em https://www.python.org/community/lists/.

1.1.11 Como faço para obter uma versão de teste beta do Python?

As versões alfa e beta estão disponíveis em https://www.python.org/downloads/. Todos os lançamentos são anunciados nos grupos de notícias comp.lang.python e comp.lang.python.announce e na página inicial do Python em https://www.python.org/; um feed RSS de notícias está disponível.

Você também pode acessar a versão de desenvolvimento do Python através do Git. Veja O Guia do Desenvolvedor Python para detalhes.

1.1.12 Como eu envio um relatório de erros e correções para o Python?

Para relatar um bug ou enviar um patch, use o rastreador de problemas em https://github.com/python/cpython/issues.

Para mais informações sobre como o Python é desenvolvido, consulte o Guia do Desenvolvedor Python.

1.1.13 Existem alguns artigos publicados sobre o Python para que eu possa fazer referência?

Provavelmente será melhor citar o seu livro favorito sobre o Python.

O primeiro artigo sobre Python foi escrito em 1991 e atualmente se encontra bastante desatualizado.

> Guido van Rossum e Jelke de Boer, "Interactively Testing Remote Servers Using the Python Programming Language", CWI Quarterly, Volume 4, Edição 4 (dezembro de 1991), Amsterdam, pp. 283–303.q

1.1.14 Existem alguns livros sobre o Python?

Sim, há muitos publicados e muitos outros que estão sendo nesse momento escritos!! Veja o wiki python.org em https://wiki.python.org/moin/PythonBooks para obter uma listagem.

Você também pode pesquisar livrarias online sobre "Python" e filtrar as referências a respeito do Monty Python; ou talvez procure por "Python" e "linguagem".

1.1.15 Onde está armazenado o site www.python.org?

A infraestrutura do projeto Python está localizada em todo o mundo e é gerenciada pela equipe de infraestrutura do Python. Detalhes aqui.

1.1.16 Por que o nome Python?

Quando o Guido van Rossum começou a implementar o Python, o mesmo também estava lendo os scripts publicados do "Monty Python's Flying Circus", uma série de comédia da BBC da década de 1970. Van Rossum pensou que precisava de um nome curto, único e ligeiramente misterioso, então resolveu chamar a sua linguagem de Python.

1.1.17 Eu tenho que gostar de "Monty Python's Flying Circus"?

Não, mas isso ajuda. :)

1.2 Python no mundo real

1.2.1 Quão estável é o Python?

Muito estável. Novos lançamentos estáveis são divulgados aproximadamente de 6 a 18 meses desde 1991, e isso provavelmente continuará. A partir da versão 3.9, o Python terá um novo grande lançamento a cada 12 meses (**PEP 602**).

Os desenvolvedores lançam versões bugfix de versões mais antigas, então a estabilidade dos lançamentos existentes melhora gradualmente. As liberações de correções de erros, indicadas por um terceiro componente do número da versão (por exemplo, 3.5.3, 3.6.2), são gerenciadas para estabilidade; somente correções para problemas conhecidos são incluídas em uma versão de correções de erros, e é garantido que as interfaces permanecerão as mesmas durante uma série de liberações de correções de erros.

The latest stable releases can always be found on the Python download page. Python 3.x is the recommended version and supported by most widely used libraries. Python 2.x **is not maintained anymore**.

1.2.2 Quantas pessoas usam o Python?

Provavelmente existem milhões de usuários, embora seja difícil obter uma contagem exata.

O Python está disponível para download gratuito, portanto, não há números de vendas, e o mesmo está disponível em vários diferentes sites e é empacotado em muitas distribuições Linux, portanto, utilizar as estatísticas de downloads não seria a melhor forma para contabilizarmos a base de usuários.

O grupo de notícias comp.lang.python é bastante ativo, mas nem todos os usuários Python postam no grupo ou mesmo o leem regularmente.

1.2.3 Existe algum projeto significativo feito em Python?

Veja a lista em https://www.python.org/about/success para obter uma listagem de projetos que usam o Python. Consultar as conferências passadas do Python revelará as contribuições de várias empresas e de diferentes organizações.

Os projetos Python de alto perfil incluem o gerenciador de lista de e-mail Mailman e o servidor de aplicativos Zope. Várias distribuições Linux, mais notavelmente o Red Hat, escreveram parte ou a totalidade dos seus instaladores e software de administração do sistema em Python. Empresas que usam Python internamente incluem Google, Yahoo e Lucasfilm Ltd.

1.2.4 Quais são os novos desenvolvimentos esperados para o Python no futuro?

Consulte https://peps.python.org/ para ver a lista de propostas de aprimoramento do python (PEPs). As PEPs são documentos de design que descrevem novos recursos que foram sugeridos para o Python, fornecendo uma especificação técnica concisa e a sua lógica. Procure uma PEP intitulado de "Python X.Y Release Schedule", onde X.Y é uma versão que ainda não foi lançada publicamente.

Novos desenvolvimentos são discutidos na lista de discussão python-dev.

1.2.5 É razoável propor mudanças incompatíveis com o Python?

Normalmente não. Já existem milhões de linhas de código Python em todo o mundo, de modo que qualquer alteração na linguagem que invalide mais de uma fração muito pequena dos programas existentes deverá ser desaprovada. Mesmo que possamos fornecer um programa de conversão, ainda haverá o problema de atualizar toda a documentação; muitos livros foram escritos sobre o Python, e não queremos invalidá-los todos de uma vez só.

Fornecer um caminho de atualização gradual será necessário se um recurso precisar ser alterado. A **PEP 5** descreve o procedimento e em seguida introduz alterações incompatíveis com versões anteriores ao mesmo tempo em que minimiza a interrupção dos usuários.

1.2.6 O Python é uma boa linguagem para quem está começando na programação agora?

Sim.

Ainda é bastante comum que os alunos iniciem com uma linguagem procedimental e estaticamente tipada como Pascal e o C ou um subconjunto do C++ ou do Java. Os alunos podem ser melhor atendidos ao aprender Python como sua primeira linguagem. Python possui uma sintaxe muito simples e consistente e uma grande quantidade de bibliotecas padrão e, o mais importante, o uso do Python em um curso de programação para iniciantes permite aos alunos se concentrarem em habilidades de programação importantes, como a decomposição do problema e o design do tipo de dados. Com Python, os alunos podem ser introduzidos rapidamente em conceitos básicos, como repetições e procedimentos. Provavelmente os mesmos até poderão trabalhar com objetos definidos por ele mesmos logo em seu primeiro curso.

Para um aluno que nunca programou antes, usar uma linguagem estaticamente tipado parece não que não é natural. Isso apresenta uma complexidade adicional que o aluno deverá dominar e geralmente retarda o ritmo do curso. Os alunos estão tentando aprender a pensar como um computador, decompor problemas, projetar interfaces consistentes

e encapsular dados. Embora aprender a usar uma linguagem tipicamente estática seja importante a longo prazo, não é necessariamente o melhor tópico a ser abordado no primeiro momento de um curso de programação.

Muitos outros aspectos do Python fazem do mesmo uma excelente linguagem para quem está aprendendo a programar. Como Java, Python possui uma biblioteca padrão grande para que os estudantes possam receber projetos de programação muito cedo no curso e que possam *fazer* trabalhos úteis. As atribuições não estão restritas à calculadora padrão de quatro funções e os programas para verificar o peso. Ao usar a biblioteca padrão, os alunos podem ter a satisfação de trabalhar em aplicações reais à medida que aprendem os fundamentos da programação. O uso da biblioteca padrão também ensina os alunos sobre a reutilização de código. Os módulos de terceiros, como o PyGame, também são úteis para ampliar o alcance dos estudantes.

O interpretador interativo do Python permite aos alunos testarem recursos da linguagem enquanto estão programando. Os mesmos podem manter uma janela com o interpretador executado enquanto digitam o fonte do seu programa numa outra janela. Se eles não conseguirem se lembrar dos métodos de uma lista, eles podem fazer algo assim:

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
↪'reverse', 'sort']

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

>>> L.append(1)
>>> L
[1]
```

Com o interpretador, a documentação nunca está longe do aluno quando estão programando.

Há também boas IDEs para o Python. O IDLE é uma IDE multiplataforma para o Python e que foi escrito em Python usando o Tkinter. Os usuários do Emacs estarão felizes em saber que existe um ótimo modo Python para Emacs. Todos esses ambientes de programação fornecem destaque de sintaxe, recuo automático e acesso ao interpretador interativo durante o tempo de desenvolvimento. Consulte o wiki do Python para obter uma lista completa dos ambientes de desenvolvimento para o Python.

Se você quiser discutir o uso do Python na educação, poderás estar interessado em se juntar à lista de discussão edu-sig.

CAPÍTULO 2

FAQ referente a Programação

2.1 Questões Gerais

2.1.1 Existe um depurador a nível de código-fonte que possuo pontos de interrupção (breakpoints), single-stepping etc.?

Sim.

Vários depuradores para Python estão descritos abaixo, e a função embutida `breakpoint()` permite que você caia em qualquer um deles.

O módulo pdb é um depurador cujo funcionamento ocorre em modo Console simples mas, adequado para o Python. Faz parte da biblioteca padrão do Python e está documentado em `documented in the Library Reference Manual 1`. Caso necessário, também é possível a construção do seu próprio depurador usando o código do pdb como um exemplo.

A IDLE (Ambiente Interativo de Desenvolvimento), que faz parte da distribuição padrão do Python (normalmente acessível como Tools/scripts/idle3), inclui um depurador gráfico

O PythonWin é uma IDE feita para o Python que inclui um depurador de GUI baseado no pdb. O depurador PythonWin colora os pontos de interrupção e tem alguns recursos legais, como a depuração de programas que não são PythonWin. O PythonWin está disponível como parte do projeto pywin32 e como parte da distribuição ActivePython.

Eric é uma IDE construída com PyQt e o componente de edição Scintilla.

trepan3k é um depurador similar ao gdb.

Visual Studio Code é uma IDE com ferramentas de depuração que integra com softwares de controle de versão.

Há uma série de IDE comerciais para desenvolvimento com o Python que incluem depuradores gráficos. Dentre tantas temos:

- Wing IDE
- Komodo IDE
- PyCharm

2.1.2 Existem ferramentas para ajudar a encontrar bugs ou fazer análise estática de desempenho?

Sim.

Pylint and Pyflakes do basic checking that will help you catch bugs sooner.

Static type checkers such as Mypy, Pyre, and Pytype can check type hints in Python source code.

2.1.3 Como posso criar um binário independente a partir de um script Python?

Você não precisa possuir a capacidade de compilar o código Python para C se o que deseja é um programa autônomo que os usuários possam baixar e executar sem ter que instalar a distribuição Python primeiro. Existem várias ferramentas que determinam o conjunto de módulos exigidos por um programa e vinculam esses módulos junto com o binário do Python para produzir um único executável.

One is to use the freeze tool, which is included in the Python source tree as Tools/freeze. It converts Python byte code to C arrays; with a C compiler you can embed all your modules into a new program, which is then linked with the standard Python modules.

Ela funciona escaneando seu código recursivamente pelas instruções de importação (ambas as formas) e procurando pelos módulos no caminho padrão do Python e também no diretório fonte (para módulos embutidos). Então torna o bytecode de módulos escritos em Python em código C (inicializadores de vetor que podem ser transformado em objetos código usando o módulo marshal) e cria um arquivo de configurações customizado que só contém aqueles módulos embutidos que são na realidade usados no programa. A ferramenta então compila os códigos gerados em C e liga como o resto do interpretador Python para formar um binário autônomo que age exatamente como seu script.

Os pacotes a seguir podem ajudar com a criação dos executáveis do console e da GUI:

- Nuitka (Multiplataforma)
- PyInstaller (Cross-platform)
- PyOxidizer (Multiplataforma)
- cx_Freeze (Multiplataforma)
- py2app (somente macOS)
- py2exe (Windows only)

2.1.4 Existem padrões para a codificação ou um guia de estilo utilizado pela comunidade Python?

Sim. O guia de estilo esperado para módulos e biblioteca padrão possui o nome de PEP8 e podes acessar a sua documentação em **PEP 8**.

2.2 Núcleo da Linguagem

2.2.1 Porque estou recebo o erro UnboundLocalError quando a variável possui um valor associado?

It can be a surprise to get the `UnboundLocalError` in previously working code when it is modified by adding an assignment statement somewhere in the body of a function.

Este código:

```
>>> x = 10
>>> def bar():
...     print(x)
...
>>> bar()
10
```

funciona, mas este código:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

resulta em uma `UnboundLocalError`:

```
>>> foo()
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'x' referenced before assignment
```

Isso acontece devido ao fato de que quando realizamos uma tarefa numa variável de um determinado escopo, essa variável torna-se-á local desse escopo acabando por esconder qualquer variável similar que foi mencionada no escopo externo. Uma vez que a última declaração de foo atribuir um novo valor a `x`, o compilador o reconhecera como uma variável local. Consequentemente, quando o `print(x)` anterior tentar imprimir a variável local não inicializada acabará resultando num.

No exemplo acima, podemos acessar a variável do escopo externo declarando-o globalmente:

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
...
>>> foobar()
10
```

Esta declaração explícita é necessária para lembrá-lo de que (ao contrário da situação superficialmente análoga com variáveis de classe e instância), você realmente está modificando o valor da variável no escopo externo:

```
>>> print(x)
11
```

Você pode fazer uma coisa semelhante num escopo aninhado usando o argumento nomeado `nonlocal`:

```
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
...
>>> foo()
10
11
```

2.2.2 Quais são as regras para variáveis locais e globais em Python?

Em Python, as variáveis que são apenas utilizadas (referenciadas) dentro de uma função são implicitamente globais. Se uma variável for associada a um valor em qualquer lugar dentro do corpo da função, presume-se que a mesma será local, a menos que seja explicitamente declarado como global.

Embora um pouco surpreendente no início, um momento de consideração explica isso. Por um lado, exigir `global` para variáveis atribuídas fornece uma barreira contra efeitos colaterais indesejados. Por outro lado, se `global` fosse

necessário para todas as referências globais, você estaria usando `global` o tempo todo. Você teria que declarar como global todas as referências a uma função embutida ou a um componente de um módulo importado. Essa desordem anularia a utilidade da declaração de `global` para identificar efeitos colaterais.

2.2.3 Por que os lambdas definidos em um loop com valores diferentes retornam o mesmo resultado?

Suponha que utilizes um loop for para definir algumas funções lambdas (ou mesmo funções simples), por exemplo.:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

Isso oferece uma lista que contém 5 lambdas que calculam `x**2`. Poderás esperar que, quando invocado, os mesmo retornem, respectivamente, `0`, `1`, `4`, `9`, e `16`. No entanto, quando realmente tentares, verás que todos retornam `16`:

```
>>> squares[2]()
16
>>> squares[4]()
16
```

Isso acontece porque `x` não é local para o lambdas, mas é definido no escopo externo, e é acessado quando o lambda for chamado — não quando é definido. No final do loop, o valor de `x` será `4`, e então, todas as funções agora retornarão `4**2`, ou seja, `16`. Também poderás verificar isso alterando o valor de `x` e vendo como os resultados dos lambdas mudam:

```
>>> x = 8
>>> squares[2]()
64
```

Para evitar isso, precisarás salvar os valores nas variáveis locais para os lambdas, para que eles não dependam do valor de `x` global:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Aqui, `n=x` cria uma nova variável `n` local para o lambda e calculada quando o lambda será definido para que ele tenha o mesmo valor que `x` tenha nesse ponto no loop. Isso significa que o valor de `n` será `0` no primeiro "ciclo" do lambda, `1` no segundo "ciclo", `2` no terceiro, e assim por diante. Portanto, cada lambda agora retornará o resultado correto:

```
>>> squares[2]()
4
>>> squares[4]()
16
```

Observe que esse comportamento não é peculiar dos lambdas, o mesmo também ocorre com as funções regulares.

2.2.4 Como definir variáveis globais dentro de módulos?

A maneira canônica de compartilhar informações entre módulos dentro de um único programa é criando um módulo especial (geralmente chamado de config ou cfg). Basta importar o módulo de configuração em todos os módulos da sua aplicação; O módulo ficará disponível como um nome global. Como há apenas uma instância de cada módulo, todas as alterações feitas no objeto do módulo se refletem em todos os lugares. Por exemplo:

config.py:

```
x = 0   # Default value of the 'x' configuration setting
```

mod.py:

```
import config
config.x = 1
```

main.py:

```
import config
import mod
print(config.x)
```

Note que usar um módulo também é a base para a implementação de padrões de design singleton, pela mesma razão.

2.2.5 Quais são as "melhores práticas" quando fazemos uso da importação de módulos?

Em geral, não use `from modulename import *`. Ao fazê-lo, o espaço de nomes do importador é mais difícil e torna muito mais difícil para as ferramentas linters detectar nomes indefinidos.

Faça a importação de módulos na parte superior do arquivo. Isso deixa claro quais outros módulos nosso código necessita e evita dúvidas sobre por exemplo, se o nome do módulo está no escopo. Usar uma importação por linha facilita a adição e exclusão de importações de módulos, porém, usar várias importações num única linha, ocupa menos espaço da tela.

É uma boa prática importar os módulos na seguinte ordem:

1. módulos padrões da biblioteca – por exemplo: `sys`, `os`, `argparse`, `re`
2. third-party library modules (anything installed in Python's site-packages directory) – e.g. `dateutil`, `requests`, `PIL.Image`
3. módulos desenvolvidos localmente

Às vezes, é necessário transferir as importações para uma função ou classe para evitar problemas com importação circular. Gordon McMillan diz:

> As importações circulares estão bem onde ambos os módulos utilizam a forma de importação "import 1". Eles falham quando o 2º módulo quer pegar um nome do primeiro ("from module import name") e a importação está no nível superior. Isso porque os nomes no primeiro ainda não estão disponíveis, porque o primeiro módulo está ocupado importando o 2º.

Nesse caso, se o segundo módulo for usado apenas numa função, a importação pode ser facilmente movida para dentro do escopo dessa função. No momento em que a importação for chamada, o primeiro módulo terá finalizado a inicialização e o segundo módulo poderá ser importado sem maiores complicações.

Também poderá ser necessário mover as importações para fora do nível superior do código se alguns dos módulos forem específicos de uma determinada plataforma (SO). Nesse caso, talvez nem seja possível importar todos os módulos na parte superior do arquivo. Nessas situações devemos importar os módulos que são específicos de cada plataforma antes de necessitar utilizar os mesmos.

Apenas mova as importações para um escopo local, como dentro da definição de função, se for necessário resolver algum tipo de problema, como exemplo, evitar importações circulares ou tentar reduzir o tempo de inicialização do módulo. Esta técnica é especialmente útil se muitas das importações forem desnecessárias, dependendo de como o programa é executado. Também podemos desejar mover as importações para uma função se os módulos forem usados somente nessa função. Note que carregar um módulo pela primeira vez pode ser demorado devido ao tempo de inicialização de cada módulo, no entanto, carregar um módulo várias vezes é praticamente imperceptível, tendo somente o custo de processamento de pesquisas no dicionário de nomes. Mesmo que o nome do módulo tenha saído do escopo, o módulo provavelmente estará disponível em `sys.modules`.

2.2.6 Por que os valores padrão são compartilhados entre objetos?

Este tipo de erro geralmente pega programadores neófitos. Considere esta função:

```
def foo(mydict={}):  # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

A primeira vez que chamares essa função, `mydict` irá conter um único item. A segunda vez, `mydict` irá conter dois itens, porque quando `foo()` começar a ser executado, `mydict` começará com um item já existente.

Muitas vezes, espera-se que ao invocar uma função seja criado novos objetos referente aos valores padrão. Isso não é o que acontecerá. Os valores padrão são criados exatamente uma vez, quando a função está sendo definida. Se esse objeto for alterado, como o dicionário neste exemplo, as chamadas subsequentes para a essa função se referirão a este objeto alterado.

Por definição, objetos imutáveis, como números, strings, tuplas e o `None`, estão protegidos de sofrerem alteração. Alterações em objetos mutáveis, como dicionários, listas e instâncias de classe, podem levar à confusão.

Por causa desse recurso, é uma boa prática de programação para evitar o uso de objetos mutáveis contendo valores padrão. Em vez disso, utilize `None` como o valor padrão e dentro da função, verifique se o parâmetro é `None` e crie uma nova lista /dicionário/ o que quer que seja. Por exemplo, escreva o seguinte código:

```
def foo(mydict={}):
    ...
```

mas:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # create a new dict for local namespace
```

Esse recurso pode ser útil. Quando tiveres uma função que consome muito tempo para calcular, uma técnica comum é armazenar em cache os parâmetros e o valor resultante de cada chamada para a função e retornar o valor em cache se o mesmo valor for solicitado novamente. Isso se chama "memoizar", e pode ser implementado da seguinte forma:

```
# Callers can only provide two parameters and optionally pass _cache by keyword
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result           # Store result in the cache
    return result
```

Poderias usar uma variável global contendo um dicionário ao invés do valor padrão; isso é uma questão de gosto.

2.2.7 Como passar parâmetros opcionais ou parâmetros na forma de keyword de uma função a outra?

Colete os argumentos usando os especificadores `*` ou `**` na lista de parâmetros da função. Isso faz com que os argumentos posicionais como tupla e os argumentos nomeados sejam passados como um dicionário. Você pode, também, passar esses argumentos ao invocar outra função usando `*` e `**`:

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

2.2.8 Qual a diferença entre argumentos e parâmetros?

Parameters are defined by the names that appear in a function definition, whereas *arguments* are the values actually passed to a function when calling it. Parameters define what *kind of arguments* a function can accept. For example, given the function definition:

```
def func(foo, bar=None, **kwargs):
    pass
```

foo, *bar* e *kwargs* são parâmetros de `func`. Dessa forma, ao invocar `func`, por exemplo:

```
func(42, bar=314, extra=somevar)
```

os valores `42`, `314`, e `somevar` são os argumentos.

2.2.9 Por que ao alterar a lista 'y' também altera a lista 'x'?

Se escreveres um código como:

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

Poderás estar se perguntando por que acrescentar um elemento a `y` também mudou `x`.

Há 2 fatores que produzem esse resultado, são eles:

1) As variáveis são simplesmente nomes que referem-se a objetos. Ao escrevermos `y = x` não criará uma cópia da lista – criará uma nova variável `y` que irá se referir ao mesmo objeto que `x` está se referindo. Isso significa que existe apenas um objeto (lista), e ambos nomes (variáveis) `x` e `y` estão associados ao mesmo.

2) Listas são objetos *mutáveis*, o que significa que você pode alterar o seu conteúdo.

After the call to `append()`, the content of the mutable object has changed from `[]` to `[10]`. Since both the variables refer to the same object, using either name accesses the modified value `[10]`.

Se por acaso, atribuímos um objeto imutável a `x`:

```
>>> x = 5  # ints are immutable
>>> y = x
>>> x = x + 1  # 5 can't be mutated, we are creating a new object here
>>> x
6
>>> y
5
```

veremos que nesse caso `x` e `y` não são mais iguais. Isso ocorre porque os números inteiros são *imutáveis*, e quando fizermos `x = x + 1` não estaremos mudando o int `5` e incrementando o seu valor; em vez disso, estamos criando um novo objeto (o int `6`) e associando `x` (isto é, mudando para o objeto no qual `x` se refere). Após esta atribuição, temos dois objetos (os ints `6` e `5`) e duas variáveis que referem-se a elas (`x` agora se refere a `6` mas `y` ainda refere-se a `5`).

Some operations (for example `y.append(10)` and `y.sort()`) mutate the object, whereas superficially similar operations (for example `y = y + [10]` and `sorted(y)`) create a new object. In general in Python (and in all cases in the standard library) a method that mutates an object will return `None` to help avoid getting the two types of operations confused. So if you mistakenly write `y.sort()` thinking it will give you a sorted copy of `y`, you'll instead end up with `None`, which will likely cause your program to generate an easily diagnosed error.

No entanto, há uma classe de operações em que a mesma operação às vezes tem comportamentos diferentes com tipos diferentes: os operadores de atribuição aumentada. Por exemplo, += transforma listas, mas não tuplas ou ints (`a_list += [1, 2, 3]` é equivalente a `a_list.extend([1, 2, 3])` a altera `a_list`, sendo que `some_tuple += (1, 2, 3)` e `some_int += 1` cria novos objetos).

Em outras palavras:

- Se tivermos objetos mutáveis (`list`, `dict`, `set`, etc.), podemos usar algumas operações específicas para altera-lo e todas as variáveis que se referem a ela sofreram também a mudança.
- Caso tenhamos um objeto imutável (`str`, `int`, `tuple`, etc.), todas as variáveis que se referem as mesmas sempre verão o mesmo valor, mas as operações que transformam-se nesses valores sempre retornarão novos objetos.

Se quiseres saber se duas variáveis se referem ao mesmo objeto ou não, podes usar a palavra-chave `is`, ou a função embutida `id()`.

2.2.10 Como escrever uma função com parâmetros de saída (invocada por referência)?

Lembre-se de que os argumentos são passados por atribuição em Python. Uma vez que a tarefa apenas cria referências a objetos, não existe "alias" entre um nome de argumento naquele que invocado e o destinatário, portanto, não há referência de chamada por si. Podes alcançar o efeito desejado de várias maneiras.

1) Retornando um Tupla com os resultados:

```
>>> def func1(a, b):
...     a = 'new-value'         # a and b are local names
...     b = b + 1               # assigned to new objects
...     return a, b             # return new values
...
>>> x, y = 'old-value', 99
>>> func1(x, y)
('new-value', 100)
```

   Esta quase sempre é a solução mais clara.

2) Utilizando variáveis globais. Essa forma de trabalho não é segura para uso com thread e portanto, a mesma não é recomendada.

3) Pela passagem de um objeto mutável (que possa ser alterado internamente):

```
>>> def func2(a):
...     a[0] = 'new-value'      # 'a' references a mutable list
...     a[1] = a[1] + 1         # changes a shared object
...
>>> args = ['old-value', 99]
>>> func2(args)
>>> args
['new-value', 100]
```

4) Pela passagem de um dicionário que seja mutável:

```
>>> def func3(args):
...     args['a'] = 'new-value'     # args is a mutable dictionary
...     args['b'] = args['b'] + 1   # change it in-place
...
>>> args = {'a': 'old-value', 'b': 99}
>>> func3(args)
>>> args
{'a': 'new-value', 'b': 100}
```

5) Ou agrupando valores numa instância de classe:

```
>>> class Namespace:
...     def __init__(self, /, **args):
...         for key, value in args.items():
...             setattr(self, key, value)
...
>>> def func4(args):
...     args.a = 'new-value'        # args is a mutable Namespace
...     args.b = args.b + 1         # change object in-place
...
>>> args = Namespace(a='old-value', b=99)
>>> func4(args)
>>> vars(args)
{'a': 'new-value', 'b': 100}
```

Quase nunca existe uma boa razão para complicar isso.

A sua melhor escolha será retornar uma Tupla contendo os múltiplos resultados.

2.2.11 Como fazer uma função de ordem superior em Python?

Existem duas opções: podes usar escopos aninhados ou poderás usar objetos chamáveis. Por exemplo, suponha que desejasses definir que `linear(a,b)` retorne uma função `f(x)` que calcule o valor `a*x+b`. Usando escopos aninhados, temos:

```
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

Ou utilizando objetos chamáveis:

```
class linear:

    def __init__(self, a, b):
        self.a, self.b = a, b

    def __call__(self, x):
        return self.a * x + self.b
```

Em ambos os casos:

```
taxes = linear(0.3, 2)
```

dado um objeto chamável, onde `taxes(10e6) == 0.3 * 10e6 + 2`.

A abordagem do objeto chamável tem a desvantagem de que é um pouco mais lento e resulta num código ligeiramente mais longo. No entanto, note que uma coleção de chamáveis pode compartilhar sua assinatura via herança:

```
class exponential(linear):
    # __init__ inherited
    def __call__(self, x):
        return self.a * (x ** self.b)
```

Objetos podem encapsular o estado para vários métodos:

```
class counter:

    value = 0
```

(continua na próxima página)

(continuação da página anterior)

```
    def set(self, x):
        self.value = x

    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

Aqui `inc()`, `dec()` e `reset()` funcionam como funções que compartilham a mesma variável contadora.

2.2.12 Como faço para copiar um objeto no Python?

Basicamente, tente utilizar a função `copy.copy()` ou a função `copy.deepcopy()` para casos gerais. Nem todos os objetos podem ser copiados, mas a maioria poderá.

Alguns objetos podem ser copiados com mais facilidade. Os dicionários têm um método `copy()`:

```
newdict = olddict.copy()
```

As sequências podem ser copiadas através do uso de fatiamento:

```
new_l = l[:]
```

2.2.13 Como posso encontrar os métodos ou atributos de um objeto?

For an instance `x` of a user-defined class, `dir(x)` returns an alphabetized list of the names containing the instance attributes and methods and attributes defined by its class.

2.2.14 Como que o meu código pode descobrir o nome de um objeto?

De um modo geral, não pode, porque os objetos realmente não têm nomes. Essencialmente, a atribuição sempre liga um nome a um valor; o mesmo é verdade para as instruções `def` e `class`, mas nesse caso o valor é um chamável. Considere o seguinte código:

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print(b)
<__main__.A object at 0x16D07CC>
>>> print(a)
<__main__.A object at 0x16D07CC>
```

Arguably the class has a name: even though it is bound to two names and invoked through the name `B` the created instance is still reported as an instance of class `A`. However, it is impossible to say whether the instance's name is `a` or `b`, since both names are bound to the same value.

De um modo geral, não deveria ser necessário que o seu código "conheça os nomes" de valores específicos. A menos que escrevas deliberadamente programas introspectivos, isso geralmente é uma indicação de que uma mudança de abordagem pode ser benéfica.

Em comp.lang.python, Fredrik Lundh deu uma excelente analogia em resposta a esta pergunta:

Da mesma forma que você pega o nome daquele gato que encontrou na sua varanda: o próprio gato (objeto) não pode lhe dizer o seu nome, e ele realmente não se importa – então a única maneira de descobrir como ele se chama é perguntar a todos os seus vizinhos (espaços de nomes) se é o gato deles (objeto)…

….e não fique surpreso se você encontrar que é conhecido por muitos nomes, ou até mesmo nenhum nome.

2.2.15 O que há com a precedência do operador vírgula?

A vírgula não é um operador em Python. Considere este código:

```
>>> "a" in "b", "a"
(False, 'a')
```

Uma vez que a vírgula não seja um operador, mas um separador entre as expressões acima, o código será avaliado como se tivéssemos entrado:

```
("a" in "b"), "a"
```

não:

```
"a" in ("b", "a")
```

O mesmo é verdade para as várias operações de atribuição (=, += etc). Eles não são operadores de verdade mas delimitadores sintáticos em instruções de atribuição.

2.2.16 Existe um equivalente ao operador "?:" ternário do C?

Sim existe. A sintaxe é a seguinte:

```
[on_true] if [expression] else [on_false]

x, y = 50, 25
small = x if x < y else y
```

Antes que essa sintaxe fosse introduzida no Python 2.5, um idioma comum era usar operadores lógicos:

```
[expression] and [on_true] or [on_false]
```

No entanto, essa forma não é segura, pois pode dar resultados inesperados quando *on_true* possuir um valor booleano falso. Portanto, é sempre melhor usar a forma `... if ... else ...`.

2.2.17 É possível escrever instruções de uma só linha ofuscadas em Python?

Yes. Usually this is done by nesting `lambda` within `lambda`. See the following three examples, slightly adapted from Ulf Bartelt:

```
from functools import reduce

# Primes < 1000
print(list(filter(None,map(lambda y:y*reduce(lambda x,y:x*y!=0,
map(lambda x,y=y:y%x,range(2,int(pow(y,0.5)+1))),1),range(2,1000)))))

# First 10 Fibonacci numbers
print(list(map(lambda x,f=lambda x,f:(f(x-1,f)+f(x-2,f)) if x>1 else 1:
f(x,f), range(10))))

# Mandelbrot set
```

(continua na próxima página)

(continuação da página anterior)

```
print((lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda x,y:x+'\n'+y,map(lambda y,
Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,i=IM,
Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda x,xc=Ru,yc=yc,Ru=Ru,Ro=Ro,
i=i,Sx=Sx,F=lambda xc,yc,x,y,k,f=lambda xc,yc,x,y,k,f:(k<=0)or (x*x+y*y
>=4.0) or 1+f(xc,yc,x*x-y*y+xc,2.0*x*y+yc,k-1,f):f(xc,yc,x,y,k,f):chr(
64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))):L(Iu+y*(Io-Iu)/Sy),range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24))
#     \___ ___/  \___ ___/  |   |   |__ lines on screen
#         V          V      |   |______ columns on screen
#         |          |      |_________ maximum of "iterations"
#         |          |________________ range on y axis
#         |_______________________________ range on x axis
```

Não tente isso em casa, crianças!

2.2.18 O que a barra(/) na lista de parâmetros de uma função significa?

Uma barra na lista de argumentos de uma função indica que os parâmetros anteriores a ela são somente-posicionais. Os parâmetros somente-posicionais são aqueles que não têm nome utilizável externamente. Ao chamar uma função que aceita parâmetros somente-posicionais, os argumentos são mapeados para parâmetros com base apenas em sua posição. Por exemplo, `divmod()` é uma função que aceita parâmetros somente-posicionais. Sua documentação tem esta forma:

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

A barra no final da lista de parâmetros significa que ambos os parâmetros são somente-posicionais. Assim, chamar `divmod()` com argumentos nomeados levaria a um erro:

```
>>> divmod(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod() takes no keyword arguments
```

2.3 Números e Strings

2.3.1 Como faço para especificar números inteiros hexadecimais e octal?

Para especificar um dígito no formato octal, preceda o valor octal com um zero e, em seguida, um "o" minúsculo ou maiúsculo. Por exemplo, para definir a variável "a" para o valor octal "10" (8 em decimal), digite:

```
>>> a = 0o10
>>> a
8
```

Hexadecimal é bem fácil. Basta preceder o número hexadecimal com um zero e, em seguida, um "x" minúsculo ou maiúsculo. Os dígitos hexadecimais podem ser especificados em letras maiúsculas e minúsculas. Por exemplo, no interpretador Python:

```
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
```

(continua na próxima página)

(continuação da página anterior)

```
>>> b
178
```

2.3.2 Por que -22 // 10 retorna -3?

É principalmente direcionado pelo desejo de que `i % j` possui o mesmo sinal que `j`. Se quiseres isso, e também se desejares:

```
i == (i // j) * j + (i % j)
```

então a divisão inteira deve retornar o piso. C também requer que essa identidade seja mantida, e então os compiladores que truncarem `i // j` precisam fazer com que `i % j` tenham o mesmo sinal que `i`.

Existem poucos casos de uso reais para `i % j` quando `j` é negativo. Quando `j` é positivo, existem muitos, e em virtualmente todos eles é mais útil para `i % j` ser `>= 0`. Se o relógio marca 10 agora, o que dizia há 200 horas? `-190 % 12 == 2` é útil, enquanto `-190 % 12 == -10` é um bug esperando para morder.

2.3.3 How do I get int literal attribute instead of SyntaxError?

Trying to lookup an `int` literal attribute in the normal manner gives a `SyntaxError` because the period is seen as a decimal point:

```
>>> 1.__class__
  File "<stdin>", line 1
  1.__class__
   ^
SyntaxError: invalid decimal literal
```

A solução é separar o literal do ponto-final com um espaço ou parênteses.

```
>>> 1 .__class__
<class 'int'>
>>> (1).__class__
<class 'int'>
```

2.3.4 Como faço para converter uma String em um número?

Para inteiros, use o tipo embutido `int()`, por exemplo, `int('144') == 144`. Da mesma forma, `float()` converterá para um valor do tipo ponto flutuante, por exemplo `float('144') == 144.0`.

Por padrão, eles interpretam o número como decimal, de modo que `int('0144') == 144` é verdadeiro e `int('0x144')` levanta `ValueError`. `int(string, base)` toma a base para converter como um segundo argumento opcional, então `int( '0x144', 16) == 324`. Se a base for especificada como 0, o número é interpretado usando as regras do Python: um "0o" à esquerda indica octal e "0x" indica um número hexadecimal.

Não use a função embutida `eval()` se tudo que você precisa é converter strings em números. `eval()` será significativamente mais lento e apresenta um risco de segurança: alguém pode passar a você uma expressão Python que pode ter efeitos colaterais indesejados. Por exemplo, alguém poderia passar `__import__('os').system("rm -rf $HOME")` que apagaria seu diretório pessoal.

`eval()` também tem o efeito de interpretar números como expressões Python, para que, por exemplo, `eval('09')` dá um erro de sintaxe porque Python não permite '0' inicial em um número decimal (exceto '0').

2.3.5 Como faço para converter um número numa string?

To convert, e.g., the number `144` to the string `'144'`, use the built-in type constructor `str()`. If you want a hexadecimal or octal representation, use the built-in functions `hex()` or `oct()`. For fancy formatting, see the f-strings and formatstrings sections, e.g. `"{:04d}".format(144)` yields `'0144'` and `"{:.3f}".format(1.0/3.0)` yields `'0.333'`.

2.3.6 Como faço para modificar uma string no lugar?

Você não poder fazer isso as Strings são objetos imutáveis. Na maioria das situações, você simplesmente deve construir uma nova string a partir das várias partes das quais desejas que a sua nova String tenha. No entanto, se precisares de um objeto com a capacidade de modificar dados Unicode internamente, tente usar a classe `io.StringIO` ou o módulo `array`:

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a = array.array('u', s)
>>> print(a)
array('u', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('u', 'yello, world')
>>> a.tounicode()
'yello, world'
```

2.3.7 Como faço para invocar funções/métodos através de uma String?

Existem várias técnicas.

- A melhor forma é usar um dicionário que mapeie a Strings para funções. A principal vantagem desta técnica é que as Strings não precisam combinar os nomes das funções. Esta é também a principal técnica utilizada para emular uma construção de maiúsculas e minúsculas

  ```
  def a():
      pass

  def b():
      pass

  dispatch = {'go': a, 'stop': b}  # Note lack of parens for funcs

  dispatch[get_input()]()  # Note trailing parens to call function
  ```

- Utilize a função embutida `getattr()`:

  ```
  import foo
  getattr(foo, 'bar')()
  ```

  Observe que a função `getattr()` funciona com qualquer objeto, incluindo classes, instâncias de classe, módulos e assim por diante.

  A mesma é usado em vários lugares na biblioteca padrão, como este:

  ```
  class Foo:
      def do_foo(self):
  ```

(continua na próxima página)

(continuação da página anterior)

```
        ...

    def do_bar(self):
        ...

f = getattr(foo_instance, 'do_' + opname)
f()
```

- Use `locals()` para determinar o nome da função:

```
def myFunc():
    print("hello")

fname = "myFunc"

f = locals()[fname]
f()
```

2.3.8 Existe um equivalente em Perl `chomp()` para remover linhas novas de uma String?

Podes utilizar `S.rstrip("\r\n")` para remover todas as ocorrência de qualquer terminador de linha que esteja no final da string `S` sem remover os espaços em branco. Se a string `S` representar mais de uma linha, contendo várias linhas vazias no final, os terminadores de linha de todas linhas em branco serão removidos:

```
>>> lines = ("line 1 \r\n"
...          "\r\n"
...          "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```

Geralmente isso só é desejado ao ler um texto linha por linha, usando `S.rstrip()` dessa maneira funciona bem.

2.3.9 Existe uma função scanf() ou sscanf() ou algo equivalente?

Não como tal.

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using the `split()` method of string objects and then convert decimal strings to numeric values using `int()` or `float()`. `split()` supports an optional "sep" parameter which is useful if the line uses something other than whitespace as a separator.

For more complicated input parsing, regular expressions are more powerful than C's `sscanf` and better suited for the task.

2.3.10 O que significa o erro 'UnicodeDecodeError' ou 'UnicodeEncodeError'?

Consulte unicode-howto.

2.3.11 Can I end a raw string with an odd number of backslashes?

A raw string ending with an odd number of backslashes will escape the string's quote:

```
>>> r'C:\this\will\not\work\'
  File "<stdin>", line 1
    r'C:\this\will\not\work\'
         ^
SyntaxError: unterminated string literal (detected at line 1)
```

There are several workarounds for this. One is to use regular strings and double the backslashes:

```
>>> 'C:\\this\\will\\work\\'
'C:\\this\\will\\work\\'
```

Another is to concatenate a regular string containing an escaped backslash to the raw string:

```
>>> r'C:\this\will\work' '\\'
'C:\\this\\will\\work\\'
```

It is also possible to use `os.path.join()` to append a backslash on Windows:

```
>>> os.path.join(r'C:\this\will\work', '')
'C:\\this\\will\\work\\'
```

Note that while a backslash will "escape" a quote for the purposes of determining where the raw string ends, no escaping occurs when interpreting the value of the raw string. That is, the backslash remains present in the value of the raw string:

```
>>> r'backslash\'preserved'
"backslash\\'preserved"
```

Also see the specification in the language reference.

2.4 Desempenho

2.4.1 Meu programa está muito lento. Como faço para melhorar o desempenho?

Isso geralmente é algo difícil de conseguir. Primeiro, aqui está uma lista de situações que devemos lembrar para melhorar o desempenho da nossa aplicação antes de buscarmos outras soluções:

- As características da desempenho podem variar conforme a implementação do Python. Esse FAQ foca em *CPython*.
- O comportamento pode variar em cada Sistemas Operacionais, especialmente quando estivermos tratando de I/o ou multi-threading.
- Sempre devemos encontrar os hot spots em nosso programa *antes de* tentar otimizar qualquer código (veja o módulo `profile`).
- Escrever Scripts de benchmark permitirá iterar rapidamente buscando melhorias (veja o módulo `timeit`).
- É altamente recomendável ter boa cobertura de código (através de testes de unidade ou qualquer outra técnica) antes de potencialmente apresentar regressões escondidas em otimizações sofisticadas.

Dito isto, existem muitos truques para acelerar nossos códigos Python. Aqui estão alguns dos principais tópicos e que geralmente ajudam a atingir níveis de desempenho aceitáveis:

- Fazer seus algoritmos rápidos (ou mudando para mais rápidos) podem produzir benefícios maiores que tentar encaixar várias micro-otimizações no seu código.
- Use as estruturas de dados corretas. Documentação de estudo para bltin-types e o módulo `collections`.
- Quando a biblioteca padrão fornecer um tipo primitivo para fazer algo, é provável (embora não garantido) que este seja mais rápido do que qualquer alternativa que possa surgir. Isso geralmente é verdade para os tipos primitivos escritos em C, como os embutidos e alguns tipos de extensão. Por exemplo, certifique-se de usar o método embutido `list.sort()` ou a função relacionada `sorted()` para fazer a ordenação (e veja sortinghowto para exemplos de uso moderadamente avançado).
- As abstrações tendem a criar indireções e forçar o interpretador a trabalhar mais. Se os níveis de indireção superarem a quantidade de trabalho útil feito, seu programa ficará mais lento. Você deve evitar a abstração excessiva, especialmente sob a forma de pequenas funções ou métodos (que também são muitas vezes prejudiciais à legibilidade).

If you have reached the limit of what pure Python can allow, there are tools to take you further away. For example, Cython can compile a slightly modified version of Python code into a C extension, and can be used on many different platforms. Cython can take advantage of compilation (and optional type annotations) to make your code significantly faster than when interpreted. If you are confident in your C programming skills, you can also write a C extension module yourself.

Ver também

A página wiki dedicada a dicas de desempenho.

2.4.2 Qual é a maneira mais eficiente de concatenar muitas Strings?

A classe `str` e a classe `bytes` são objetos imutáveis, portanto, concatenar muitas Strings em é ineficiente, pois cada concatenação criará um novo objeto String. No caso geral, o custo total do tempo de execução é quadrático no comprimento total da String.

Para juntar vários objetos `str`, a linguagem recomendada colocá-los numa lista e invocar o método `str.join()`:

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

(outra forma razoavelmente eficiente é usar a classe `io.StringIO`)

Para juntar vários objetos `bytes`, a forma recomendada é estender uma classe `bytearray` usando a concatenação local (com o operador +=):

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

2.5 Sequencias (Tuples/Lists)

2.5.1 Como faço para converter tuplas em listas?

O construtor de tipo `tuple(seq)` converte qualquer sequência (na verdade, qualquer iterável) numa tupla com os mesmos itens na mesma ordem.

Por exemplo, `tuple([1, 2, 3])` produz `(1, 2, 3)` e `tuple('abc')` produz `('a', 'b', 'c')`. Se o argumento for uma tupla, a mesma não faz uma cópia, mas retorna o mesmo objeto, por isso é barato invocar a função `tuple()` quando você não tiver certeza que determinado objeto já é uma tupla.

O construtor de tipos `list(seq)` converte qualquer sequência ou iterável em uma lista com os mesmos itens na mesma ordem. Por exemplo, `list((1, 2, 3))` produz `[1, 2, 3]` e `list('abc')` produz `['a', 'b', 'c']`. Se o argumento for uma lista, o meso fará uma cópia como em `seq[:]`.

2.5.2 O que é um índice negativo?

Python sequences are indexed with positive numbers and negative numbers. For positive numbers 0 is the first index 1 is the second index and so forth. For negative indices -1 is the last index and -2 is the penultimate (next to last) index and so forth. Think of `seq[-n]` as the same as `seq[len(seq)-n]`.

Using negative indices can be very convenient. For example `S[:-1]` is all of the string except for its last character, which is useful for removing the trailing newline from a string.

2.5.3 Como que eu itero uma sequência na ordem inversa?

Use a função embutida `reversed()`:

```
for x in reversed(sequence):
    ...  # do something with x ...
```

Isso não vai alterar sua sequência original, mas construir uma nova cópia com a ordem inversa para iteração.

2.5.4 Como que remove itens duplicados de uma lista?

See the Python Cookbook for a long discussion of many ways to do this:

https://code.activestate.com/recipes/52560/

If you don't mind reordering the list, sort it and then scan from the end of the list, deleting duplicates as you go:

```
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]
```

Se todos os elementos da lista podem ser usados como chaves de conjunto (isto é, eles são todos *hasheáveis*) isso é muitas vezes mais rápido

```
mylist = list(set(mylist))
```

Isso converte a lista em um conjunto, deste modo removendo itens duplicados, e depois de volta em uma lista.

2.5.5 Como remover múltiplos itens de uma lista

As with removing duplicates, explicitly iterating in reverse with a delete condition is one possibility. However, it is easier and faster to use slice replacement with an implicit or explicit forward iteration. Here are three variations.:

```
mylist[:] = filter(keep_function, mylist)
mylist[:] = (x for x in mylist if keep_condition)
mylist[:] = [x for x in mylist if keep_condition]
```

A compreensão de lista pode ser a mais rápida.

2.5.6 Como fazer um vetor em Python?

Utilize uma lista:

```
["this", 1, "is", "an", "array"]
```

Lists are equivalent to C or Pascal arrays in their time complexity; the primary difference is that a Python list can contain objects of many different types.

The `array` module also provides methods for creating arrays of fixed types with compact representations, but they are slower to index than lists. Also note that NumPy and other third party packages define array-like structures with various characteristics as well.

To get Lisp-style linked lists, you can emulate *cons cells* using tuples:

```
lisp_list = ("like",  ("this",  ("example", None) ) )
```

If mutability is desired, you could use lists instead of tuples. Here the analogue of a Lisp *car* is `lisp_list[0]` and the analogue of *cdr* is `lisp_list[1]`. Only do this if you're sure you really need to, because it's usually a lot slower than using Python lists.

2.5.7 Como faço para criar uma lista multidimensional?

Você provavelmente tentou fazer um Array multidimensional como isso:

```
>>> A = [[None] * 2] * 3
```

Isso parece correto se você imprimir:

```
>>> A
[[None, None], [None, None], [None, None]]
```

Mas quando atribuíres um valor, o mesmo aparecerá em vários lugares:

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

A razão é que replicar uma lista com `*` não cria cópias, ela apenas cria referências aos objetos existentes. O `*3` cria uma lista contendo 3 referências para a mesma lista que contém 2 itens cada. Mudanças numa linha serão mostradas em todas as linhas, o que certamente não é o que você deseja.

A abordagem sugerida é criar uma lista de comprimento desejado primeiro e, em seguida, preencher cada elemento com uma lista recém-criada:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

Isso gera uma lista contendo 3 listas diferentes contendo 2 itens cadas. Você também pode usar uma compreensão de lista:

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

Or, you can use an extension that provides a matrix datatype; NumPy is the best known.

2.5.8 How do I apply a method or function to a sequence of objects?

To call a method or function and accumulate the return values is a list, a *list comprehension* is an elegant solution:

```
result = [obj.method() for obj in mylist]

result = [function(obj) for obj in mylist]
```

To just run the method or function without saving the return values, a plain `for` loop will suffice:

```
for obj in mylist:
    obj.method()

for obj in mylist:
    function(obj)
```

2.5.9 Porque a_tuple[i] += ['item'] levanta uma exceção quando a adição funciona?

This is because of a combination of the fact that augmented assignment operators are *assignment* operators, and the difference between mutable and immutable objects in Python.

This discussion applies in general when augmented assignment operators are applied to elements of a tuple that point to mutable objects, but we'll use a `list` and `+=` as our exemplar.

Se você escrever:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The reason for the exception should be immediately clear: `1` is added to the object `a_tuple[0]` points to (`1`), producing the result object, `2`, but when we attempt to assign the result of the computation, `2`, to element `0` of the tuple, we get an error because we can't change what an element of a tuple points to.

Por baixo, o que a instrução de atribuição aumentada está fazendo é aproximadamente isso:

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

A parte da atribuição da operação que produz o erro, já que a tupla é imutável.

Quando você escreve algo como:

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The exception is a bit more surprising, and even more surprising is the fact that even though there was an error, the append worked:

```
>>> a_tuple[0]
['foo', 'item']
```

To see why this happens, you need to know that (a) if an object implements an `__iadd__()` magic method, it gets called when the `+=` augmented assignment is executed, and its return value is what gets used in the assignment statement; and (b) for lists, `__iadd__()` is equivalent to calling `extend()` on the list and returning the list. That's why we say that for lists, `+=` is a "shorthand" for `list.extend()`:

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

Isso equivale a:

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

The object pointed to by a_list has been mutated, and the pointer to the mutated object is assigned back to `a_list`. The end result of the assignment is a no-op, since it is a pointer to the same object that `a_list` was previously pointing to, but the assignment still happens.

Thus, in our tuple example what is happening is equivalent to:

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

The `__iadd__()` succeeds, and thus the list is extended, but even though `result` points to the same object that `a_tuple[0]` already points to, that final assignment still results in an error, because tuples are immutable.

2.5.10 I want to do a complicated sort: can you do a Schwartzian Transform in Python?

The technique, attributed to Randal Schwartz of the Perl community, sorts the elements of a list by a metric which maps each element to its "sort value". In Python, use the `key` argument for the `list.sort()` method:

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

2.5.11 Como eu posso ordenar uma lista pelos valores de outra lista?

Merge them into an iterator of tuples, sort the resulting list, and then pick out the element you want.

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)
>>> pairs
[("I'm", 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
>>> result = [x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

2.6 Objetos

2.6.1 O que é uma classe?

A class is the particular object type created by executing a class statement. Class objects are used as templates to create instance objects, which embody both the data (attributes) and code (methods) specific to a datatype.

A class can be based on one or more other classes, called its base class(es). It then inherits the attributes and methods of its base classes. This allows an object model to be successively refined by inheritance. You might have a generic `Mailbox` class that provides basic accessor methods for a mailbox, and subclasses such as `MboxMailbox`, `MaildirMailbox`, `OutlookMailbox` that handle various specific mailbox formats.

2.6.2 O que é um método?

A method is a function on some object `x` that you normally call as `x.name(arguments...)`. Methods are defined as functions inside the class definition:

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

2.6.3 O que é o self?

Self is merely a conventional name for the first argument of a method. A method defined as `meth(self, a, b, c)` should be called as `x.meth(a, b, c)` for some instance `x` of the class in which the definition occurs; the called method will think it is called as `meth(x, a, b, c)`.

Veja também *Por que o 'self' deve ser usado explicitamente em definições de método e chamadas?*.

2.6.4 Como eu verifico se um objeto é uma instância de uma dada classe ou de uma subclasse dela?

Use the built-in function `isinstance(obj, cls)`. You can check if an object is an instance of any of a number of classes by providing a tuple instead of a single class, e.g. `isinstance(obj, (class1, class2, ...))`, and can also check whether an object is one of Python's built-in types, e.g. `isinstance(obj, str)` or `isinstance(obj, (int, float, complex))`.

Note that `isinstance()` also checks for virtual inheritance from an *abstract base class*. So, the test will return `True` for a registered class even if hasn't directly or indirectly inherited from it. To test for "true inheritance", scan the *MRO* of the class:

```
from collections.abc import Mapping

class P:
     pass

class C(P):
    pass

Mapping.register(P)
```

```
>>> c = C()
>>> isinstance(c, C)        # direct
True
>>> isinstance(c, P)        # indirect
True
>>> isinstance(c, Mapping)  # virtual
True

# Actual inheritance chain
>>> type(c).__mro__
(<class 'C'>, <class 'P'>, <class 'object'>)

# Test for "true inheritance"
>>> Mapping in type(c).__mro__
False
```

Note that most programs do not use `isinstance()` on user-defined classes very often. If you are developing the classes yourself, a more proper object-oriented style is to define methods on the classes that encapsulate a particular behaviour, instead of checking the object's class and doing a different thing based on what class it is. For example, if you have a function that does something:

```
def search(obj):
    if isinstance(obj, Mailbox):
        ...  # code to search a mailbox
    elif isinstance(obj, Document):
        ...  # code to search a document
    elif ...
```

A better approach is to define a `search()` method on all the classes and just call it:

```
class Mailbox:
    def search(self):
        ...  # code to search a mailbox

class Document:
    def search(self):
        ...  # code to search a document

obj.search()
```

2.6.5 O que é delegation?

Delegation is an object oriented technique (also called a design pattern). Let's say you have an object `x` and want to change the behaviour of just one of its methods. You can create a new class that provides a new implementation of the method you're interested in changing and delegates all other methods to the corresponding method of `x`.

Python programmers can easily implement delegation. For example, the following class implements a class that behaves like a file but converts all written data to uppercase:

```
class UpperOut:

    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)
```

Here the `UpperOut` class redefines the `write()` method to convert the argument string to uppercase before calling the underlying `self._outfile.write()` method. All other methods are delegated to the underlying `self._outfile` object. The delegation is accomplished via the `__getattr__()` method; consult the language reference for more information about controlling attribute access.

Note that for more general cases delegation can get trickier. When attributes must be set as well as retrieved, the class must define a `__setattr__()` method too, and it must do so carefully. The basic implementation of `__setattr__()` is roughly equivalent to the following:

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

Many `__setattr__()` implementations call `object.__setattr__()` to set an attribute on self without causing infinite recursion:

```
class X:
    def __setattr__(self, name, value):
        # Custom logic here...
        object.__setattr__(self, name, value)
```

Alternatively, it is possible to set attributes by inserting entries into `self.__dict__` directly.

2.6.6 Como eu chamo um método definido numa classe base derivada de uma classe que estende ela?

Use a função embutida `super()`:

```
class Derived(Base):
    def meth(self):
        super().meth()  # calls Base.meth
```

In the example, `super()` will automatically determine the instance from which it was called (the `self` value), look up the *method resolution order* (MRO) with `type(self).__mro__`, and return the next in line after `Derived` in the MRO: `Base`.

2.6.7 Como eu posso organizar meu código para facilitar a troca da classe base?

You could assign the base class to an alias and derive from the alias. Then all you have to change is the value assigned to the alias. Incidentally, this trick is also handy if you want to decide dynamically (e.g. depending on availability of resources) which base class to use. Example:

```
class Base:
    ...

BaseAlias = Base

class Derived(BaseAlias):
    ...
```

2.6.8 How do I create static class data and static class methods?

Both static data and static methods (in the sense of C++ or Java) are supported in Python.

For static data, simply define a class attribute. To assign a new value to the attribute, you have to explicitly use the class name in the assignment:

```
class C:
    count = 0   # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count  # or return self.count
```

`c.count` also refers to `C.count` for any `c` such that `isinstance(c, C)` holds, unless overridden by `c` itself or by some class on the base-class search path from `c.__class__` back to `C`.

Caution: within a method of C, an assignment like `self.count = 42` creates a new and unrelated instance named "count" in `self`'s own dict. Rebinding of a class-static data name must always specify the class whether inside a method or not:

```
C.count = 314
```

Métodos estáticos são possíveis:

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
```

However, a far more straightforward way to get the effect of a static method is via a simple module-level function:

```
def getcount():
    return C.count
```

If your code is structured so as to define one class (or tightly related class hierarchy) per module, this supplies the desired encapsulation.

2.6.9 Como eu posso sobrecarregar construtores (ou métodos) em Python?

Essa resposta na verdade se aplica para todos os métodos, mas a pergunta normalmente aparece primeiro no contexto de construtores.

Em C++ escreveríamos

```
class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n"; }
}
```

Em Python você tem que escrever um único construtor que pega todos os casos usando argumentos padrão. Por exemplo:

```
class C:
    def __init__(self, i=None):
        if i is None:
            print("No arguments")
        else:
            print("Argument is", i)
```

Isso não é inteiramente equivalente, mas já está bem próximo.

Você também pode tentar uma lista de argumentos de comprimento variável, por exemplo:

```
def __init__(self, *args):
    ...
```

A mesma abordagem funciona para todas as definições de métodos.

2.6.10 Eu tentei usar __spam e recebi um erro sobre _SomeClassName__spam.

Variable names with double leading underscores are "mangled" to provide a simple but effective way to define class private variables. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with any leading underscores stripped.

The identifier can be used unchanged within the class, but to access it outside the class, the mangled name must be used:

```
class A:
    def __one(self):
        return 1
    def two(self):
        return 2 * self.__one()

class B(A):
    def three(self):
        return 3 * self._A__one()

four = 4 * A()._A__one()
```

In particular, this does not guarantee privacy since an outside user can still deliberately access the private attribute; many Python programmers never bother to use private variable names at all.

Ver também

O especificações de desfiguração de nome privado para detalhes e casos especiais.

2.6.11 Minha classe define __del__, mas o mesmo não é chamado quando eu excluo o objeto.

Há várias razões possíveis para isto.

The `del` statement does not necessarily call `__del__()` – it simply decrements the object's reference count, and if this reaches zero `__del__()` is called.

If your data structures contain circular links (e.g. a tree where each child has a parent reference and each parent has a list of children) the reference counts will never go back to zero. Once in a while Python runs an algorithm to detect such cycles, but the garbage collector might run some time after the last reference to your data structure vanishes, so your `__del__()` method may be called at an inconvenient and random time. This is inconvenient if you're trying to reproduce a problem. Worse, the order in which object's `__del__()` methods are executed is arbitrary. You can run `gc.collect()` to force a collection, but there *are* pathological cases where objects will never be collected.

Despite the cycle collector, it's still a good idea to define an explicit `close()` method on objects to be called whenever you're done with them. The `close()` method can then remove attributes that refer to subobjects. Don't call `__del__()` directly – `__del__()` should call `close()` and `close()` should make sure that it can be called more than once for the same object.

Another way to avoid cyclical references is to use the `weakref` module, which allows you to point to objects without incrementing their reference count. Tree data structures, for instance, should use weak references for their parent and sibling references (if they need them!).

Finally, if your `__del__()` method raises an exception, a warning message is printed to `sys.stderr`.

2.6.12 Como eu consigo pegar uma lista de todas as instâncias de uma dada classe?

Python does not keep track of all instances of a class (or of a built-in type). You can program the class's constructor to keep track of all instances by keeping a list of weak references to each instance.

2.6.13 Por que o resultado de `id()` aparenta não ser único?

The `id()` builtin returns an integer that is guaranteed to be unique during the lifetime of the object. Since in CPython, this is the object's memory address, it happens frequently that after an object is deleted from memory, the next freshly created object is allocated at the same position in memory. This is illustrated by this example:

```
>>> id(1000)
13901272
>>> id(2000)
13901272
```

The two ids belong to different integer objects that are created before, and deleted immediately after execution of the `id()` call. To be sure that objects whose id you want to examine are still alive, create another reference to the object:

```
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
```

2.6.14 Quando eu posso depender dos testes de identidade com o operador *is*?

The `is` operator tests for object identity. The test `a is b` is equivalent to `id(a) == id(b)`.

The most important property of an identity test is that an object is always identical to itself, `a is a` always returns `True`. Identity tests are usually faster than equality tests. And unlike equality tests, identity tests are guaranteed to return a boolean `True` or `False`.

However, identity tests can *only* be substituted for equality tests when object identity is assured. Generally, there are three circumstances where identity is guaranteed:

1) Assignments create new names but do not change object identity. After the assignment `new = old`, it is guaranteed that `new is old`.

2) Putting an object in a container that stores object references does not change object identity. After the list assignment `s[0] = x`, it is guaranteed that `s[0] is x`.

3) If an object is a singleton, it means that only one instance of that object can exist. After the assignments `a = None` and `b = None`, it is guaranteed that `a is b` because `None` is a singleton.

In most other circumstances, identity tests are inadvisable and equality tests are preferred. In particular, identity tests should not be used to check constants such as `int` and `str` which aren't guaranteed to be singletons:

```
>>> a = 1000
>>> b = 500
>>> c = b + 500
>>> a is c
False

>>> a = 'Python'
>>> b = 'Py'
>>> c = b + 'thon'
>>> a is c
False
```

Do mesmo jeito, novas instâncias de contêineres mutáveis nunca são idênticas:

```
>>> a = []
>>> b = []
>>> a is b
False
```

In the standard library code, you will see several common patterns for correctly using identity tests:

1) As recommended by **PEP 8**, an identity test is the preferred way to check for `None`. This reads like plain English in code and avoids confusion with other objects that may have boolean values that evaluate to false.

2) Detecting optional arguments can be tricky when `None` is a valid input value. In those situations, you can create a singleton sentinel object guaranteed to be distinct from other objects. For example, here is how to implement a method that behaves like `dict.pop()`:

```
_sentinel = object()

def pop(self, key, default=_sentinel):
    if key in self:
        value = self[key]
        del self[key]
        return value
    if default is _sentinel:
        raise KeyError(key)
    return default
```

3) Container implementations sometimes need to augment equality tests with identity tests. This prevents the code from being confused by objects such as `float('NaN')` that are not equal to themselves.

For example, here is the implementation of `collections.abc.Sequence.__contains__()`:

```
def __contains__(self, value):
    for v in self:
        if v is value or v == value:
            return True
    return False
```

2.6.15 How can a subclass control what data is stored in an immutable instance?

When subclassing an immutable type, override the `__new__()` method instead of the `__init__()` method. The latter only runs *after* an instance is created, which is too late to alter data in an immutable instance.

All of these immutable classes have a different signature than their parent class:

```
from datetime import date

class FirstOfMonthDate(date):
    "Always choose the first day of the month"
    def __new__(cls, year, month, day):
        return super().__new__(cls, year, month, 1)

class NamedInt(int):
    "Allow text names for some numbers"
    xlat = {'zero': 0, 'one': 1, 'ten': 10}
    def __new__(cls, value):
        value = cls.xlat.get(value, value)
        return super().__new__(cls, value)

class TitleStr(str):
    "Convert str to name suitable for a URL path"
    def __new__(cls, s):
        s = s.lower().replace(' ', '-')
        s = ''.join([c for c in s if c.isalnum() or c == '-'])
        return super().__new__(cls, s)
```

The classes can be used like this:

```
>>> FirstOfMonthDate(2012, 2, 14)
FirstOfMonthDate(2012, 2, 1)
>>> NamedInt('ten')
10
>>> NamedInt(20)
20
>>> TitleStr('Blog: Why Python Rocks')
'blog-why-python-rocks'
```

2.6.16 How do I cache method calls?

The two principal tools for caching methods are `functools.cached_property()` and `functools.lru_cache()`. The former stores results at the instance level and the latter at the class level.

The *cached_property* approach only works with methods that do not take any arguments. It does not create a reference to the instance. The cached method result will be kept only as long as the instance is alive.

The advantage is that when an instance is no longer used, the cached method result will be released right away. The disadvantage is that if instances accumulate, so too will the accumulated method results. They can grow without

bound.

The *lru_cache* approach works with methods that have *hashable* arguments. It creates a reference to the instance unless special efforts are made to pass in weak references.

The advantage of the least recently used algorithm is that the cache is bounded by the specified *maxsize*. The disadvantage is that instances are kept alive until they age out of the cache or until the cache is cleared.

Esse exemplo mostra as várias técnicas:

```
class Weather:
    "Lookup weather information on a government website"

    def __init__(self, station_id):
        self._station_id = station_id
        # The _station_id is private and immutable

    def current_temperature(self):
        "Latest hourly observation"
        # Do not cache this because old results
        # can be out of date.

    @cached_property
    def location(self):
        "Return the longitude/latitude coordinates of the station"
        # Result only depends on the station_id

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='mm'):
        "Rainfall on a given date"
        # Depends on the station_id, date, and units.
```

The above example assumes that the *station_id* never changes. If the relevant instance attributes are mutable, the *cached_property* approach can't be made to work because it cannot detect changes to the attributes.

To make the *lru_cache* approach work when the *station_id* is mutable, the class needs to define the `__eq__()` and `__hash__()` methods so that the cache can detect relevant attribute updates:

```
class Weather:
    "Example with a mutable station identifier"

    def __init__(self, station_id):
        self.station_id = station_id

    def change_station(self, station_id):
        self.station_id = station_id

    def __eq__(self, other):
        return self.station_id == other.station_id

    def __hash__(self):
        return hash(self.station_id)

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='cm'):
        'Rainfall on a given date'
        # Depends on the station_id, date, and units.
```

2.7 Módulos

2.7.1 Como faço para criar um arquivo .pyc?

When a module is imported for the first time (or when the source file has changed since the current compiled file was created) a `.pyc` file containing the compiled code should be created in a `__pycache__` subdirectory of the directory containing the `.py` file. The `.pyc` file will have a filename that starts with the same name as the `.py` file, and ends with `.pyc`, with a middle component that depends on the particular `python` binary that created it. (See **PEP 3147** for details.)

One reason that a `.pyc` file may not be created is a permissions problem with the directory containing the source file, meaning that the `__pycache__` subdirectory cannot be created. This can happen, for example, if you develop as one user but run as another, such as if you are testing with a web server.

Unless the `PYTHONDONTWRITEBYTECODE` environment variable is set, creation of a .pyc file is automatic if you're importing a module and Python has the ability (permissions, free space, etc…) to create a `__pycache__` subdirectory and write the compiled module to that subdirectory.

Running Python on a top level script is not considered an import and no `.pyc` will be created. For example, if you have a top-level module `foo.py` that imports another module `xyz.py`, when you run `foo` (by typing `python foo.py` as a shell command), a `.pyc` will be created for `xyz` because `xyz` is imported, but no `.pyc` file will be created for `foo` since `foo.py` isn't being imported.

If you need to create a `.pyc` file for `foo` – that is, to create a `.pyc` file for a module that is not imported – you can, using the `py_compile` and `compileall` modules.

The `py_compile` module can manually compile any module. One way is to use the `compile()` function in that module interactively:

```
>>> import py_compile
>>> py_compile.compile('foo.py')
```

This will write the `.pyc` to a `__pycache__` subdirectory in the same location as `foo.py` (or you can override that with the optional parameter `cfile`).

You can also automatically compile all files in a directory or directories using the `compileall` module. You can do it from the shell prompt by running `compileall.py` and providing the path of a directory containing Python files to compile:

```
python -m compileall .
```

2.7.2 Como encontro o nome do módulo atual?

A module can find out its own module name by looking at the predefined global variable `__name__`. If this has the value `'__main__'`, the program is running as a script. Many modules that are usually used by importing them also provide a command-line interface or a self-test, and only execute this code after checking `__name__`:

```
def main():
    print('Running test...')
    ...

if __name__ == '__main__':
    main()
```

2.7.3 How can I have modules that mutually import each other?

Suponha que tenhas os seguintes módulos:

`foo.py`:

```
from bar import bar_var
foo_var = 1
```

bar.py:

```
from foo import foo_var
bar_var = 2
```

O problema é que o interpretador vai realizar os seguintes passos:

- main imports `foo`
- Empty globals for `foo` are created
- `foo` is compiled and starts executing
- `foo` imports `bar`
- Empty globals for `bar` are created
- `bar` is compiled and starts executing
- `bar` imports `foo` (which is a no-op since there already is a module named `foo`)
- The import mechanism tries to read `foo_var` from `foo` globals, to set `bar.foo_var = foo.foo_var`

The last step fails, because Python isn't done with interpreting `foo` yet and the global symbol dictionary for `foo` is still empty.

The same thing happens when you use `import foo`, and then try to access `foo.foo_var` in global code.

There are (at least) three possible workarounds for this problem.

Guido van Rossum recommends avoiding all uses of `from <module> import ...`, and placing all code inside functions. Initializations of global variables and class variables should use constants or built-in functions only. This means everything from an imported module is referenced as `<module>.<name>`.

Jim Roskind suggests performing steps in the following order in each module:

- exports (globals, functions, and classes that don't need imported base classes)
- Declaração `import`
- código ativo (incluindo globais que são inicializadas de valores importados)

Van Rossum doesn't like this approach much because the imports appear in a strange place, but it does work.

Matthias Urlichs recommends restructuring your code so that the recursive import is not necessary in the first place.

Essas soluções não são mutualmente exclusivas.

2.7.4 __import__('x.y.z') returns <module 'x'>; how do I get z?

Consider using the convenience function `import_module()` from `importlib` instead:

```
z = importlib.import_module('x.y.z')
```

2.7.5 Quando eu edito um módulo importado e o reimporto, as mudanças não aparecem. Por que isso acontece?

For reasons of efficiency as well as consistency, Python only reads the module file on the first time a module is imported. If it didn't, in a program consisting of many modules where each one imports the same basic module, the basic module would be parsed and re-parsed many times. To force re-reading of a changed module, do this:

```
import importlib
import modname
importlib.reload(modname)
```

Aviso: essa técnica não é 100% a prova de falhas. Em particular, módulos contendo instruções como

```
from modname import some_objects
```

will continue to work with the old version of the imported objects. If the module contains class definitions, existing class instances will *not* be updated to use the new class definition. This can result in the following paradoxical behaviour:

```
>>> import importlib
>>> import cls
>>> c = cls.C()                # Create an instance of C
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)       # isinstance is false?!?
False
```

A natureza do problema fica clara se você exibir a "identidade" dos objetos da classe:

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```

CAPÍTULO 3

Design e Histórico FAQ

3.1 Por que o Python usa indentação para agrupamento de instruções?

Guido van Rossum acredita que usar indentação para agrupamento é extremamente elegante e contribui muito para a clareza de programa Python mediano. Muitas pessoas aprendem a amar esta ferramenta depois de um tempo.

Uma vez que não há colchetes de início / fim, não pode haver um desacordo entre o agrupamento percebido pelo analisador sintático e pelo leitor humano. Ocasionalmente, programadores C irão encontrar um fragmento de código como este:

```
if (x <= y)
        x++;
        y--;
z++;
```

Somente a instrução `x++` é executada se a condição for verdadeira, mas a indentação leva muitos a acreditarem no contrário. Com frequência, até programadores C experientes a observam fixamente por um longo tempo, perguntando-se por que `y` está sendo decrementada até mesmo para `x > y`.

Como não há chaves de início / fim, o Python é muito menos propenso a conflitos no estilo de codificação. Em C, existem muitas maneiras diferentes de colocar as chaves. Depois de se tornar habitual a leitura e escrita de código usando um estilo específico, é normal sentir-se um pouco receoso ao ler (ou precisar escrever) em um estilo diferente.

Muitos estilos de codificação colocam chaves de início / fim em uma linha sozinhos. Isto torna os programas consideravelmente mais longos e desperdiça espaço valioso na tela, dificultando a obtenção de uma boa visão geral de um programa. Idealmente, uma função deve caber em uma tela (digamos, 20 a 30 linhas). 20 linhas de Python podem fazer muito mais trabalho do que 20 linhas de C. Isso não se deve apenas à falta de colchetes de início/fim – a falta de declarações e os tipos de dados de alto nível também são responsáveis – mas a sintaxe baseada em indentação certamente ajuda.

3.2 Por que eu estou recebendo resultados estranhos com simples operações aritméticas?

Veja a próxima questão.

3.3 Por que o cálculo de pontos flutuantes são tão imprecisos?

Usuários são frequentemente surpreendidos por resultados como este:

```
>>> 1.2 - 1.0
0.19999999999999996
```

e pensam que isto é um bug do Python. Não é não. Isto tem pouco a ver com o Python, e muito mais a ver com como a estrutura da plataforma lida com números em ponto flutuante.

O tipo `float` no CPython usa um `double` do C para armazenamento. O valor de um objeto `float` é armazenado em ponto flutuante binário com uma precisão fixa (normalmente 53 bits) e Python usa operações C, que por sua vez dependem da implementação de hardware no processador, para realizar operações de ponto flutuante. Isso significa que, no que diz respeito às operações de ponto flutuante, Python se comporta como muitas linguagens populares, incluindo C e Java.

Muitos números podem ser escritos facilmente em notação decimal, mas não podem ser expressados exatamente em ponto flutuante binário. Por exemplo, após:

```
>>> x = 1.2
```

o valor armazenado para `x` é uma (ótima) aproximação para o valor decimal `1.2`, mas não é exatamente igual. Em uma máquina típica, o valor real armazenado é:

```
1.0011001100110011001100110011001100110011001100110011 (binary)
```

que é exatamente:

```
1.1999999999999999555910790149937383830547332763671875 (decimal)
```

A precisão típica de 53 bits fornece floats do Python com precisão de 15 a 16 dígitos decimais.

Para uma explicação mais completa, consulte o capítulo de aritmética de ponto flutuante no tutorial Python.

3.4 Por que strings do Python são imutáveis?

Existem várias vantagens.

Uma delas é o desempenho: saber que uma string é imutável significa que podemos alocar espaço para ela no momento da criação, e os requisitos de armazenamento são fixos e imutáveis. Esta é também uma das razões para a distinção entre tuplas e listas.

Outra vantagem é que strings em Python são consideradas tão "elementares" quanto números. Nenhuma atividade alterará o valor 8 para qualquer outra coisa e, em Python, nenhuma atividade alterará a string "oito" para qualquer outra coisa.

3.5 Por que o 'self' deve ser usado explicitamente em definições de método e chamadas?

A ideia foi emprestada do Modula-3. Acontece dela ser muito útil, por vários motivos.

Primeiro, é mais óbvio que você está usando um método ou atributo de instância em vez de uma variável local. Ler `self.x` ou `self.meth()` deixa absolutamente claro que uma variável de instância ou método é usado mesmo se você não souber a definição da classe de cor. Em C++, você pode perceber pela falta de uma declaração de variável local (presumindo que globais são raras ou facilmente reconhecíveis) – mas no Python não há declarações de variáveis locais, então você teria que procurar a definição de classe para tenha certeza. Alguns padrões de codificação C++ e Java exigem que atributos de instância tenham um prefixo `m_`, portanto, essa explicitação ainda é útil nessas linguagens também.

Segundo, significa que nenhuma sintaxe especial é necessária se você quiser referenciar ou chamar explicitamente o método de uma classe específica. Em C++, se você quiser usar um método de uma classe base que é substituído em uma classe derivada, você deve usar o operador `::` – no Python, você pode escrever `baseclass.methodname(self, <lista de argumentos>)`. Isto é particularmente útil para métodos `__init__()` e, em geral, em casos onde um método de classe derivada deseja estender o método da classe base de mesmo nome e, portanto, precisa chamar o método da classe base de alguma forma.

Finalmente, por exemplo, variáveis, ele resolve um problema sintático com atribuição: uma vez que variáveis locais em Python são (por definição!) aquelas variáveis às quais um valor é atribuído em um corpo de função (e que não são explicitamente declaradas globais), é necessário deve haver alguma forma de dizer ao interpretador que uma atribuição deveria ser atribuída a uma variável de instância em vez de a uma variável local, e deve ser preferencialmente sintática (por razões de eficiência). C++ faz isso através de declarações, mas Python não possui declarações e seria uma pena ter que introduzi-las apenas para esse fim. Usar o `self.var` explícito resolve isso muito bem. Da mesma forma, para usar variáveis de instância, ter que escrever `self.var` significa que referências a nomes não qualificados dentro de um método não precisam pesquisar nos diretórios da instância. Em outras palavras, variáveis locais e variáveis de instância residem em dois espaço de nomes diferentes, e você precisa informar ao Python qual espaço de nomes usar.

3.6 Por que não posso usar uma atribuição em uma expressão?

A partir do Python 3.8, você pode!

Expressões de atribuição usando o operador morsa `:=` atribuem uma variável em uma expressão:

```
while chunk := fp.read(200):
   print(chunk)
```

Veja **PEP 572** para mais informações.

3.7 Por que o Python usa métodos para algumas funcionalidades (ex: lista.index()) mas funções para outras (ex: len(lista))?

Como Guido disse:

> (a) Para algumas operações, a notação de prefixo é melhor lida do que as operações de prefixo (e infixo!) têm uma longa tradição em matemática que gosta de notações onde os recursos visuais ajudam o matemático a pensar sobre um problema. Compare a facilidade com que reescrevemos uma fórmula como x*(a+b) em x*a + x*b com a falta de jeito de fazer a mesma coisa usando uma notação OO bruta.
>
> (b) Quando leio o código que diz len(x) eu *sei* que ele está perguntando o comprimento de alguma coisa. Isso me diz duas coisas: o resultado é um número inteiro e o argumento é algum tipo de contêiner. Pelo contrário, quando leio x.len(), já devo saber que x é algum tipo de contêiner implementando uma interface ou herdando de uma classe que possui um len() padrão. Veja a confusão que ocasionalmente temos quando uma classe que não está implementando um mapeamento tem um método get() ou keys(), ou algo que não é um arquivo tem um método write().
>
> —https://mail.python.org/pipermail/python-3000/2006-November/004643.html

3.8 Por que o join() é um método de string em vez de ser um método de lista ou tupla?

Strings se tornaram muito parecidas com outros tipos padrão a partir do Python 1.6, quando métodos que dão a mesma funcionalidade que sempre esteve disponível utilizando as funções do módulo de string foram adicionados. A maior parte desses novos métodos foram amplamente aceitos, mas o que parece deixar alguns programadores desconfortáveis é:

```
", ".join(['1', '2', '4', '8', '16'])
```

que dá o resultado:

```
"1, 2, 4, 8, 16"
```

Existem dois argumentos comuns contra esse uso.

O primeiro segue as linhas de: "Parece muito feio usar um método de uma string literal (constante de string)", para o qual a resposta é que pode, mas uma string literal é apenas um valor fixo. Se os métodos devem ser permitidos em nomes vinculados a strings, não há razão lógica para torná-los indisponíveis em literais.

A segunda objeção é normalmente formulada como: "Na verdade, estou dizendo a uma sequência para unir seus membros com uma constante de string". Infelizmente, você não está. Por alguma razão parece haver muito menos dificuldade em ter `split()` como um método string, já que nesse caso é fácil ver que

```
"1, 2, 4, 8, 16".split(", ")
```

é uma instrução para uma string literal para retornar as substrings delimitadas pelo separador fornecido (ou, por padrão, execuções arbitrárias de espaço em branco).

`join()` é um método de string porque ao usá-lo você está dizendo à string separadora para iterar sobre uma sequência de strings e se inserir entre elementos adjacentes. Este método pode ser usado com qualquer argumento que obedeça às regras para objetos sequência, incluindo quaisquer novas classes que você mesmo possa definir. Existem métodos semelhantes para bytes e objetos bytearray.

3.9 Quão rápidas são as exceções?

Um bloco `try/except` é extremamente eficiente se nenhuma exceção for levantada. Na verdade, capturar uma exceção é caro. Nas versões do Python anteriores à 2.0 era comum usar esta expressão:

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

Isso somente fazia sentido quando você esperava que o dicionário tivesse uma chave quase que toda vez. Se esse não fosse o caso, você escrevia desta maneira:

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

Para este caso específico, você também pode usar `value = dict.setdefault(key, getvalue(key))`, mas apenas se a chamada `getvalue()` tiver pouco custosa o suficiente porque é avaliada em todos os casos.

3.10 Por que não existe uma instrução de switch ou case no Python?

Em geral, as instruções switch estruturadas executam um bloco de código quando uma expressão possui um valor ou conjunto de valores específico. Desde o Python 3.10 é possível combinar facilmente valores literais, ou constantes dentro de um espaço de nomes, com uma instrução `match ... case`. Uma alternativa mais antiga é uma sequência de `if... elif... elif... else`.

Para casos em que você precisa escolher entre um grande número de possibilidades, você pode criar um dicionário mapeando valores de caso para funções a serem chamadas. Por exemplo:

```
functions = {'a': function_1,
             'b': function_2,
             'c': self.method_1}

func = functions[value]
func()
```

Para chamar métodos em objetos, você pode simplificar ainda mais usando olá função embutida `getattr()` para recuperar métodos com um nome específico:

```
class MyVisitor:
    def visit_a(self):
        ...

    def dispatch(self, value):
        method_name = 'visit_' + str(value)
        method = getattr(self, method_name)
        method()
```

É sugerido que você use um prefixo para os nomes dos métodos, como `visit_` neste exemplo. Sem esse prefixo, se os valores vierem de uma fonte não confiável, um invasor poderá chamar qualquer método no seu objeto.

Imitar o comportamento do switch no C, em que a execução atravessa as instruções de um case para outro caso não seja interrompida, é possível, mais difícil, e menos necessário.

3.11 Você não pode emular threads no interpretador em vez de confiar em uma implementação de thread específica do sistema operacional?

Resposta 1: Infelizmente, o interpretador envia pelo menos um quadro de pilha C para cada quadro de pilha Python. Além disso, as extensões podem retornar ao Python em momentos quase aleatórios. Portanto, uma implementação completa de threads requer suporte de thread para C.

Resposta 2: Felizmente, existe o Stackless Python, que tem um laço de interpretador completamente redesenhado que evita a pilha C.

3.12 Por que expressões lambda não podem conter instruções?

Expressões lambda no Python não podem conter instruções porque o framework sintático do Python não consegue manipular instruções aninhadas dentro de expressões. No entanto, no Python, isso não é um problema sério. Diferentemente das formas de lambda em outras linguagens, onde elas adicionam funcionalidade, lambdas de Python são apenas notações simplificadas se você tiver muita preguiça de definir uma função.

Funções já são objetos de primeira classe em Python, e podem ser declaradas em um escopo local. Portanto a única vantagem de usar um lambda em vez de uma função definida localmente é que você não precisa inventar um nome para a função – mas esta só é uma variável local para a qual o objeto da função (que é exatamente do mesmo tipo de um objeto que uma expressão lambda carrega) é atribuído!

3.13 O Python pode ser compilado para linguagem de máquina, C ou alguma outra linguagem?

Cython compiles a modified version of Python with optional annotations into C extensions. Nuitka is an up-and--coming compiler of Python into C++ code, aiming to support the full Python language.

3.14 Como o Python gerencia memória?

Os detalhes do gerenciamento de memória Python dependem da implementação. A implementação padrão do Python, *CPython*, usa contagem de referências para detectar objetos inacessíveis, e outro mecanismo para coletar ciclos de referência, executando periodicamente um algoritmo de detecção de ciclo que procura por ciclos inacessíveis e exclui os objetos envolvidos. O módulo `gc` fornece funções para realizar uma coleta de lixo, obter estatísticas de depuração e ajustar os parâmetros do coletor.

Other implementations (such as Jython or PyPy), however, can rely on a different mechanism such as a full-blown garbage collector. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

Em algumas implementações do Python, o código a seguir (que funciona bem no CPython) provavelmente ficará sem descritores de arquivo:

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

Na verdade, usando o esquema de contagem de referências e destrutor de referências do CPython, cada nova atribuição a `f` fecha o arquivo anterior. Com um GC tradicional, entretanto, esses objetos de arquivo só serão coletados (e fechados) em intervalos variados e possivelmente longos.

Se você quiser escrever um código que funcione com qualquer implementação Python, você deve fechar explicitamente o arquivo ou usar a instrução `with`; isso funcionará independentemente do esquema de gerenciamento de memória:

```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

3.15 Por que o CPython não usa uma forma mais tradicional de esquema de coleta de lixo?

Por um lado, este não é um recurso padrão C e, portanto, não é portátil. (Sim, sabemos sobre a biblioteca Boehm GC. Ela possui pedaços de código assembler para a *maioria* das plataformas comuns, não para todas elas, e embora seja em sua maioria transparente, não é completamente transparente; são necessários patches para obter Python para trabalhar com isso.)

O GC tradicional também se torna um problema quando o Python é incorporado em outros aplicativos. Embora em um Python independente seja bom substituir o padrão `malloc()` e `free()` por versões fornecidas pela biblioteca GC, uma aplicação que incorpora Python pode querer ter seu *próprio* substituto para `malloc()` e `free()`, e podem não querer Python. No momento, CPython funciona com qualquer coisa que implemente `malloc()` e `free()` corretamente.

3.16 Por que toda memória não é liberada quando o CPython fecha?

Os objetos referenciados nos espaço de nomes globais dos módulos Python nem sempre são desalocados quando o Python é encerrado. Isso pode acontecer se houver referências circulares. Existem também certos bits de memória alocados pela biblioteca C que são impossíveis de liberar (por exemplo, uma ferramenta como o Purify reclamará disso). Python é, no entanto, agressivo quanto à limpeza de memória na saída e tenta destruir todos os objetos.

Se você quiser forçar o Python a excluir certas coisas na desalocação, use o módulo `atexit` para executar uma função que forçará essas exclusões.

3.17 Por que existem tipos de dados separados para tuplas e listas?

Listas e tuplas, embora semelhantes em muitos aspectos, geralmente são usadas de maneiras fundamentalmente diferentes. Tuplas podem ser consideradas semelhantes a `records` de Pascal ou `structs` de C; são pequenas coleções de dados relacionados que podem ser de diferentes tipos e operados como um grupo. Por exemplo, uma coordenada cartesiana é representada apropriadamente como uma tupla de dois ou três números.

As listas, por outro lado, são mais parecidas com arrays em outras linguagens. Eles tendem a conter um número variável de objetos, todos do mesmo tipo e operados um por um. Por exemplo, `os.listdir('.')` retorna uma lista de strings representando os arquivos no diretório atual. Funções que operam nesta saída geralmente não seriam interrompidas se você adicionasse outro arquivo ou dois ao diretório.

As tuplas são imutáveis, o que significa que, uma vez criada uma tupla, você não pode substituir nenhum de seus elementos por um novo valor. As listas são mutáveis, o que significa que você sempre pode alterar os elementos de uma lista. Somente elementos imutáveis podem ser usados como chaves de dicionário e, portanto, apenas tuplas e não listas podem ser usadas como chaves.

3.18 Como as listas são implementadas no CPython?

As listas do CPython são, na verdade, vetores de comprimento variável, listas vinculadas não no estilo Lisp. A implementação usa um vetor contíguo de referências a outros objetos e mantém um ponteiro para esse vetor e o comprimento de vetor em uma estrutura de cabeçalho de lista.

Isso torna a indexação de uma lista `a[i]` uma operação cujo custo é independente do tamanho da lista ou do valor do índice.

Quando itens são anexados ou inseridos, o vetor de referências é redimensionado. Alguma inteligência é aplicada para melhorar o desempenho de anexar itens repetidamente; quando o vetor precisa ser aumentado, algum espaço extra é alocado para que as próximas vezes não exijam um redimensionamento real.

3.19 Como são os dicionários implementados no CPython?

Os dicionários do CPython são implementados como tabelas hash redimensionáveis. Em comparação com árvores B, isso oferece melhor desempenho para pesquisa (de longe a operação mais comum) na maioria das circunstâncias, e a implementação é mais simples.

Os dicionários funcionam calculando um código hash para cada chave armazenada no dicionário usando a função embutida `hash()`. O código hash varia amplamente dependendo da chave e da semente por processo; por exemplo, `'Python'` poderia fazer hash para `-539294296` enquanto `'python'`, uma string que difere por um único bit, poderia fazer hash para `1142331976`. O código hash é então usado para calcular um local em um vetor interno onde o valor será armazenado. Supondo que você esteja armazenando chaves com valores de hash diferentes, isso significa que os dicionários levam um tempo constante – $O(1)$, na notação Big-O – para recuperar uma chave.

3.20 Por que chaves de dicionário devem ser imutáveis?

A implementação da tabela hash de dicionários usa um valor hash calculado a partir do valor da chave para encontrar a chave. Se a chave fosse um objeto mutável, seu valor poderia mudar e, portanto, seu hash também poderia mudar. Mas como quem altera o objeto-chave não pode saber que ele estava sendo usado como chave de dicionário, ele não pode mover a entrada no dicionário. Então, quando você tentar procurar o mesmo objeto no dicionário, ele não será encontrado porque seu valor de hash é diferente. Se você tentasse procurar o valor antigo, ele também não seria encontrado, porque o valor do objeto encontrado naquele hash seria diferente.

Se você deseja que um dicionário seja indexado com uma lista, simplesmente converta primeiro a lista em uma tupla; a função `tuple(L)` cria uma tupla com as mesmas entradas da lista `L`. As tuplas são imutáveis e, portanto, podem ser usadas como chaves de dicionário.

Algumas soluções inaceitáveis que foram propostas:

- Listas de hash por endereço (ID do objeto). Isto não funciona porque se você construir uma nova lista com o mesmo valor ela não será encontrada; por exemplo.:

```
mydict = {[1, 2]: '12'}
print(mydict[[1, 2]])
```

  levantaria uma exceção `KeyError` porque o id do `[1, 2]` usado na segunda linha difere daquele da primeira linha. Em outras palavras, as chaves de dicionário devem ser comparadas usando `==`, não usando `is`.

- Fazer uma cópia ao usar uma lista como chave. Isso não funciona porque a lista, sendo um objeto mutável, poderia conter uma referência a si mesma e então o código copiado entraria em um laço infinito.

- Permitir listas como chaves, mas dizer ao usuário para não modificá-las. Isso permitiria uma classe de bugs difíceis de rastrear em programas quando você esquecesse ou modificasse uma lista por acidente. Também invalida uma importante invariante dos dicionários: todo valor em `d.keys()` pode ser usado como chave do dicionário.

- Marcar listas como somente leitura quando forem usadas como chave de dicionário. O problema é que não é apenas o objeto de nível superior que pode alterar seu valor; você poderia usar uma tupla contendo uma lista como chave. Inserir qualquer coisa como chave em um dicionário exigiria marcar todos os objetos acessíveis a partir daí como somente leitura – e, novamente, objetos autorreferenciais poderiam causar um laço infinito.

Existe um truque para contornar isso se você precisar, mas use-o por sua própria conta e risco: você pode agrupar uma estrutura mutável dentro de uma instância de classe que tenha um método `__eq__()` e um método `__hash__()`. Você deve então certificar-se de que o valor de hash para todos os objetos invólucros que residem em um dicionário (ou outra estrutura baseada em hash) permaneça fixo enquanto o objeto estiver no dicionário (ou outra estrutura).

```
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
            try:
                result = result + (hash(el) % 9999999) * 1001 + i
            except Exception:
                result = (result % 7777777) + i * 333
        return result
```

Observe que o cálculo do hash é complicado pela possibilidade de que alguns membros da lista possam ser não não-hasheável e também pela possibilidade de estouro aritmético.

Além disso, deve ser sempre o caso que se `o1 == o2` (ou seja, `o1.__eq__(o2) is True`) então `hash(o1) == hash(o2)` (ou seja, `o1.__hash__() == o2.__hash__()`), independentemente de o objeto estar em um dicionário ou não. Se você não cumprir essas restrições, os dicionários e outras estruturas baseadas em hash se comportarão mal.

No caso de `ListWrapper`, sempre que o objeto invólucro estiver em um dicionário, a lista agrupada não deve ser alterada para evitar anomalias. Não faça isso a menos que esteja preparado para pensar muito sobre os requisitos e as consequências de não atendê-los corretamente. Considere-se avisado.

3.21 Por que lista.sort() não retorna a lista ordenada?

Em situações nas quais desempenho importa, fazer uma cópia da lista só para ordenar seria desperdício. Portanto, `lista.sort()` ordena a lista. De forma a lembrá-lo desse fato, isso não retorna a lista ordenada. Desta forma, você

não vai ser confundido a acidentalmente sobrescrever uma lista quando você precisar de uma cópia ordenada mas também precisar manter a versão não ordenada.

Se você quiser retornar uma nova lista, use a função embutida `sorted()` ao invés. Essa função cria uma nova lista a partir de um iterável provido, o ordena e retorna. Por exemplo, aqui é como se itera em cima das chaves de um dicionário de maneira ordenada:

```
for key in sorted(mydict):
    ...  # do whatever with mydict[key]...
```

3.22 Como você especifica e aplica uma especificação de interface no Python?

Uma especificação de interface para um módulo fornecida por linguagens como C++ e Java descreve os protótipos para os métodos e funções do módulo. Muitos acham que a aplicação de especificações de interface em tempo de compilação ajuda na construção de programas grandes.

Python 2.6 adiciona um módulo `abc` que permite definir Classes Base Abstratas (ABCs). Você pode então usar `isinstance()` e `issubclass()` para verificar se uma instância ou classe implementa um ABC específico. O módulo `collections.abc` define um conjunto de ABCs úteis como `Iterable`, `Container` e `MutableMapping`

Para Python, muitas das vantagens das especificações de interface podem ser obtidas por uma disciplina de teste apropriada para componentes.

Um bom conjunto de testes para um módulo pode fornecer um teste de regressão e servir como uma especificação de interface do módulo e um conjunto de exemplos. Muitos módulos Python podem ser executados como um script para fornecer um simples "autoteste". Mesmo módulos que usam interfaces externas complexas muitas vezes podem ser testados isoladamente usando emulações triviais da interface externa. Os módulos `doctest` e `unittest` ou estruturas de teste de terceiros podem ser usados para construir conjuntos de testes exaustivos que exercitam cada linha de código em um módulo.

Uma disciplina de teste apropriada pode ajudar a construir aplicações grandes e complexas no Python, assim como ter especificações de interface. Na verdade, pode ser melhor porque uma especificação de interface não pode testar certas propriedades de um programa. Por exemplo, espera-se que o método `list.append()` adicione novos elementos ao final de alguma lista interna; uma especificação de interface não pode testar se sua implementação `list.append()` realmente fará isso corretamente, mas é trivial verificar esta propriedade em um conjunto de testes.

Escrever conjuntos de testes é muito útil e você pode querer projetar seu código para torná-lo facilmente testável. Uma técnica cada vez mais popular, o desenvolvimento orientado a testes, exige a escrita de partes do conjunto de testes primeiro, antes de escrever qualquer parte do código real. É claro que o Python permite que você seja desleixado e nem escreva casos de teste.

3.23 Por que não há goto?

Na década de 1970, as pessoas perceberam que o goto irrestrito poderia levar a um código "espaguete" confuso, difícil de entender e revisar. Em uma linguagem de alto nível, também é desnecessário, desde que existam maneiras de ramificar (em Python, com instruções `if` e `or`, `and` e expressões `if/else`) e iterar (com instruções `while` e `for`, possivelmente contendo `continue` e `break`).

Também é possível usar exceções para fornecer um "goto estruturado" que funcione mesmo em chamadas de função. Muitos acham que as exceções podem convenientemente emular todos os usos razoáveis das construções `go` ou `goto` de C, Fortran e outras linguagens. Por exemplo:

```
class label(Exception): pass  # declare a label

try:
    ...
    if condition: raise label()  # goto label
    ...
```

(continua na próxima página)

(continuação da página anterior)

```
except label:  # where to goto
    pass
...
```

Isso não permite que você pule para o meio de um laço, mas isso geralmente é considerado um abuso de `goto` de qualquer maneira. Use com moderação.

3.24 Por que strings brutas (r-strings) não podem terminar com uma contrabarra?

Mais precisamente, eles não podem terminar com um número ímpar de contrabarras: a contrabarra não pareada no final escapa do caractere de aspa de fechamento, deixando uma string não terminada.

Strings brutas foram projetadas para facilitar a criação de entrada para processadores (principalmente mecanismos de expressão regular) que desejam fazer seu próprio processamento de escape de contrabarra. De qualquer forma, esses processadores consideram uma contrabarra incomparável como um erro, portanto, as strings brutas não permitem isso. Em troca, eles permitem que você transmita o caractere de aspas da string escapando dele com uma contrabarra. Essas regras funcionam bem quando r-strings são usadas para a finalidade pretendida.

Se você estiver tentando criar nomes de caminho do Windows, observe que todas as chamadas do sistema do Windows também aceitam barras:

```
f = open("/mydir/file.txt")  # works fine!
```

Se você estiver tentando criar um nome de caminho para um comando DOS, tente, por exemplo. um de

```
dir = r"\this\is\my\dos\dir" "\\"
dir = r"\this\is\my\dos\dir\ "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```

3.25 Por que o Python não tem uma instrução "with" para atribuição de atributos?

Python tem uma instrução `with` que envolve a execução de um bloco, chamando o código na entrada e na saída do bloco. Algumas linguagens têm uma construção desta forma:

```
with obj:
    a = 1                # equivalent to obj.a = 1
    total = total + 1    # obj.total = obj.total + 1
```

Em Python, tal construção seria ambígua.

Outras linguagens, como Object Pascal, Delphi, e C++, usam tipos estáticos, então é possível saber, de maneira não ambígua, que membro está sendo atribuído. Esse é o principal ponto da tipagem estática – o compilador *sempre* sabe o escopo de toda variável em tempo de compilação.

O Python usa tipos dinâmicos. É impossível saber com antecedência que atributo vai ser referenciado em tempo de execução. Atributos membro podem ser adicionados ou removidos de objetos dinamicamente. Isso torna impossível saber, de uma leitura simples, que atributo está sendo referenciado: um atributo local, um atributo global ou um atributo membro?

Por exemplo, pegue o seguinte trecho incompleto:

```
def foo(a):
    with a:
        print(x)
```

O trecho presume que `a` deve ter um atributo de membro chamado `x`. No entanto, não há nada em Python que diga isso ao interpretador. O que deveria acontecer se `a` for, digamos, um número inteiro? Se houver uma variável global chamada `x`, ela será usada dentro do bloco `with`? Como você pode ver, a natureza dinâmica do Python torna essas escolhas muito mais difíceis.

O principal benefício de `with` e recursos de linguagem semelhantes (redução do volume de código) pode, no entanto, ser facilmente alcançado em Python por atribuição. Em vez de:

```
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

escreva isso:

```
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

Isso também tem o efeito colateral de aumentar a velocidade de execução porque as ligações de nome são resolvidas a tempo de execução em Python, e a segunda versão só precisa performar a resolução uma vez.

Propostas semelhantes que introduziriam sintaxe para reduzir ainda mais o volume do código, como o uso de um 'ponto inicial', foram rejeitadas em favor da explicitação (consulte https://mail.python.org/pipermail/python-ideas/2016-May/040070.html).

3.26 Por que os geradores não suportam a instrução with?

Por razões técnicas, um gerador usado diretamente como gerenciador de contexto não funcionaria corretamente. Quando, como é mais comum, um gerador é usado como um iterador executado até a conclusão, nenhum fechamento é necessário. Quando estiver, envolva-o como `contextlib.closing(generator)` na instrução `with`.

3.27 Por que dois pontos são necessários para as instruções de if/while/def/class?

Os dois pontos são obrigatórios primeiramente para melhorar a leitura (um dos resultados da linguagem experimental ABC). Considere isso:

```
if a == b
    print(a)
```

versus

```
if a == b:
    print(a)
```

Note como a segunda é ligeiramente mais fácil de ler. Note com mais atenção como os dois pontos iniciam o exemplo nessa resposta de perguntas frequentes; é um uso padrão em Português.

Outro motivo menor é que os dois pontos deixam mais fácil para os editores com realce de sintaxe; eles podem procurar por dois pontos para decidir quando a recuo precisa ser aumentada em vez de precisarem fazer uma análise mais elaborada do texto do programa.

3.28 Por que o Python permite vírgulas ao final de listas e tuplas?

O Python deixa você adicionar uma vírgula ao final de listas, tuplas e dicionários:

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7],  # last trailing comma is optional but good style
}
```

Existem várias razões para permitir isso.

Quando você possui um valor literal para uma lista, tupla, ou dicionário disposta através de múltiplas linhas, é mais fácil adicionar mais elementos porque você não precisa lembrar de adicionar uma vírgula na linha anterior. As linhas também podem ser reordenadas sem criar um erro de sintaxe.

Acidentalmente omitir a vírgula pode levar a erros que são difíceis de diagnosticar. Por exemplo:

```
x = [
  "fee",
  "fie"
  "foo",
  "fum"
]
```

Essa lista parece ter quatro elementos, mas na verdade contém três: “fee”, “fiefoo” e “fum”. Sempre adicionar a vírgula evita essa fonte de erro.

Permitir a vírgula no final também pode deixar a geração de código programático mais fácil.

CAPÍTULO 4

FAQ de Bibliotecas e Extensões

4.1 Questões gerais sobre bibliotecas

4.1.1 Como encontrar um módulo ou aplicação para realizar uma tarefa X?

Verifique a Referência de Bibliotecas para ver se há um módulo relevante da biblioteca padrão. (Eventualmente, você aprenderá o que está na biblioteca padrão e poderá pular esta etapa.)

Para pacotes de terceiros, pesquise no Python Package Index ou tente no Google ou outro buscador na web. Pesquisando por "Python" mais uma ou dois argumentos nomeados do seu tópico de interesse geralmente encontrará algo útil.

4.1.2 Onde está o código-fonte do math.py (socket.py, regex.py, etc.)?

Se você não conseguir encontrar um arquivo de origem para um módulo, ele pode ser um módulo embutido ou carregado dinamicamente, implementado em C, C++ ou outra linguagem compilada. Nesse caso, você pode não ter o arquivo de origem ou pode ser algo como `mathmodule.c`, em algum lugar do diretório de origem C (não no caminho do Python).

Existem (pelo menos) três tipos de módulos no Python:

1) módulos escritos em Python (.py)

2) módulos escritos em C e carregados dinamicamente (.dll, .pyd, .so, .sl, etc.);

3) módulos escritos em C e vinculados ao interpretador; para obter uma dessas listas, digite:

```
import sys
print(sys.builtin_module_names)
```

4.1.3 Como tornar um script Python executável no Unix?

Você precisa fazer duas coisas: o arquivo do script deve ser executável e a primeira linha deve começar com `#!` seguido do caminho do interpretador Python.

Inicialmente, execute o `chmod +x scriptfile` ou, talvez, o `chmod 755 scriptfile`.

A segunda coisa pode ser feita de várias maneiras. A maneira mais direta é escrever

```
#!/usr/local/bin/python
```

como a primeira linha do seu arquivo, usando o endereço do caminho onde o interpretador Python está instalado.

Se você deseja que o script seja independente de onde o interpretador Python mora, você pode usar o programa **env**. Quase todas as variantes do Unix suportam o seguinte, presumindo que o interpretador Python esteja em um diretório no `PATH` do usuário:

```
#!/usr/bin/env python
```

Não faça isso para CGI scripts. A variável `PATH` para CGI scripts é normalmente muito pequena, portanto, você precisa usar o caminho completo do interpretador.

Ocasionalmente, o ambiente de um usuário está tão cheio que o programa **/usr/bin/env** falha; ou não há nenhum programa env. Nesse caso, você pode tentar o seguinte hack (graças a Alex Rezinsky):

```
#! /bin/sh
""":"
exec python $0 ${1+"$@"}
"""
```

Uma pequena desvantagem é que isso define o script's __doc__ string. Entretanto, você pode corrigir isso adicionando

```
__doc__ = """...Whatever..."""
```

4.1.4 Existe um pacote de curses/termcap para Python?

Para variantes Unix: A distribuição fonte padrão do Python vem com um módulo do curses no subdiretório Modules, embora não seja compilado por padrão. (Observe que isso não está disponível na distribuição do Windows – não há módulo curses para o Windows.)

O módulo `curses` provê recursos básicos de curses, bem como muitas funções adicionais de ncurses e curses SYSV, como cor, suporte a conjuntos de caracteres alternativos, pads e suporte a mouse. Isso significa que o módulo não é compatível com sistemas operacionais que possuem apenas maldições BSD, mas não parece haver nenhum sistema operacional mantido atualmente que se enquadre nesta categoria.

4.1.5 Existe a função onexit() equivalente ao C no Python?

O módulo `atexit` fornece uma função de registro similar ao `onexit()` do C.

4.1.6 Por que o meu manipulador de sinal não funciona?

O maior problema é que o manipulador de sinal é declarado com uma lista de argumentos incorretos. Isso é chamado como

```
handler(signum, frame)
```

portanto, isso deve ser declarado com dois parâmetros:

```
def handler(signum, frame):
    ...
```

4.2 Tarefas comuns

4.2.1 Como testar um programa ou componente Python?

A Python vem com dois frameworks de teste. O módulo `doctest` busca por exemplos nas docstrings de um módulo e os executa, comparando o resultado com a saída esperada informada na docstring.

O módulo `unittest` é uma estrutura de teste mais sofisticada, modelada nas estruturas de teste do Java e do Smalltalk.

Para facilitar os testes, você deve usar um bom design modular em seu programa. Seu programa deve ter quase todas as funcionalidades encapsuladas em funções ou métodos de classe – e isso às vezes tem o efeito surpreendente e agradável de fazer o programa executar mais rápido (porque os acessos às variáveis locais são mais rápidos que os acessos globais). Além disso, o programa deve evitar depender de variáveis globais mutantes, pois isso torna os testes muito mais difíceis de serem realizados.

A lógica principal do seu programa pode tão simples quanto

```
if __name__ == "__main__":
    main_logic()
```

no botão do módulo principal do seus programa.

Depois que seu programa estiver organizado como uma coleção tratável de comportamentos de funções e classes, você deverá escrever funções de teste que exercitem os comportamentos. Um conjunto de testes que automatiza uma sequência de testes pode ser associado a cada módulo. Parece muito trabalhoso, mas como o Python é tão conciso e flexível, é surpreendentemente fácil. Você pode tornar a codificação muito mais agradável e divertida escrevendo suas funções de teste em paralelo com o "código de produção", pois isso torna mais fácil encontrar bugs e até mesmo falhas de design mais cedo.

Os "módulos de suporte" que não se destinam a ser o módulo principal de um programa podem incluir um autoteste do módulo.

```
if __name__ == "__main__":
    self_test()
```

Mesmo quando as interfaces externas não estiverem disponíveis, os programas que interagem com interfaces externas complexas podem ser testados usando as interfaces "falsas" implementadas no Python.

4.2.2 Como faço para criar uma documentação de doc strings?

O módulo `pydoc` pode criar HTML a partir das strings de documentos em seu código-fonte Python. Uma alternativa para criar documentação de API puramente a partir de docstrings é epydoc. Sphinx também pode incluir conteúdo docstring.

4.2.3 Como faço para pressionar uma tecla de cada vez?

Para variantes do Unix existem várias soluções. Apesar de ser um módulo grande para aprender, é simples fazer isso usando o módulo curses.

4.3 Threads

4.3.1 Como faço para programar usando threads?

Certifique-se de usar o módulo `threading` e não o módulo `_thread`. O módulo `threading` constrói abstrações convenientes sobre as primitivas de baixo nível fornecidas pelo módulo `_thread`.

4.3.2 Nenhuma de minhas threads parece funcionar, por que?

Assim que a thread principal acaba, todas as threads são eliminadas. Sua thread principal está sendo executada tão rápida que não está dando tempo para realizar qualquer trabalho.

Uma solução simples é adicionar um tempo de espera no final do programa até que todos os threads sejam concluídos:

```
import threading, time

def thread_task(name, n):
    for i in range(n):
        print(name, i)
```

(continua na próxima página)

(continuação da página anterior)

```
for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)  # <----------------------------!
```

Mas agora (em muitas plataformas) as threads não funcionam em paralelo, mas parecem funcionar sequencialmente, um de cada vez! O motivo é que o agendador de threads do sistema operacional não inicia uma nova thread até que a thread anterior seja bloqueada.

Uma solução simples é adicionar um pequeno tempo de espera no início da função:

```
def thread_task(name, n):
    time.sleep(0.001)  # <--------------------!
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)
```

Em vez de tentar adivinhar um bom valor de atraso para `time.sleep()`, é melhor usar algum tipo de mecanismo de semáforo. Uma ideia é usar o módulo `queue` para criar um objeto fila, deixar cada thread anexar um token à fila quando terminar e deixar a thread principal ler tantos tokens da fila quantos threads houver.

4.3.3 Como distribuo o trabalho entre várias threads de trabalho?

A maneira mais fácil é usar o módulo `concurrent.futures`, especialmente a classe `ThreadPoolExecutor`.

Ou, se quiser um controle preciso sobre o algoritmo de despacho, você pode escrever sua própria lógica manualmente. Use o módulo `queue` para criar uma fila contendo uma lista de tarefas. A classe `Queue` mantém uma lista de objetos e possui um método `.put(obj)` que adiciona itens à fila e um método `.get()` para retorná-los. A classe cuidará da trava necessário para garantir que cada trabalho seja entregue exatamente uma vez.

Aqui está um exemplo simples:

```
import threading, queue, time

# The worker thread gets jobs off the queue.  When the queue is empty, it
# assumes there will be no more work and exits.
# (Realistically workers will run until terminated.)
def worker():
    print('Running worker')
    time.sleep(0.1)
    while True:
        try:
            arg = q.get(block=False)
        except queue.Empty:
            print('Worker', threading.current_thread(), end=' ')
            print('queue empty')
            break
        else:
            print('Worker', threading.current_thread(), end=' ')
            print('running with argument', arg)
            time.sleep(0.5)
```

(continua na próxima página)

(continuação da página anterior)

```
# Create queue
q = queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' % (i+1))
    t.start()

# Begin adding work to the queue
for i in range(50):
    q.put(i)

# Give threads time to run
print('Main thread sleeping')
time.sleep(5)
```

Quando executado, isso produzirá a seguinte saída:

```
Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started 130283832797456)> running with argument 0
Worker <Thread(worker 2, started 130283824404752)> running with argument 1
Worker <Thread(worker 3, started 130283816012048)> running with argument 2
Worker <Thread(worker 4, started 130283807619344)> running with argument 3
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

Consulte a documentação do módulo para mais detalhes; a classe `Queue` fornece uma interface com recursos.

4.3.4 Que tipos de variáveis globais mutáveis são seguras para thread?

Uma *trava global do interpretador* (GIL) é usada internamente para garantir que apenas um thread seja executado na VM Python por vez. Em geral, Python oferece alternar entre threads apenas entre instruções de bytecode; a frequência com que ele muda pode ser definida via `sys.setswitchinterval()`. Cada instrução de bytecode e, portanto, todo o código de implementação C alcançado por cada instrução é, portanto, atômico do ponto de vista de um programa Python.

Em teoria, isso significa que uma contabilidade exata requer um entendimento exato da implementação do bytecode PVM. Na prática, isso significa que as operações em variáveis compartilhadas de tipos de dados integrados (inteiros, listas, dicionarios, etc.) que "parecem atômicas" realmente são.

Por exemplo, as seguintes operações são todas atômicas (L, L1, L2 são listas, D, D1, D2 são dicionários, x, y são objetos, i, j são inteiros):

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
```

(continua na próxima página)

(continuação da página anterior)

```
D1.update(D2)
D.keys()
```

Esses não são:

```
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

Operações que substituem outros objetos podem invocar o método `__del__()` desses outros objetos quando sua contagem de referências chega a zero, e isso pode afetar as coisas. Isto é especialmente verdadeiro para as atualizações em massa de dicionários e listas. Em caso de dúvida, use um mutex!

4.3.5 Não podemos remover a Trava Global do interpretador?

A *trava global do interpretador* (GIL) é frequentemente vista como um obstáculo para a implantação do Python em máquinas servidoras multiprocessadas de ponta, porque um programa Python multi-threaded efetivamente usa apenas uma CPU, devido à insistência de que (quase) todo código Python só pode ser executado enquanto a GIL é mantida.

Na época do Python 1.5, Greg Stein implementou um conjunto abrangente de patches (os patches de "threading livre") que removeu o GIL e o substituiu por bloqueio refinado. Adam Olsen recentemente fez um experimento semelhante em seu projeto python-safethread. Infelizmente, ambos os experimentos exibiram uma queda acentuada no desempenho de thread único (pelo menos 30% mais lento), devido à quantidade de bloqueio de granulação fina necessária para compensar a remoção do GIL.

Isso não significa que você não possa fazer bom uso do Python em máquinas com várias CPUs! Você só precisa ser criativo ao dividir o trabalho entre vários *processos* em vez de vários *threads*. A classe `ProcessPoolExecutor` no novo módulo `concurrent.futures` fornece uma maneira fácil de fazer isso; o módulo `multiprocessing` fornece uma API de nível inferior caso você queira mais controle sobre o envio de tarefas.

O uso criterioso de extensões C também ajudará; se você usar uma extensão C para executar uma tarefa demorada, a extensão poderá liberar a GIL enquanto o thread de execução estiver no código C e permitir que outros threads realizem algum trabalho. Alguns módulos de biblioteca padrão como `zlib` e `hashlib` já fazem isso.

Foi sugerido que o GIL deveria ser um bloqueio por estado por interpretador, em vez de verdadeiramente global; os interpretadores não seriam capazes de compartilhar objetos. Infelizmente, isso também não é provável que aconteça. Seria uma quantidade enorme de trabalho, porque muitas implementações de objetos atualmente possuem estado global. Por exemplo, inteiros pequenos e strings curtas são armazenados em cache; esses caches teriam que ser movidos para o estado de interpretador. Outros tipos de objetos possuem sua própria lista livre; essas listas livres teriam que ser movidas para o estado de interpretador. E assim por diante.

E duvido que isso possa ser feito em tempo finito, porque o mesmo problema existe para extensões de terceiros. É provável que extensões de terceiros estejam sendo gravadas em uma taxa mais rápida do que você pode convertê-las para armazenar todo o seu estado global no estado do interpretador.

E finalmente, uma vez que você tem vários interpretadores que não compartilham seu estado, o que você ganhou ao executar processos separados em cada interpretador?

4.4 Entrada e Saída

4.4.1 Como faço para excluir um arquivo? (E outras perguntas sobre arquivos)

Use `os.remove(arquivo)` ou `os.unlink(arquivo)`; para documentação, veja o módulo `os`. As duas funções são idênticas; `unlink()` é simplesmente o nome da chamada do sistema para esta função no Unix.

Para remover um diretório, use `os.rmdir()`; use `os.mkdir()` para criar um. `os.makedirs(caminho)` criará quaisquer diretórios intermediários em `path` que não existam. `os.removedirs(caminho)` removerá diretórios

intermediários, desde que estejam vazios; se quiser excluir toda uma árvore de diretórios e seu conteúdo, use `shutil.rmtree()`.

Para renomear um arquivos, use `os.rename(caminho_antigo, caminho_novo)`.

Para truncar um arquivo, abra-o usando `f = open(arquivo, "rb+")`, e use `f.truncate(posição)`; a posição tem como padrão a posição atual de busca. Há também `os.ftruncate(fd, posição)` para arquivos abertos com `os.open()`, onde *fd* é o descritor de arquivo (um pequeno inteiro).

O módulo `shutil` também contém uma série de funções para trabalhar em arquivos, incluindo `copyfile()`, `copytree()` e `rmtree()`.

4.4.2 Como eu copio um arquivo?

O módulo `shutil` contém uma função `copyfile()`. Observe que em volumes NTFS do Windows, ele não copia fluxos de dados alternativos nem forks de recursos em volumes HFS+ do macOS, embora ambos sejam raramente usados agora. Ele também não copia permissões de arquivo e metadados, embora usar `shutil.copy2()` em vez disso preserve a maioria (embora não todos) deles.

4.4.3 Como leio (ou escrevo) dados binários?

Para ler ou escrever formatos de dados binários complexos, é melhor usar o módulo `struct`. Ele permite que você pegue uma string contendo dados binários (geralmente números) e a converta em objetos Python; e vice-versa.

Por exemplo, o código a seguir lê dois inteiros de 2 bytes e um inteiro de 4 bytes no formato big-endian de um arquivo:

```
import struct

with open(filename, "rb") as f:
    s = f.read(8)
    x, y, z = struct.unpack(">hhl", s)
```

O '>' na string de formato força dados big-endian; a letra 'h' lê um "inteiro curto" (2 bytes) e 'l' lê um "inteiro longo" (4 bytes) da string.

Para dados mais regulares (por exemplo, uma lista homogênea de ints ou floats), você também pode usar o módulo `array`.

> **Nota**
>
> Para ler e escrever dados binários, é obrigatório abrir o arquivo no modo binário (aqui, passando `"rb"` para `open()`). Se você usar `"r"` em vez disso (o padrão), o arquivo será aberto no modo texto e `f.read()` retornará objetos `str` em vez de objetos `bytes`.

4.4.4 Por que não consigo usar os.read() em um encadeamento com os.popen()?

`os.read()` é uma função de baixo nível que pega um descritor de arquivo, um pequeno inteiro representando o arquivo aberto. `os.popen()` cria um objeto arquivo de alto nível, o mesmo tipo retornado pela função embutida `open()`. Assim, para ler *n* bytes de um pipe *p* criado com `os.popen()`, você precisa usar `p.read(n)`.

4.4.5 Como acesso a porta serial (RS232)?

Para Win32, OSX, Linux, BSD, Jython, IronPython:

pyserial

Para Unix, veja uma postagem da Usenet de Mitch Chapman:

https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com

4.4.6 Por que o sys.stdout (stdin, stderr) não fecha?

Os *objetos arquivos* do Python são uma camada de alto nível de abstração em descritores de arquivo C de baixo nível.

Para a maioria dos objetos arquivo que você cria em Python por meio da função embutida `open()`, `f.close()` marca o objeto arquivo Python como fechado do ponto de vista do Python e também organiza o fechamento do descritor de arquivo C subjacente. Isso também acontece automaticamente no destrutor de `f`, quando `f` se torna lixo.

Mas stdin, stdout e stderr são tratados de forma especial pelo Python, devido ao status especial que também lhes é dado pelo C. Executar `sys.stdout.close()` marca o objeto arquivo de nível Python como fechado, mas *não* fecha o descritor de arquivo C associado.

Para fechar o descritor de arquivo C subjacente para um desses três, você deve primeiro ter certeza de que é isso que você realmente quer fazer (por exemplo, você pode confundir módulos de extensão tentando fazer E/S). Se for, use `os.close()`:

```
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

Ou você pode usar as constantes numéricas 0, 1 e 2, respectivamente.

4.5 Programação Rede / Internet

4.5.1 Quais ferramentas para WWW existem no Python?

Veja os capítulos intitulados internet e netdata no Manual de Referência da Biblioteca. Python tem muitos módulos que ajudarão você a construir sistemas web do lado do servidor e do lado do cliente.

Um resumo dos frameworks disponíveis é disponibilizado por Paul Boddie em https://wiki.python.org/moin/WebProgramming.

Cameron Laird maintains a useful set of pages about Python web technologies at https://web.archive.org/web/20210224183619/http://phaseit.net/claird/comp.lang.python/web_python.

4.5.2 Como faço para imitar a submissão de formulário CGI (METHOD=POST)?

Gostaria de recuperar páginas da WEB resultantes de um formulário POST. Existe algum código que consigo fazer isso facilmente?

Yes. Here's a simple example that uses `urllib.request`:

```
#!/usr/local/bin/python

import urllib.request

# build the query string
qs = "First=Josephine&MI=Q&Last=Public"

# connect and send the server a path
req = urllib.request.urlopen('http://www.some-server.out-there'
                             '/cgi-bin/some-cgi-script', data=qs)
with req:
    msg, hdrs = req.read(), req.info()
```

Note that in general for percent-encoded POST operations, query strings must be quoted using `urllib.parse.urlencode()`. For example, to send `name=Guy Steele, Jr.`:

```
>>> import urllib.parse
>>> urllib.parse.urlencode({'name': 'Guy Steele, Jr.'})
'name=Guy+Steele%2C+Jr.'
```

Ver também

urllib-howto para mais exemplos.

4.5.3 Qual módulo devo usar para ajudar na geração do HTML?

Você pode encontrar uma coleção de links úteis na página wiki WebProgramming.

4.5.4 Como envio um e-mail de um script Python?

Use a biblioteca padrão do módulo `smtplib`.

Aqui está um remetente de e-mail interativo muito simples. Este método funcionará em qualquer host que suporte o protocolo SMTP.

```
import sys, smtplib

fromaddr = input("From: ")
toaddrs  = input("To: ").split(',')
print("Enter message, end with ^D:")
msg = ''
while True:
    line = sys.stdin.readline()
    if not line:
        break
    msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Uma alternativa somente para Unix usa o sendmail. A localização do programa sendmail varia entre os sistemas; às vezes é `/usr/lib/sendmail`, às vezes `/usr/sbin/sendmail`. A página de manual do sendmail vai ajudar você. Aqui está um código de exemplo:

```
import os

SENDMAIL = "/usr/sbin/sendmail"  # sendmail location
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n")  # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print("Sendmail exit status", sts)
```

4.5.5 Como evito um bloqueio no método connect() de um soquete?

O módulo `select` é normalmente usado para ajudar com E/S assíncrona nos soquetes.

Para evitar que a conexão TCP bloqueie, você pode definir o soquete para o modo sem bloqueio. Então, quando você fizer o `connect()`, você se conectará imediatamente (improvável) ou obterá uma exceção que contém o número de erro como `.errno`. `errno.EINPROGRESS` indica que a conexão está em andamento, mas ainda não terminou. Diferentes sistemas operacionais retornarão valores diferentes, então você terá que verificar o que é retornado no seu sistema.

Você pode usar o método `connect_ex()` para evitar criar uma exceção. Ele retornará apenas o valor de errno. Para pesquisar, você pode chamar `connect_ex()` novamente mais tarde – `0` ou `errno.EISCONN` indicam que você está conectado – ou você pode passar este soquete para `select.select()` para verificar se ele é gravável.

> **Nota**
>
> O módulo `asyncio` fornece uma biblioteca assíncrona de thread única e concorrente de propósito geral, que pode ser usada para escrever código de rede não bloqueante. A biblioteca de terceiros Twisted é uma alternativa popular e rica em recursos.

4.6 Base de Dados

4.6.1 Existem interfaces para banco de dados em Python?

Sim.

Interfaces para hashes baseados em disco, como `DBM` e `GDBM` também estão incluídas no Python padrão. Há também o módulo `sqlite3`, que fornece um banco de dados relacional baseado em disco leve.

Suporte para a maioria dos bancos de dados relacionais está disponível. Para mais detalhes, veja a página wiki DatabaseProgramming para detalhes.

4.6.2 Como você implementa objetos persistentes no Python?

O módulo de biblioteca `pickle` resolve isso de uma maneira muito geral (embora você ainda não possa armazenar coisas como arquivos abertos, soquetes ou janelas), e o módulo de biblioteca `shelve` usa pickle e (g)dbm para criar mapeamentos persistentes contendo objetos Python arbitrários.

4.7 Matemáticos e Numéricos

4.7.1 Como gero número aleatórios no Python?

O módulo padrão `random` implementa um gerador de números aleatórios. O uso é simples:

```
import random
random.random()
```

Isso retorna um número flutuante aleatório no intervalo [0, 1).

Existem também muitos outros geradores aleatórios neste módulo, como:

- `randrange(a, b)` escolhe um número inteiro no intervalo entre [a, b).
- `uniform(a, b)` escolhe um número de ponto flutuante no intervalo [a, b).
- `normalvariate(mean, sdev)` gera números pseudoaleatórios que seguem uma distribuição normal (Gaussiana).

Algumas funções de nível elevado operam diretamente em sequencia, como:

- `choice(S)` escolhe um elemento aleatório de uma determinada sequência.
- `shuffle(L)` embaralha uma lista internamente, ou seja permuta seus elementos aleatoriamente.

Existe também uma classe `Random` que você pode instanciar para criar vários geradores de números aleatórios independentes.

CAPÍTULO 5

FAQ sobre Extensão/Incorporação

5.1 Posso criar minhas próprias funções em C?

Sim, você pode construir módulos embutidos contendo funções, variáveis, exceções e até mesmo novos tipos em C. Isso é explicado no documento extending-index.

A maioria dos livros intermediários ou avançados em Python também abordará esse tópico.

5.2 Posso criar minhas próprias funções em C++?

Sim, usando recursos de compatibilidade encontrados em C++. Coloque `extern "C" { ... }` em torno dos arquivos de inclusão do Python e coloque `extern "C"` antes de cada função que será chamada pelo interpretador do Python. Objetos globais ou estáticos em C++ com construtores provavelmente não são uma boa ideia.

5.3 A escrita em C é difícil, Há algumas alternativas?

Há um número de alternativas para escrever suas próprias extensões em C, dependendo daquilo que você está tentando fazer.

Cython and its relative Pyrex are compilers that accept a slightly modified form of Python and generate the corresponding C code. Cython and Pyrex make it possible to write an extension without having to learn Python's C API.

If you need to interface to some C or C++ library for which no Python extension currently exists, you can try wrapping the library's data types and functions with a tool such as SWIG. SIP, CXX Boost, or Weave are also alternatives for wrapping C++ libraries.

5.4 Como posso executar instruções arbitrárias de Python a partir de C?

The highest-level function to do this is `PyRun_SimpleString()` which takes a single string argument to be executed in the context of the module `__main__` and returns `0` for success and `-1` when an exception occurred (including `SyntaxError`). If you want more control, use `PyRun_String()`; see the source for `PyRun_SimpleString()` in `Python/pythonrun.c`.

5.5 How can I evaluate an arbitrary Python expression from C?

Call the function `PyRun_String()` from the previous question with the start symbol `Py_eval_input`; it parses an expression, evaluates it and returns its value.

5.6 Como extraio valores em C a partir de um objeto Python?

That depends on the object's type. If it's a tuple, `PyTuple_Size()` returns its length and `PyTuple_GetItem()` returns the item at a specified index. Lists have similar functions, `PyList_Size()` and `PyList_GetItem()`.

For bytes, `PyBytes_Size()` returns its length and `PyBytes_AsStringAndSize()` provides a pointer to its value and its length. Note that Python bytes objects may contain null bytes so C's `strlen()` should not be used.

To test the type of an object, first make sure it isn't `NULL`, and then use `PyBytes_Check()`, `PyTuple_Check()`, `PyList_Check()`, etc.

There is also a high-level API to Python objects which is provided by the so-called 'abstract' interface – read `Include/abstract.h` for further details. It allows interfacing with any kind of Python sequence using calls like `PySequence_Length()`, `PySequence_GetItem()`, etc. as well as many other useful protocols such as numbers (`PyNumber_Index()` et al.) and mappings in the PyMapping APIs.

5.7 Como posso utilizar Py_BuildValue() para criar uma tupla de comprimento arbitrário?

You can't. Use `PyTuple_Pack()` instead.

5.8 How do I call an object's method from C?

The `PyObject_CallMethod()` function can be used to call an arbitrary method of an object. The parameters are the object, the name of the method to call, a format string like that used with `Py_BuildValue()`, and the argument values:

```
PyObject *
PyObject_CallMethod(PyObject *object, const char *method_name,
                    const char *arg_format, ...);
```

This works for any object that has methods – whether built-in or user-defined. You are responsible for eventually `Py_DECREF()`'ing the return value.

To call, e.g., a file object's "seek" method with arguments 10, 0 (assuming the file object pointer is "f"):

```
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
        ... an exception occurred ...
}
else {
        Py_DECREF(res);
}
```

Note that since `PyObject_CallObject()` *always* wants a tuple for the argument list, to call a function without arguments, pass "()" for the format, and to call a function with one argument, surround the argument in parentheses, e.g. "(i)".

5.9 How do I catch the output from PyErr_Print() (or anything that prints to stdout/stderr)?

In Python code, define an object that supports the `write()` method. Assign this object to `sys.stdout` and `sys.stderr`. Call print_error, or just allow the standard traceback mechanism to work. Then, the output will go wherever your `write()` method sends it.

The easiest way to do this is to use the `io.StringIO` class:

```
>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!
```

Um objeto personalizado para fazer a mesma coisa seria esse:

```
>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(''.join(sys.stdout.data))
foo
hello world!
```

5.10 How do I access a module written in Python from C?

You can get a pointer to the module object as follows:

```
module = PyImport_ImportModule("<modulename>");
```

If the module hasn't been imported yet (i.e. it is not yet present in `sys.modules`), this initializes the module; otherwise it simply returns the value of `sys.modules["<modulename>"]`. Note that it doesn't enter the module into any namespace – it only ensures it has been initialized and is stored in `sys.modules`.

You can then access the module's attributes (i.e. any name defined in the module) as follows:

```
attr = PyObject_GetAttrString(module, "<attrname>");
```

Calling `PyObject_SetAttrString()` to assign to variables in the module also works.

5.11 How do I interface to C++ objects from Python?

Depending on your requirements, there are many approaches. To do this manually, begin by reading the "Extending and Embedding" document. Realize that for the Python run-time system, there isn't a whole lot of difference between C and C++ – so the strategy of building a new Python type around a C structure (pointer) type will also work for C++ objects.

For C++ libraries, see *A escrita em C é difícil, Há algumas alternativas?*.

5.12 I added a module using the Setup file and the make fails; why?

Setup must end in a newline, if there is no newline there, the build process fails. (Fixing this requires some ugly shell script hackery, and this bug is so minor that it doesn't seem worth the effort.)

5.13 How do I debug an extension?

When using GDB with dynamically loaded extensions, you can't set a breakpoint in your extension until your extension is loaded.

In your `.gdbinit` file (or interactively), add the command:

```
br _PyImport_LoadDynamicModule
```

Then, when you run GDB:

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish   # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```

5.14 I want to compile a Python module on my Linux system, but some files are missing. Why?

Most packaged versions of Python omit some files required for compiling Python extensions.

For Red Hat, install the python3-devel RPM to get the necessary files.

For Debian, run `apt-get install python3-dev`.

5.15 How do I tell "incomplete input" from "invalid input"?

Sometimes you want to emulate the Python interactive interpreter's behavior, where it gives you a continuation prompt when the input is incomplete (e.g. you typed the start of an "if" statement or you didn't close your parentheses or triple string quotes), but it gives you a syntax error message immediately when the input is invalid.

In Python you can use the `codeop` module, which approximates the parser's behavior sufficiently. IDLE uses this, for example.

The easiest way to do it in C is to call `PyRun_InteractiveLoop()` (perhaps in a separate thread) and let the Python interpreter handle the input for you. You can also set the `PyOS_ReadlineFunctionPointer()` to point at your custom input function. See `Modules/readline.c` and `Parser/myreadline.c` for more hints.

5.16 How do I find undefined g++ symbols __builtin_new or __pure_virtual?

To dynamically load g++ extension modules, you must recompile Python, relink it using g++ (change LINKCC in the Python Modules Makefile), and link your extension module using g++ (e.g., `g++ -shared -o mymodule.so mymodule.o`).

5.17 Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?

Yes, you can inherit from built-in classes such as `int`, `list`, `dict`, etc.

The Boost Python Library (BPL, https://www.boost.org/libs/python/doc/index.html) provides a way of doing this from C++ (i.e. you can inherit from an extension class written in C++ using the BPL).

CAPÍTULO 6

Python no Windows

6.1 Como faço para executar um programa Python no Windows?

Esta não é necessariamente uma questão direta. Se você já está familiarizado com a execução de programas através das linha de comando do Windows, então tudo parecerá óbvio; caso contrário, poderá precisar de um pouco mais de orientação.

A menos que você use algum tipo de ambiente de desenvolvimento integrado, você vai acabar digitando os comandos do Windows no que é chamado "janela do DOS" ou "janela do prompt de comando". Geralmente você pode abrir essas janelas procurando na barra de pesquisa por `cmd`. Você deverá reconhecer quando iniciar porque você verá um "Prompt de Comando do Windows", que geralmente tem esta forma:

```
C:\>
```

A letra pode ser diferente, e pode haver outras coisas depois, então você facilmente pode ver algo como:

```
D:\YourName\Projects\Python>
```

dependendo de como seu computador foi configurado e o que mais você tem feito com ele recentemente. Uma vez que você tenha iniciado a janela, você estará no caminho para executar os seus programas Python.

Você deve observar que seu código Python deve ser processado por outro programa chamado interpretador. O interpretador lê o seu código, compila em bytecodes, e depois executa os bytecodes para rodar o seu programa. Então, como você pode organizar o interpretador para lidar com seu Python?

Primeiro, você precisa ter certeza de que sua janela de comando reconhece a palavra "py" como uma instrução para iniciar o interpretador. Se você abriu a janela de comando, você deve tentar digitar o comando "py" e o observar o retorno:

```
C:\Users\YourName> py
```

Você deve então ver algo como:

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]␣
↪on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Você iniciou o interpretador no "modo interativo". Que significa que você pode inserir instruções ou expressões Python interativamente e executá-las ou calculá-las enquanto espera. Esta é uma das características mais fortes do Python. Verifique isso digitando algumas instruções de sua escolha e veja os resultados:

```
>>> print("Hello")
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

Muitas pessoas usam o modo interativo como uma calculadora conveniente, mas altamente programável. Quando quiser encerrar sua sessão interativa do Python, chame a função `exit()` ou mantenha pressionada a tecla `Ctrl` enquanto você digita a `Z` e pressione a tecla "`Enter`" para voltar ao prompt de comando do Windows.

Você também pode descobrir que você tem um item no Menu Iniciar como *Iniciar ▸ Programas▸ Python 3.x ▸ Python (linha de comando)* que resultará em você vendo o prompt `>>>` em uma nova janela. Se acontecer isso, a janela desaparecerá depois que você chamar a função `exit()` ou inserir o caractere `Ctrl-Z`; o Windows está executando um único comando "python" na janela, e fecha quando você termina o interpretador.

Agora que sabemos que o comando `py` é reconhecido, você pode dar seu script Python para ele. Você terá que dar um caminho absoluto ou relativo para o script Python. Vamos dizer que seu script Python está localizado na sua área de trabalho e se chama `hello.py`, e seu prompt de comando está aberto no seu diretório raiz de forma que você está vendo algo similar a:

```
C:\Users\YourName>
```

Então, agora você solicitará o comando `py` para fornecer seu script para Python, digitando `py` seguido pelo seu caminho de script:

```
C:\Users\YourName> py Desktop\hello.py
hello
```

6.2 Como eu faço para criar programas Python executáveis?

No Windows, o instalador padrão do Python já associa a extensão .py com o tipo de arquivo (Python.File) e dá àquele tipo de arquivo um comando aberto que executa o interpretador (`D:\Program Files\Python\python.exe "%1" %*`). Isso é o bastante para fazer scripts executáveis pelo prompt de comando como 'foo.py'. Se você preferir executar o script simplesmente digitando 'foo' sem extensão você precisa adicionar .py à variável de ambiente PATHEXT.

6.3 Por que Python às vezes demora tanto para iniciar?

Geralmente, Python inicia muito rapidamente no Windows, mas ocasionalmente há relatos de erros que, de repente, o Python começa a demorar muito tempo para iniciar. Isso é ainda mais intrigante, porque Python funciona bem em outros sistemas Windows que parecem estar configurados de forma idêntica.

O problema pode ser causado por uma desconfiguração de software antivírus na máquina problemática. Alguns antivírus são conhecidos por introduzir sobrecarga de duas ordens de magnitude no início quando estão configurados para monitorar todas as leituras do sistema de arquivos. Tente verificar a configuração do antivírus nos seus sistemas para assegurar que eles estão de fato configurados identicamente. O McAfee, quando configurado para escanear todo a atividade do sistema de arquivos, é um ofensor conhecido.

6.4 Como eu faço para criar um executável a partir de um código Python?

Consulte *Como posso criar um binário independente a partir de um script Python?* para uma lista de ferramentas que podem ser usada para criar executáveis.

6.5 Um arquivo `*.pyd` é o mesmo que um DLL?

Sim, os arquivos .pyd são dll, mas existem algumas diferenças. Se você possui uma DLL chamada `foo.pyd`, ela deve ter a função `PyInit_foo()`. Você pode escrever "import foo" do Python, e o Python procurará por foo.pyd (assim como foo.py, foo.pyc) e, se o encontrar, tentará chamar `PyInit_foo()` para inicializá-lo. Você não vincula seu arquivo .exe ao arquivo foo.lib, pois isso faria com que o Windows exigisse a presença da DLL.

Observe que o caminho de pesquisa para foo.pyd é PYTHONPATH, não o mesmo que o Windows usa para procurar por foo.dll. Além disso, foo.pyd não precisa estar presente para executar seu programa, enquanto que se você vinculou seu programa a uma dll, a dll será necessária. Obviamente, o foo.pyd é necessário se você quiser dizer `import foo`. Em uma DLL, o vínculo é declarado no código-fonte com `__declspec(dllexport)`. Em um .pyd, o vínculo é definido em uma lista de funções disponíveis.

6.6 Como eu posso embutir Python dentro de uma aplicação do Windows?

A incorporação do interpretador Python em um aplicativo do Windows pode ser resumida da seguinte forma:

1. **Não** compile o Python diretamente em seu arquivo .exe. No Windows, o Python deve ser uma DLL para manipular os módulos de importação que são eles próprios. (Este é o primeiro fato chave não documentado.) Em vez disso, vincule a `python*NN*.dll`; normalmente é instalado em `C:\Windows\System`. *NN* é a versão do Python, um número como "33" para o Python 3.3.

   Você pode vincular ao Python de duas maneiras diferentes. A vinculação em tempo de carregamento significa vincular contra `python*NN*.lib`, enquanto a vinculação em tempo de execução significa vincular a `python*NN*.dll`. (Nota geral: `python*NN*.lib` é a chamada "import lib" correspondente a `python*NN*.dll`. Apenas define símbolos para o vinculador.)

   A vinculação em tempo de execução simplifica bastante as opções de vinculação; tudo acontece em tempo de execução. Seu código deve carregar `python*NN*.dll` usando a rotina `LoadLibraryEx()` do Windows. O código também deve usar rotinas de acesso e dados em `python*NN*.dll` (ou seja, as APIs C do Python) usando ponteiros obtidos pela rotina `GetProcAddress()` do Windows. As macros podem tornar o uso desses ponteiros transparente para qualquer código C que chama rotinas na API C do Python.

2. Se você usa SWIG, é fácil criar um "módulo de extensão" do Python que disponibilizará os dados e os métodos da aplicação para o Python. O SWIG cuidará de todos os detalhes obscuros para você. O resultado é o código C que você vincula *ao* arquivo.exe (!) Você **não** precisa criar um arquivo DLL, o que também simplifica a vinculação.

3. O SWIG criará uma função init (uma função C) cujo nome depende do nome do módulo de extensão. Por exemplo, se o nome do módulo for leo, a função init será chamada initleo(). Se você usa classes de sombra SWIG, como deveria, a função init será chamada initleoc(). Isso inicializa uma classe auxiliar principalmente oculta usada pela classe shadow.

   O motivo pelo qual você pode vincular o código C na etapa 2 ao seu arquivo .exe é que chamar a função de inicialização equivale a importar o módulo para o Python! (Este é o segundo fato chave não documentado.)

4. Em suma, você pode utilizar o código a seguir para inicializar o interpretador Python com seu módulo de extensão.

```
#include <Python.h>
...
Py_Initialize();  // Initialize Python.
initmyAppc();  // Initialize (import) the helper class.
PyRun_SimpleString("import myApp");  // Import the shadow class.
```

5. Existem dois problemas com a API C do Python que se tornarão aparentes se você utiliza um compilador que não seja o MSVC, o compilador utilizado no pythonNN.dll.

   Problema 1: As chamadas funções de "Nível Muito Alto" que recebem argumentos `FILE *` não funcionarão em um ambiente com vários compiladores porque a noção de cada `struct FILE` de um compilador será

diferente. Do ponto de vista da implementação, essas são funções de nível muito baixo.

Problema 2: SWIG gera o seguinte código ao gerar invólucros para funções sem retorno:

```
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

Infelizmente, Py_None é uma macro que se expande para uma referência a uma estrutura de dados complexa chamada _Py_NoneStruct dentro de pythonNN.dll. Novamente, esse código falhará em um ambiente com vários compiladores. Substitua esse código por:

```
return Py_BuildValue("");
```

Pode ser possível usar o comando `%typemap` do SWIG para fazer a alteração automaticamente, embora eu não tenha conseguido fazer isso funcionar (eu sou um completo novato em SWIG).

6. Usar um script de shell do Python para criar uma janela do interpretador Python de dentro da aplicação do Windows não é uma boa ideia; a janela resultante será independente do sistema de janelas da sua aplicação. Em vez disso, você (ou a classe wxPythonWindow) deve criar uma janela "nativa" do interpretador. É fácil conectar essa janela ao interpretador Python. Você pode redirecionar a E/S do Python para qualquer objeto que suporte leitura e gravação; portanto, tudo que você precisa é de um objeto Python (definido no seu módulo de extensão) que contenha métodos read() e write().

6.7 Como eu impeço editores de adicionarem tabulações na minha source do Python?

As perguntas frequentes não recomendam a utilização de tabulações, e o guia de estilo Python, :pep:*8*, recomenda 4 espaços para código de Python distribuído; esse também é o padrão do python-mode do Emacs.

Sob qualquer editor, misturar tabulações e espaços é uma má ideia. O MSVC não é diferente nesse aspecto e é facilmente configurado para usar espaços: Selecione *Tools* ‣ *Options* ‣ *Tabs* e, para o tipo de arquivo "Default", defina "Tab size" e "Indent size" para 4 e selecione o botão de opção "Insert spaces".

O Python levanta `IndentationError` ou `TabError` se tabulações e espaços misturados estiverem causando problemas no espaço em branco à esquerda. Você também pode executar o módulo `tabnanny` para verificar uma árvore de diretórios no modo em lote.

6.8 Como faço para verificar uma tecla pressionada sem bloquear?

Use o módulo `msvcrt`. Este é um módulo de extensão padrão específico do Windows. Ele define uma função `kbhit()` que verifica se um toque no teclado está presente, e `getch()` que recebe um caractere sem ecoá-lo.

6.9 Como resolvo o erro da api-ms-win-crt-runtime-l1-1-0.dll ausente?

Isso pode ocorrer no Python 3.5 e posterior ao usar o Windows 8.1 ou anterior sem que todas as atualizações tenham sido instaladas. Primeiro, certifique-se de que seu sistema operacional seja compatível e esteja atualizado e, se isso não resolver o problema, visite a página de suporte da Microsoft para obter orientação sobre como instalar manualmente a atualização do C Runtime.

CAPÍTULO 7

FAQ da Interface Gráfica do Usuário

7.1 Perguntas Gerais sobre a GUI

7.2 Quais toolkits de GUI existem para o Python?

As versões padrão do Python incluem uma interface orientada a objetos para o conjunto de widgets Tcl/Tk, chamado tkinter. Este é provavelmente o mais fácil de instalar (uma vez que vem incluído na maioria das distribuições binárias do Python) e usar. Para obter mais informações sobre o Tk, incluindo ponteiros para a fonte, consulte a página inicial do Tcl/Tk. Tcl/Tk é totalmente portátil para as plataformas macOS, Windows e Unix.

Dependendo da(s) plataforma(s) que você está visando, também existem várias alternativas. Uma lista de frameworks GUI de plataformas cruzadas e frameworks GUI específicas de plataforma podem ser encontradas na wiki do python.

7.3 Perguntas do Tkinter

7.3.1 Como eu congelo as aplicações Tkinter?

Freeze é uma ferramenta para criar aplicativos autônomos. Ao congelar aplicativos Tkinter, os aplicativos não serão verdadeiramente autônomos, pois o aplicativo ainda precisará das bibliotecas Tcl e Tk.

Uma solução é enviar a aplicação com as bibliotecas Tcl e Tk e apontá-las em tempo de execução usando as variáveis de ambiente `TCL_LIBRARY` e `TK_LIBRARY`.

Para obter aplicativos verdadeiramente autônomos, os scripts Tcl que formam a biblioteca também precisam ser integrados no aplicativo. Uma ferramenta que suporta isso é SAM (módulos autônomos), que faz parte da distribuição Tix (https://tix.sourceforge.net/).

Compile o Tix com SAM habilitado, execute a chamada apropriada para `Tclsam_init()` etc. dentro do `Modules/tkappinit.c` do Python e faça um link com libtclsam e libtksam (você também pode incluir as bibliotecas Tix).

7.3.2 Posso ter eventos Tk manipulados enquanto aguardo pelo E/S?

Em plataformas diferentes do Windows, sim, e você nem precisa de threads! Mas você terá que reestruturar seu código de E/S um pouco. O Tk tem o equivalente à chamada `XtAddInput()` do Xt, que permite que você registre uma função de retorno de chamada que será chamada a partir do loop principal do Tk quando E/S é possível em um descritor de arquivo. Consulte tkinter-file-handlers.

7.3.3 Não consigo fazer as ligações de tecla funcionarem no Tkinter: por que?

Uma queixa frequentemente ouvida é que os manipuladores de eventos vinculados a eventos com o método `bind()` não são manipulados mesmo quando a tecla apropriada é pressionada.

A causa mais comum é que o widget para o qual a ligação se aplica não possui "foco no teclado". Confira a documentação do Tk para o comando de foco. Normalmente, um widget é dado o foco do teclado clicando nele (mas não para rótulos, veja a opção takefocus).

CAPÍTULO 8

FAD de “Por que o Python está instalado em meu computador?”

8.1 O que é Python?

Python é uma linguagem de programação. É usada para muitas e diversas aplicações. É usada em escolas e faculdades como uma linguagem de programação introdutória, porque Python é fácil de aprender, mas também é usada por desenvolvedores profissionais de software em lugares como Google, Nasa e Lucasfilm Ltd.

Se você quiser aprender mais sobre Python, comece com o Beginner's Guide to Python.

8.2 Porque Python está instalado em minha máquina?

Se você encontrar Python instalado em seu sistema, mas não se lembra de tê-lo instalado, então podem haver muitas maneiras diferentes dele ter ido parar lá

- Possivelmente outro usuário do computador pretendia aprender programação e o instalou; você terá que descobrir quem estava usando a máquina e pode o ter instalado.
- Um aplicativo de terceiros pode ter sido instalado na máquina e sido escrito em Python e incluído uma instalação do Python. Há muitos desses aplicativos, desde programas com interface gráfica até servidores de rede e scripts administrativos.
- Algumas máquinas Windows já possuem o Python instalado. No presente momento nós temos conhecimento de computadores da Hewlett-Packard e da Compaq que incluem Python. Aparentemente algumas das ferramentas administrativas da HP/Compaq são escritas em Python.
- Muitos sistemas operacionais derivados do Unix, como macOS e algumas distribuições Linux, possuem o python instalado por padrão; está incluído na instalação base.

8.3 Eu posso apagar o Python?

Isso depende de como o Python veio.

Se alguém o instalou deliberadamente, você pode removê-lo sem machucar ninguém. No Windows use o Adicionar ou remover programas que se encontra no Painel de Controle.

Se o Python foi instalado por um aplicativo de terceiros, você pode removê-lo mas aquela aplicação não irá mais funcionar. Você deve usar o desinstalador daquele aplicativo para remover o Python diretamente.

Se o Python veio junto com seu sistema operacional, removê-lo não é recomendado. Se você o remover, qualquer ferramenta que tenha sido escrita em Python não vão mais funcionar, e algumas delas podem ser importantes para você. A reinstalação do sistema inteiro seria necessária para consertar as coisas de novo.

APÊNDICE A

Glossário

>>>

O prompt padrão do console interativo do Python. Normalmente visto em exemplos de código que podem ser executados interativamente no interpretador.

...

Pode se referir a:

- O prompt padrão do shell interativo do Python ao inserir o código para um bloco de código recuado, quando dentro de um par de delimitadores correspondentes esquerdo e direito (parênteses, colchetes, chaves ou aspas triplas) ou após especificar um decorador.
- A constante embutida `Ellipsis`.

2to3

Uma ferramenta que tenta converter código Python 2.x em código Python 3.x tratando a maioria das incompatibilidades que podem ser detectadas com análise do código-fonte e navegação na árvore sintática.

O 2to3 está disponível na biblioteca padrão como `lib2to3`; um ponto de entrada é disponibilizado como `Tools/scripts/2to3`. Veja 2to3-reference.

classe base abstrata

Classes bases abstratas complementam *tipagem pato*, fornecendo uma maneira de definir interfaces quando outras técnicas, como `hasattr()`, seriam desajeitadas ou sutilmente erradas (por exemplo, com métodos mágicos). CBAs introduzem subclasses virtuais, classes que não herdam de uma classe mas ainda são reconhecidas por `isinstance()` e `issubclass()`; veja a documentação do módulo `abc`. Python vem com muitas CBAs embutidas para estruturas de dados (no módulo `collections.abc`), números (no módulo `numbers`), fluxos (no módulo `io`), localizadores e carregadores de importação (no módulo `importlib.abc`). Você pode criar suas próprias CBAs com o módulo `abc`.

anotação

Um rótulo associado a uma variável, um atributo de classe ou um parâmetro de função ou valor de retorno, usado por convenção como *dica de tipo*.

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 `__annotations__` de módulos, classes e funções, respectivamente.

Veja *anotação de variável*, *anotação de função*, **PEP 484** e **PEP 526**, que descrevem esta funcionalidade. Veja também annotations-howto para as melhores práticas sobre como trabalhar com anotações.

argumento

Um valor passado para uma *função* (ou *método*) ao chamar a função. Existem dois tipos de argumento:

- *argumento nomeado*: um argumento precedido por um identificador (por exemplo, `name=`) na chamada de uma função ou passada como um valor em um dicionário precedido por `**`. Por exemplo, `3` e `5` são ambos argumentos nomeados na chamada da função `complex()` a seguir:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argumento posicional*: um argumento que não é um argumento nomeado. Argumentos posicionais podem aparecer no início da lista de argumentos e/ou podem ser passados com elementos de um *iterável* precedido por `*`. Por exemplo, `3` e `5` são ambos argumentos posicionais nas chamadas a seguir:

```
complex(3, 5)
complex(*(3, 5))
```

Argumentos são atribuídos às variáveis locais nomeadas no corpo da função. Veja a seção calls para as regras de atribuição. Sintaticamente, qualquer expressão pode ser usada para representar um argumento; avaliada a expressão, o valor é atribuído à variável local.

Veja também o termo *parâmetro* no glossário, a pergunta no FAQ sobre *a diferença entre argumentos e parâmetros* e **PEP 362**.

gerenciador de contexto assíncrono

Um objeto que controla o ambiente visto numa instrução `async with` por meio da definição dos métodos `__aenter__()` e `__aexit__()`. Introduzido pela **PEP 492**.

gerador assíncrono

Uma função que retorna um *iterador gerador assíncrono*. É parecida com uma função de corrotina definida com `async def` exceto pelo fato de conter instruções `yield` para produzir uma série de valores que podem ser usados em um laço `async for`.

Normalmente se refere a uma função geradora assíncrona, mas pode se referir a um *iterador gerador assíncrono* em alguns contextos. Em casos em que o significado não esteja claro, usar o termo completo evita a ambiguidade.

Uma função geradora assíncrona pode conter expressões `await` e também as instruções `async for` e `async with`.

iterador gerador assíncrono

Um objeto criado por uma função *geradora assíncrona*.

Este é um *iterador assíncrono* que, quando chamado usando o método `__anext__()`, retorna um objeto aguardável que executará o corpo da função geradora assíncrona até a próxima expressão `yield`.

Cada `yield` suspende temporariamente o processamento, lembrando o estado de execução do local (incluindo variáveis locais e instruções `try` pendentes). Quando o *iterador gerador assíncrono* é efetivamente retomado com outro aguardável retornado por `__anext__()`, ele inicia de onde parou. Veja **PEP 492** e **PEP 525**.

iterável assíncrono

Um objeto que pode ser usado em uma instrução `async for`. Deve retornar um *iterador assíncrono* do seu método `__aiter__()`. Introduzido por **PEP 492**.

iterador assíncrono

Um objeto que implementa os métodos `__aiter__()` e `__anext__()`. `__anext__()` deve retornar um objeto *aguardável*. `async for` resolve os aguardáveis retornados por um método `__anext__()` do iterador assíncrono até que ele levante uma exceção `StopAsyncIteration`. Introduzido pela **PEP 492**.

atributo

Um valor associado a um objeto que é geralmente referenciado pelo nome separado por um ponto. Por exemplo, se um objeto *o* tem um atributo *a* esse seria referenciado como *o.a*.

É possível dar a um objeto um atributo cujo nome não seja um identificador conforme definido por identifiers, por exemplo usando `setattr()`, se o objeto permitir. Tal atributo não será acessível usando uma expressão pontilhada e, em vez disso, precisaria ser recuperado com `getattr()`.

aguardável
Um objeto que pode ser usado em uma expressão `await`. Pode ser uma *corrotina* ou um objeto com um método `__await__()`. Veja também a **PEP 492**.

BDFL
Abreviação da expressão da língua inglesa "Benevolent Dictator for Life" (em português, "Ditador Benevolente Vitalício"), referindo-se a Guido van Rossum, criador do Python.

arquivo binário
Um *objeto arquivo* capaz de ler e gravar em *objetos bytes 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 objeto arquivo capaz de ler e gravar em objetos `str`.

referência emprestada
Na API C do Python, uma referência emprestada é uma referência a um objeto que não é dona da referência. Ela se torna um ponteiro solto se o objeto for destruído. Por exemplo, uma coleta de lixo pode remover a última *referência forte* para o objeto e assim destruí-lo.

Chamar `Py_INCREF()` na *referência emprestada* é recomendado para convertê-lo, internamente, em uma *referência forte*, exceto quando o objeto não pode ser destruído antes do último uso da referência emprestada. A função `Py_NewRef()` pode ser usada para criar uma nova *referência forte*.

objeto byte ou similar
Um objeto com suporte ao o bufferobjects e que pode exportar um buffer C *contíguo*. Isso inclui todos os objetos `bytes`, `bytearray` e `array.array`, além de muitos objetos `memoryview` comuns. 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 similar para leitura e 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
O código-fonte Python é compilado para bytecode, a representação interna de um programa em Python no interpretador CPython. O bytecode também é mantido em cache em arquivos `.pyc` e `.pyo`, de forma que executar um mesmo arquivo é mais rápido na segunda vez (a recompilação dos fontes para bytecode não é necessária). Esta "linguagem intermediária" é adequada para execução em uma *máquina virtual*, que executa o código de máquina correspondente para cada bytecode. Tenha em mente que não se espera que bytecodes sejam executados entre máquinas virtuais Python diferentes, nem que se mantenham estáveis entre versões de Python.

Uma lista de instruções bytecode pode ser encontrada na documentação para o módulo dis.

chamável
Um chamável é um objeto que pode ser chamado, possivelmente com um conjunto de argumentos (veja *argumento*), com a seguinte sintaxe:

```
chamavel(argumento1, argumento2, argumentoN)
```

Uma *função*, e por extensão um *método*, é um chamável. Uma instância de uma classe que implementa o método `__call__()` também é um chamável.

função de retorno
Também conhecida como callback, é uma função sub-rotina que é passada como um argumento a ser executado em algum ponto no futuro.

classe
Um modelo para criação de objetos definidos pelo usuário. Definições de classe normalmente contém definições de métodos que operam sobre instâncias da classe.

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).

número complexo
Uma extensão ao familiar sistema de números reais em que todos os números são expressos como uma soma de uma parte real e uma parte imaginária. Números imaginários são múltiplos reais da unidade imaginária (a raiz quadrada de `-1`), normalmente escrita como `i` em matemática ou `j` em engenharia. O Python tem suporte nativo para números complexos, que são escritos com esta última notação; a parte imaginária escrita com um sufixo `j`, p.ex., `3+1j`. Para ter acesso aos equivalentes para números complexos do módulo `math`, utilize `cmath`. O uso de números complexos é uma funcionalidade matemática bastante avançada. Se você não sabe se irá precisar deles, é quase certo que você pode ignorá-los sem problemas.

gerenciador de contexto
Um objeto que controla o ambiente visto numa instrução `with` por meio da definição dos métodos `__enter__()` e `__exit__()`. Veja **PEP 343**.

variável de contexto
Uma variável que pode ter valores diferentes, dependendo do seu contexto. Isso é semelhante ao armazenamento local de threads, no qual cada thread pode ter um valor diferente para uma variável. No entanto, com variáveis de contexto, pode haver vários contextos em uma thread e o principal uso para variáveis de contexto é acompanhar as variáveis em tarefas assíncronas simultâneas. Veja `contextvars`.

contíguo
Um buffer é considerado contíguo exatamente se for *contíguo C* ou *contíguo Fortran*. Os buffers de dimensão zero são contíguos C e Fortran. Em vetores unidimensionais, os itens devem ser dispostos na memória próximos um do outro, em ordem crescente de índices, começando do zero. Em vetores multidimensionais contíguos C, o último índice varia mais rapidamente ao visitar itens em ordem de endereço de memória. No entanto, nos vetores contíguos do Fortran, o primeiro índice varia mais rapidamente.

corrotina
Corrotinas são uma forma mais generalizada de sub-rotinas. Sub-rotinas tem a entrada iniciada em um ponto, e a saída em outro ponto. Corrotinas podem entrar, sair, e continuar em muitos pontos diferentes. Elas podem ser implementadas com a instrução `async def`. Veja também **PEP 492**.

função de corrotina
Uma função que retorna um objeto do tipo *corrotina*. Uma função de corrotina 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
A implementação canônica da linguagem de programação Python, como disponibilizada pelo python.org. O termo "CPython" é usado quando necessário distinguir esta implementação de outras como Jython ou 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(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

O mesmo conceito existe para as classes, mas não é comumente utilizado. Veja a documentação de definições de função e definições de classe para obter mais informações sobre decoradores.

descritor
Qualquer objeto que define os métodos `__get__()`, `__set__()` ou `__delete__()`. Quando um atributo de classe é um descritor, seu comportamento de associação especial é acionado no acesso a um atributo. Normalmente, ao se utilizar *a.b* para se obter, definir ou excluir, um atributo dispara uma busca no objeto chamado *b* no dicionário de classe de *a*, mas se *b* for um descritor, o respectivo método descritor é chamado. Compreender descritores é a chave para um profundo entendimento de Python pois eles são a base de muitas funcionalidades incluindo funções, métodos, propriedades, métodos de classe, métodos estáticos e referências para superclasses.

Para obter mais informações sobre os métodos dos descritores, veja: descriptors ou o Guia de Descritores.

dicionário
Um vetor associativo em que chaves arbitrárias são mapeadas para valores. As chaves podem ser quaisquer objetos que possuam os métodos `__hash__()` e `__eq__()`. Isso é chamado de hash em Perl.

compreensão de dicionário
Uma maneira compacta de processar todos ou parte dos elementos de um iterável e retornar um dicionário com os resultados. `results = {n: n ** 2 for n in range(10)}` gera um dicionário contendo a chave `n` mapeada para o valor `n ** 2`. Veja comprehensions.

visão de dicionário
Os objetos retornados por `dict.keys()`, `dict.values()` e `dict.items()` são chamados de visões de dicionário. Eles fornecem uma visão dinâmica das entradas do dicionário, o que significa que quando o dicionário é alterado, a visão reflete essas alterações. Para forçar a visão de dicionário a se tornar uma lista completa use `list(dictview)`. Veja dict-views.

docstring
Abreviatura de "documentation string" (string de documentação). 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.

tipagem pato
Também conhecida como *duck-typing*, é um estilo de programação que não verifica o tipo do objeto para determinar se ele possui a interface correta; em vez disso, o método ou atributo é simplesmente chamado ou utilizado ("Se se parece com um pato e grasna como um pato, então deve ser um pato.") Enfatizando interfaces ao invés de tipos específicos, o código bem desenvolvido aprimora sua flexibilidade por permitir substituição polimórfica. Tipagem pato evita necessidade de testes que usem `type()` ou `isinstance()`. (Note, porém, que a tipagem pato pode ser complementada com o uso de *classes base abstratas*.) Ao invés disso, são normalmente empregados testes `hasattr()` ou programação *EAFP*.

EAFP
Iniciais da expressão em inglês "easier to ask for forgiveness than permission" que significa "é mais fácil pedir perdão que permissão". Este estilo de codificação comum no Python presume a existência de chaves ou atributos válidos e captura exceções caso essa premissa se prove falsa. Este estilo limpo e rápido se caracteriza pela presença de várias instruções `try` e `except`. A técnica diverge do estilo *LBYL*, comum em outras linguagens como C, por exemplo.

expressão
Uma parte da sintaxe que pode ser avaliada para algum valor. Em outras palavras, uma expressão é a acumulação de elementos de expressão como literais, nomes, atributos de acesso, operadores ou chamadas de funções, todos os quais retornam um valor. Em contraste com muitas outras linguagens, nem todas as construções de linguagem são expressões. Também existem *instruções*, as quais não podem ser usadas como expressões, como, por exemplo, `while`. Atribuições também são instruções, não expressões.

módulo de extensão
Um módulo escrito em C ou C++, usando a API C do 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**.

objeto arquivo

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, soquetes, pipes, etc.). Objetos arquivo também são chamados de *objetos arquivo ou similares* ou *fluxos*.

Atualmente há três categorias de objetos arquivo: *arquivos binários* brutos, *arquivos binários* em buffer e *arquivos textos*. Suas interfaces estão definidas no módulo `io`. A forma canônica para criar um objeto arquivo é usando a função `open()`.

objeto arquivo ou similar

Um sinônimo do termo *objeto arquivo*.

tratador de erros e codificação do sistema de arquivos

Tratador de erros e codificação usado pelo Python para decodificar bytes do sistema operacional e codificar Unicode para o sistema operacional.

A codificação do sistema de arquivos deve garantir a decodificação bem-sucedida de todos os bytes abaixo de 128. Se a codificação do sistema de arquivos falhar em fornecer essa garantia, as funções da API podem levantar `UnicodeError`.

As funções `sys.getfilesystemencoding()` e `sys.getfilesystemencodeerrors()` podem ser usadas para obter o tratador de erros e codificação do sistema de arquivos.

O *tratador de erros e codificação do sistema de arquivos* são configurados na inicialização do Python pela função `PyConfig_Read()`: veja os membros `filesystem_encoding` e `filesystem_errors` do `PyConfig`.

Veja também a *codificação da localidade*.

localizador

Um objeto que tenta encontrar o *carregador* para um módulo que está sendo importado.

Existem dois tipos de localizador: *localizadores de metacaminho* para uso com `sys.meta_path`, e *localizadores de entrada de caminho* para uso com `sys.path_hooks`.

See finders-and-loaders and `importlib` for much more detail.

divisão pelo piso

Divisão matemática que arredonda para baixo para o inteiro mais próximo. O operador de divisão pelo piso é `//`. Por exemplo, a expressão `11 // 4` retorna o valor `2` ao invés de `2.75`, que seria retornado pela divisão de ponto flutuante. Note que `(-11) // 4` é `-3` porque é `-2.75` arredondado *para baixo*. Consulte a **PEP 238**.

função

Uma série de instruções que retorna algum valor para um chamador. Também pode ser passado zero ou mais *argumentos* que podem ser usados na execução do corpo. Veja também *parâmetro*, *método* e a seção function.

anotação de função

Uma *anotação* de um parâmetro de função ou valor de retorno.

Anotações de função são comumente usados por *dicas de tipo*: por exemplo, essa função espera receber dois argumentos `int` e também é esperado que devolva um valor `int`:

```
def soma_dois_numeros(a: int, b: int) -> int:
   return a + b
```

A sintaxe de anotação de função é explicada na seção function.

Veja *anotação de variável* e **PEP 484**, que descrevem esta funcionalidade. Veja também annotations-howto para as melhores práticas sobre como trabalhar com anotações.

__future__

A instrução future, `from __future__ import <feature>`, direciona o compilador a compilar o módulo atual usando sintaxe ou semântica que será padrão em uma versão futura de Python. O módulo `__future__` documenta os possíveis valores de *feature*. Importando esse módulo e avaliando suas variáveis, você pode ver quando um novo recurso foi inicialmente adicionado à linguagem e quando será (ou se já é) o padrão:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

coleta de lixo

Também conhecido como *garbage collection*, é 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

Uma função que retorna um *iterador gerador*. É parecida com uma função normal, exceto pelo fato de conter expressões `yield` para produzir uma série de valores que podem ser usados em um laço "for" ou que podem ser obtidos um de cada vez com a função `next()`.

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 *geradora*.

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).

expressão geradora

Uma expressão que retorna um iterador. Parece uma expressão normal, seguido de uma cláusula `for` definindo uma variável de loop, um range, e uma cláusula `if` opcional. A expressão combinada gera valores para uma função encapsuladora:

```
>>> sum(i*i for i in range(10))         # soma dos quadrados 0, 1, 4, ... 81
285
```

função genérica

Uma função composta por várias 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 *despacho único* no glossário, o decorador `functools.singledispatch()`, e a **PEP 443**.

tipo genérico

Um *tipo* que pode ser parametrizado; tipicamente uma classe contêiner tal como `list` ou `dict`. Usado para *dicas de tipo* e *anotações*.

Para mais detalhes, veja tipo apelido genérico, **PEP 483**, **PEP 484**, **PEP 585**, e o módulo `typing`.

GIL

Veja *trava global do interpretador*.

trava global do interpretador

O mecanismo utilizado pelo interpretador *CPython* para garantir que apenas uma thread execute o *bytecode* Python por vez. Isto simplifica a implementação do CPython ao fazer com que o modelo de objetos (incluindo tipos embutidos críticos como o `dict`) ganhem segurança implícita contra acesso concorrente. Travar todo o interpretador facilita que o interpretador em si seja multitarefa, às custas de muito do paralelismo já provido por máquinas multiprocessador.

No entanto, alguns módulos de extensão, tanto da biblioteca padrão quanto de terceiros, são desenvolvidos de forma a liberar a GIL ao realizar tarefas computacionalmente muito intensas, como compactação ou cálculos

de hash. Além disso, a GIL é sempre liberado nas operações de E/S.

No passado, esforços para criar um interpretador que lidasse plenamente com threads (travando dados compartilhados numa granularidade bem mais fina) não foram bem sucedidos devido a queda no desempenho ao serem executados em processadores de apenas um núcleo. Acredita-se que superar essa questão de desempenho acabaria tornando a implementação muito mais complicada e bem mais difícil de manter.

pyc baseado em hash
Um arquivo de cache em bytecode que usa hash ao invés do tempo, no qual o arquivo de código-fonte foi modificado pela última vez, para determinar a sua validade. Veja pyc-invalidation.

hasheável
Um objeto é *hasheável* se tem um valor de hash que nunca muda durante seu ciclo de vida (precisa ter um método `__hash__()`) e pode ser comparado com outros objetos (precisa ter um método `__eq__()`). Objetos hasheáveis que são comparados como iguais devem ter o mesmo valor de hash.

A hasheabilidade faz com que um objeto possa ser usado como uma chave de dicionário e como um membro de conjunto, pois estas estruturas de dados utilizam os valores de hash internamente.

A maioria dos objetos embutidos imutáveis do Python são hasheáveis; containers mutáveis (tais como listas ou dicionários) não são; containers imutáveis (tais como tuplas e frozensets) são hasheáveis apenas se os seus elementos são hasheáveis. Objetos que são instâncias de classes definidas pelo usuário são hasheáveis por padrão. Todos eles comparam de forma desigual (exceto entre si mesmos), e o seu valor hash é derivado a partir do seu `id()`.

IDLE
Um ambiente de desenvolvimento e aprendizado integrado para Python. idle é um editor básico e um ambiente interpretador que vem junto com a distribuição padrão do Python.

imortal
Objetos imortais são um detalhe da implementação do CPython introduzida na **PEP 683**.

Se um objeto é imortal, sua *contagem de referências* nunca é modificada e, portanto, nunca é desalocado enquanto o interpretador está em execução. Por exemplo, `True` e `None` são imortais no CPython.

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.

caminho de importação
Uma lista de localizações (ou *entradas de caminho*) que são buscadas pelo *localizador baseado no caminho* por módulos para importar. Durante a importação, esta lista de localizações usualmente vem a partir de `sys.path`, mas para subpacotes ela também pode vir do atributo `__path__` de pacotes-pai.

importação
O processo pelo qual o código Python em um módulo é disponibilizado para o código Python em outro módulo.

importador
Um objeto que localiza e carrega um módulo; Tanto um *localizador* e o objeto *carregador*.

interativo
Python tem um interpretador interativo, o que significa que você pode digitar instruções e expressões no prompt do interpretador, executá-los imediatamente e ver seus resultados. Apenas execute `python` sem argumentos (possivelmente selecionando-o a partir do menu de aplicações de seu sistema operacional). O interpretador interativo é uma maneira poderosa de testar novas ideias ou aprender mais sobre módulos e pacotes (lembre-se do comando `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*.

desligamento do interpretador

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 *coletor de lixo*. Isto pode disparar a execução de código em destrutores definidos pelo usuário ou função de retorno 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`, *objetos arquivos*, 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 embutida `iter()`, ela retorna um iterador para o objeto. Este iterador é adequado para se varrer todo o conjunto de valores. Ao usar iteráveis, normalmente não é necessário chamar `iter()` ou lidar com os objetos iteradores em si. A instrução `for` faz isso automaticamente para você, criando uma variável temporária para armazenar o iterador durante a execução do laço. Veja também *iterador*, *sequência*, e *gerador*.

iterador

Um objeto que representa um fluxo de dados. Repetidas chamadas ao método `__next__()` de um iterador (ou passando o objeto para a função embutida `next()`) vão retornar itens sucessivos do fluxo. Quando não houver mais dados disponíveis uma exceção `StopIteration` 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.

Detalhes da implementação do CPython: O CPython não aplica consistentemente o requisito que um iterador define `__iter__()`.

função chave

Uma função chave ou função colação é um chamável que retorna um valor usado para ordenação ou classificação. Por exemplo, `locale.strxfrm()` é usada para produzir uma chave de ordenação que leva o locale em consideração para fins de ordenação.

Uma porção de ferramentas no Python aceitam funções chave para controlar como os elementos são ordenados ou agrupados. Algumas delas incluem `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` e `itertools.groupby()`.

Há várias maneiras de se criar funções chave. Por exemplo, o método `str.lower()` pode servir como uma função chave para ordenações insensíveis à caixa. Alternativamente, uma função chave ad-hoc pode ser construída a partir de uma expressão `lambda`, como `lambda r: (r[0], r[2])`. Além disso, `operator.attrgetter()`, `operator.itemgetter()` e `operator.methodcaller()` são três construtores de função chave. Consulte o guia de Ordenação para ver exemplos de como criar e utilizar funções chave.

argumento nomeado

Veja *argumento*.

lambda

Uma função de linha anônima consistindo de uma única *expressão*, que é avaliada quando a função é chamada. A sintaxe para criar uma função lambda é `lambda [parameters]: expression`

LBYL

Iniciais da expressão em inglês "look before you leap", que significa algo como "olhe antes de pisar". Este estilo de codificação testa as pré-condições explicitamente antes de fazer chamadas ou buscas. Este estilo contrasta com a abordagem *EAFP* e é caracterizada pela presença de muitas instruções `if`.

Em um ambiente multithread, a abordagem LBYL pode arriscar a introdução de uma condição de corrida entre "o olhar" e "o pisar". Por exemplo, o código `if key in mapping: return mapping[key]` pode falhar se outra thread remover *key* do *mapping* após o teste, mas antes da olhada. Esse problema pode ser resolvido com travas ou usando a abordagem EAFP.

lista

Uma *sequência* 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 $O(1)$.

compreensão de lista

Uma maneira compacta de processar todos ou parte dos elementos de uma sequência e retornar os resultados em uma lista. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` gera uma lista de strings contendo números hexadecimais (0x..) no intervalo de 0 a 255. A cláusula `if` é opcional. Se omitida, todos os elementos no `range(256)` serão processados.

carregador

An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See also:

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

codificação da localidade

No Unix, é a codificação da localidade do LC_CTYPE, que pode ser definida com `locale.setlocale(locale.LC_CTYPE, new_locale)`.

No Windows, é a página de código ANSI (ex: `"cp1252"`).

No Android e no VxWorks, o Python usa `"utf-8"` como a codificação da localidade.

`locale.getencoding()` pode ser usado para obter a codificação da localidade.

Veja também *tratador de erros e codificação do sistema de arquivos*.

método mágico

Um sinônimo informal para um *método especial*.

mapeamento

Um objeto contêiner que tem suporte a pesquisas de chave arbitrária e implementa os métodos especificados nas `collections.abc.Mapping` ou `collections.abc.MutableMapping` classes base abstratas. Exemplos incluem `dict`, `collections.defaultdict`, `collections.OrderedDict` e `collections.Counter`.

localizador de metacaminho

Um *localizador* retornado por uma busca de `sys.meta_path`. Localizadores de metacaminho são relacionados a, mas diferentes de, *localizadores de entrada de caminho*.

Veja `importlib.abc.MetaPathFinder` para os métodos que localizadores de metacaminho implementam.

metaclasse

A classe de uma classe. Definições de classe criam um nome de classe, um dicionário de classe e uma lista de classes base. A metaclasse é responsável por receber estes três argumentos e criar a classe. A maioria das linguagens de programação orientadas a objetos provê uma implementação default. O que torna o Python especial é o fato de ser possível criar metaclasses personalizadas. A maioria dos usuários nunca vai precisar deste recurso, mas quando houver necessidade, metaclasses possibilitam soluções poderosas e elegantes. Metaclasses têm sido utilizadas para gerar registros de acesso a atributos, para incluir proteção contra acesso concorrente, rastrear a criação de objetos, implementar singletons, dentre muitas outras tarefas.

Mais informações podem ser encontradas em metaclasses.

método
Uma função que é definida dentro do corpo de uma classe. Se chamada como um atributo de uma instância daquela classe, o método receberá a instância do objeto como seu primeiro *argumento* (que comumente é chamado de `self`). Veja *função* e *escopo aninhado*.

ordem de resolução de métodos
Ordem de resolução de métodos é a ordem em que os membros de uma classe base são buscados durante a pesquisa. Veja python_2.3_mro para detalhes do algoritmo usado pelo interpretador do Python desde a versão 2.3.

módulo
Um objeto que serve como uma unidade organizacional de código Python. Os módulos têm um espaço de nomes contendo objetos Python arbitrários. Os módulos são carregados pelo Python através do processo de *importação*.

Veja também *pacote*.

módulo spec
Um espaço de nomes que contém as informações relacionadas à importação usadas para carregar um módulo. Uma instância de `importlib.machinery.ModuleSpec`.

See also module-specs.

MRO
Veja *ordem de resolução de métodos*.

mutável
Objeto mutável é aquele que pode modificar seus valor mas manter seu `id()`. Veja também *imutável*.

tupla nomeada
O termo "tupla nomeada" é aplicado a qualquer tipo ou classe que herda de tupla e cujos elementos indexáveis também são acessíveis usando atributos nomeados. O tipo ou classe pode ter outras funcionalidades também.

Diversos tipos embutidos são tuplas nomeadas, incluindo os valores retornados por `time.localtime()` e `os.stat()`. Outro exemplo é `sys.float_info`:

```
>>> sys.float_info[1]                    # acesso indexado
1024
>>> sys.float_info.max_exp               # acesso a campo nomeado
1024
>>> isinstance(sys.float_info, tuple)    # tipo de tupla
True
```

Algumas tuplas nomeadas são tipos embutidos (tal como os exemplos acima). Alternativamente, uma tupla nomeada pode ser criada a partir de uma definição de classe regular, que herde de `tuple` e que defina campos nomeados. Tal classe pode ser escrita a mão, ou ela pode ser criada herdando `typing.NamedTuple` ou com uma função fábrica `collections.namedtuple()`. As duas últimas técnicas também adicionam alguns métodos extras, que podem não ser encontrados quando foi escrita manualmente, ou em tuplas nomeadas embutidas.

espaço de nomes
O lugar em que uma variável é armazenada. Espaços de nomes são implementados como dicionários. Existem os espaços de nomes local, global e nativo, bem como espaços de nomes aninhados em objetos (em métodos). Espaços de nomes suportam modularidade ao prevenir conflitos de nomes. Por exemplo, as funções `__builtin__.open()` e `os.open()` são diferenciadas por seus espaços de nomes. Espaços de nomes 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.

pacote de espaço de nomes
Um *pacote* da **PEP 420** que serve apenas como container para sub pacotes. Pacotes de espaços de nomes podem não ter representação física, e especificamente não são como um *pacote regular* porque eles não tem um arquivo `__init__.py`.

Veja também *módulo*.

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 espaço de nomes global. O `nonlocal` permite escrita para escopos externos.

classe estilo novo
Antigo nome para o tipo de classes agora usado para todos os objetos de classes. Em versões anteriores do Python, apenas classes estilo podiam usar recursos novos e versáteis do Python, tais como `__slots__`, descritores, propriedades, `__getattribute__()`, métodos de classe, e métodos estáticos.

objeto
Qualquer dado que tenha estado (atributos ou valores) e comportamento definidos (métodos). Também a última classe base de qualquer *classe estilo novo*.

pacote
Um *módulo* Python é capaz de conter submódulos ou recursivamente, subpacotes. Tecnicamente, um pacote é um módulo Python com um atributo `__path__`.

Veja também *pacote regular* e *pacote de espaço de nomes*.

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 fornecido apenas por posição. Parâmetros somente-posicionais podem ser definidos incluindo o caractere `/` na lista de parâmetros da definição da função após eles, por exemplo *somentepos1* e *somentepos2* a seguir:

```
def func(somentepos1, somentepos2, /, posicional_ou_nomeado): ...
```

- *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 *somente_nom1* and *somente_nom2* a seguir:

```
def func(arg, *, somente_nom1, somente_nom2): ...
```

- *var-posicional*: especifica que 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 do parâmetro, 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 padrão para alguns argumentos opcionais.

Veja o termo *argumento* no glossário, a pergunta sobre *a diferença entre argumentos e parâmetros*, a classe `inspect.Parameter`, a seção function e a **PEP 362**.

entrada de caminho
Um local único no *caminho de importação* que o *localizador baseado no caminho* consulta para encontrar módulos a serem importados.

localizador de entrada de caminho
Um *localizador* retornado por um chamável em `sys.path_hooks` (ou seja, um *gancho de entrada de caminho*) que sabe como localizar os módulos *entrada de caminho*.

Veja `importlib.abc.PathEntryFinder` para os métodos que localizadores de entrada de caminho implementam.

gancho de entrada de caminho
Um chamável na lista `sys.path_hooks` que retorna um *localizador de entrada de caminho* caso saiba como localizar módulos em uma *entrada de caminho* específica.

localizador baseado no caminho
Um dos *localizadores de metacaminho* padrão que procura por um *caminho de importação* de módulos.

objeto caminho ou similar
Um objeto representando um caminho de sistema de arquivos. 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 têm 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**.

porção
Um conjunto de arquivos em um único diretório (possivelmente armazenado em um arquivo zip) que contribuem para um pacote de espaço de nomes, conforme definido em **PEP 420**.

argumento posicional
Veja *argumento*.

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 *API provisória*.

Python 3000
Apelido para a linha de lançamento da versão do Python 3.x (cunhada há muito tempo, quando o lançamento da versão 3 era algo em um futuro muito distante.) Esse termo possui a seguinte abreviação: "Py3k".

Pythônico

Uma ideia ou um pedaço de código que segue de perto as formas de escritas mais comuns da linguagem Python, ao invés de implementar códigos usando conceitos comuns a outras linguagens. Por exemplo, um formato comum em Python é fazer um laço sobre todos os elementos de uma iterável usando a instrução `for`. Muitas outras linguagens 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(comida)):
    print(comida[i])
```

Ao contrário do método mais limpo, Pythônico:

```
for parte in comida:
    print(parte)
```

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 metodo(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.metodo.__qualname__
'C.D.metodo'
```

Quando usado para se referir a módulos, o *nome totalmente qualificado* 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'
```

contagem de referências

O número de referências a um objeto. Quando a contagem de referências de um objeto cai para zero, ele é desalocado. Alguns objetos são "imortais" e têm contagens de referências que nunca são modificadas e, portanto, os objetos nunca são desalocados. A contagem de referências geralmente não é visível para o código Python, mas é um elemento-chave da implementação do *CPython*. Os programadores podem chamar a função `sys.getrefcount()` para retornar a contagem de referências para um objeto específico.

pacote regular

Um *pacote* tradicional, como um diretório contendo um arquivo `__init__.py`.

Veja também *pacote de espaço de nomes*.

__slots__

Uma declaração dentro de uma classe que economiza memória pré-declarando espaço para atributos de 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 *iterável* 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 embutidos são: `list`, `str`, `tuple`, e `bytes`. Note que `dict` também tem suporte para

`__getitem__()` e `__len__()`, mas é considerado um mapeamento e não uma sequência porque a busca usa uma chave *hasheável* 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()`. Para mais documentação sobre métodos de sequências em geral, veja Operações comuns de sequências.

compreensão de conjunto
Uma maneira compacta de processar todos ou parte dos elementos em iterável e retornar um conjunto com os resultados. `results = {c for c in 'abracadabra' if c not in 'abc'}` gera um conjunto de strings `{'r', 'd'}`. Veja comprehensions.

despacho único
Uma forma de despacho de *função genérica* onde a implementação é escolhida com base no tipo de um único argumento.

fatia
Um objeto geralmente contendo uma parte de uma *sequência*. 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.

suavemente descontinuado
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

Veja PEP 387: Descontinuação suave.

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 estão documentados em specialnames.

instrução
Uma instrução é parte de uma suíte (um "bloco" de código). Uma instrução é ou uma *expressão* ou uma de várias construções com uma palavra reservada, tal como `if`, `while` ou `for`.

verificador de tipo estático
Uma ferramenta externa que lê o código Python e o analisa, procurando por problemas como tipos incorretos. Consulte também *dicas de tipo* e o módulo `typing`.

referência forte
Na API C do Python, uma referência forte é uma referência a um objeto que pertence ao código que contém a referência. A referência forte é obtida chamando `Py_INCREF()` quando a referência é criada e liberada com `Py_DECREF()` quando a referência é excluída.

A função `Py_NewRef()` pode ser usada para criar uma referência forte para um objeto. Normalmente, a função `Py_DECREF()` deve ser chamada na referência forte antes de sair do escopo da referência forte, para evitar o vazamento de uma referência.

Veja também *referência emprestada*.

codificador de texto
Uma string em Python é uma sequência de pontos de código Unicode (no intervalo `U+0000–U+10FFFF`). Para armazenar ou transferir uma string, ela precisa ser serializada como uma sequência de bytes.

A serialização de uma string em uma sequência de bytes é conhecida como "codificação" e a recriação da string a partir de uma sequência de bytes é conhecida como "decodificação".

Há uma variedade de diferentes serializações de texto codecs, que são coletivamente chamadas de "codificações de texto".

arquivo texto
Um *objeto arquivo* apto a ler e escrever objetos `str`. Geralmente, um arquivo texto, na verdade, acessa um

fluxo de dados de bytes e captura o *codificador de texto* 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 *arquivo binário* para um objeto arquivo apto a ler e escrever *objetos byte ou similar*.

string entre aspas triplas
Uma string que está definida com três ocorrências de aspas duplas (") ou apóstrofos ('). 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 escapadas dentro de uma string, e elas podem utilizar múltiplas linhas sem o uso de caractere de continuação, fazendo-as especialmente úteis quando escrevemos documentação em docstrings.

tipo
O tipo de um objeto Python determina qual classe de objeto ele é; cada objeto tem um tipo. Um tipo de objeto é acessível pelo atributo `__class__` ou pode ser recuperado com `type(obj)`.

apelido de tipo
Um sinônimo para um tipo, criado através da atribuição do tipo para um identificador.

Apelidos de tipo são úteis para simplificar *dicas de tipo*. Por exemplo:

```
def remove_tons_de_cinza(
        cores: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

pode tornar-se mais legível desta forma:

```
Cor = tuple[int, int, int]

def remove_tons_de_cinza(cores: list[Cor]) -> list[Cor]:
    pass
```

Veja `typing` e **PEP 484**, a qual descreve esta funcionalidade.

dica de tipo
Uma *anotação* 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 *verificadores de tipo estático*. Eles também 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**, a qual descreve esta funcionalidade.

novas linhas universais
Uma maneira de interpretar fluxos de textos, na qual todos estes são reconhecidos como caracteres de fim 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 de variável
Uma *anotação* de uma variável ou um atributo de classe.

Ao fazer uma anotação de uma variável ou um atributo de classe, a atribuição é opcional:

```
class C:
    campo: 'anotação'
```

Anotações de variáveis são normalmente usadas para *dicas de tipo*: por exemplo, espera-se que esta variável receba valores do tipo `int`:

```
contagem: int = 0
```

A sintaxe de anotação de variável é explicada na seção annassign.

Veja *anotação de função*, **PEP 484** e **PEP 526**, que descrevem esta funcionalidade. Veja também annotations--howto para as melhores práticas sobre como trabalhar com anotações.

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`.

máquina virtual
Um computador definido inteiramente em software. A máquina virtual de Python executa o *bytecode* emitido pelo compilador de bytecode.

Zen do 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

Esses documentos são gerados a partir de reStructuredText pelo Sphinx, um processador de documentos especificamente escrito para documentação Python.

O desenvolvimento da documentação e de suas ferramentas é um esforço totalmente voluntário, como Python em si. Se você quer contribuir, por favor dê uma olhada na página reporting-bugs para informações sobre como fazer. Novos voluntários são sempre bem-vindos!

Agradecimentos especiais para:

- Fred L. Drake, Jr., o criador do primeiro conjunto de ferramentas para documentar Python e escritor de boa parte do conteúdo;
- O projeto Docutils por criar reStructuredText e o pacote Docutils;
- Fredrik Lundh, pelo seu projeto de referência alternativa em Python, do qual Sphinx pegou muitas boas ideias.

B.1 Contribuidores da Documentação Python

Muitas pessoas tem contribuído para a linguagem Python, sua biblioteca padrão e sua documentação. Veja Misc/ACKS na distribuição do código do Python para ver uma lista parcial de contribuidores.

Tudo isso só foi possível com o esforço e a contribuição da comunidade Python, por isso temos essa maravilhosa documentação – Obrigado a todos!

APÊNDICE C

História e Licença

C.1 História do software

O Python foi criado no início dos anos 1990 por Guido van Rossum na Stichting Mathematisch Centrum (CWI, veja https://www.cwi.nl/) na Holanda como um sucessor de uma linguagem chamada ABC. Guido continua a ser o principal autor de Python, embora inclua muitas contribuições de outros.

Em 1995, Guido continuou seu trabalho em Python na Corporação para Iniciativas Nacionais de Pesquisa (CNRI, veja https://www.cnri.reston.va.us/) em Reston, Virgínia, onde lançou várias versões do software.

Em maio de 2000, Guido e a equipe principal de desenvolvimento do Python mudaram-se para o BeOpen.com para formar a equipe BeOpen PythonLabs. Em outubro do mesmo ano, a equipe da PythonLabs mudou para a Digital Creations (agora Zope Corporation; veja https://www.zope.org/). Em 2001, formou-se a Python Software Foundation (PSF, veja https://www.python.org/psf/), uma organização sem fins lucrativos criada especificamente para possuir propriedade intelectual relacionada a 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.

Versão	Derivada de	Ano	Proprietário	Compatível com a GPL?
0.9.0 a 1.2	n/a	1991-1995	CWI	sim
1.3 a 1.5.2	1.2	1995-1999	CNRI	sim
1.6	1.5.2	2000	CNRI	não
2.0	1.6	2000	BeOpen.com	não
1.6.1	1.6	2001	CNRI	não
2.1	2.0+1.6.1	2001	PSF	não
2.0.1	2.0+1.6.1	2001	PSF	sim
2.1.1	2.1+2.0.1	2001	PSF	sim
2.1.2	2.1.1	2002	PSF	sim
2.1.3	2.1.2	2002	PSF	sim
2.2 e acima	2.1.1	2001-agora	PSF	sim

Nota

> Compatível com a 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 a 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

O software e a documentação do Python são licenciados sob o *Acordo de Licenciamento PSF*.

A partir do Python 3.8.6, exemplos, receitas e outros códigos na documentação são licenciados duplamente sob o Acordo de Licenciamento PSF e a *Licença BSD de Zero Cláusula*.

Alguns softwares incorporados ao Python estão sob licenças diferentes. As licenças são listadas com o código abrangido por essa licença. Veja *Licenças e Reconhecimentos para Software Incorporado* para uma lista incompleta dessas licenças.

C.2.1 ACORDO DE LICENCIAMENTO DA PSF PARA PYTHON 3.12.7

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using Python
   3.12.7 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 3.12.7 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All Rights
   Reserved" are retained in Python 3.12.7 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 3.12.7 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
   3.12.7.

4. PSF is making Python 3.12.7 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 3.12.7 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.12.7
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.12.7, 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 3.12.7, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 ACORDO DE LICENCIAMENTO DA BEOPEN.COM PARA PYTHON 2.0

ACORDO DE LICENCIAMENTO DA BEOPEN DE FONTE ABERTA DO 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
   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.
```

(continua na próxima página)

(continuação da página anterior)

```
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
   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.2.5 LICENÇA BSD DE ZERO CLÁUSULA PARA CÓDIGO NA DOCUMENTAÇÃO DO PYTHON 3.12.7

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
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 reconhecimentos para softwares de terceiros incorporados na distribuição do Python.

C.3.1 Mersenne Twister

A extensão C `_random` subjacente ao módulo `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).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
```

(continua na próxima página)

(continuação da página anterior)

```
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 Soquetes

O módulo `socket` usa as funções `getaddrinfo()` e `getnameinfo()`, que são codificadas em arquivos de origem separados do Projeto WIDE, https://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.

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
```

(continua na próxima página)

(continuação da página anterior)

```
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Serviços de soquete assíncrono

Os módulos `test.support.asynchat` e `test.support.asyncore` contêm o seguinte aviso:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Gerenciamento de cookies

O módulo `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
```

(continua na próxima página)

(continuação da página anterior)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Rastreamento de execução

O módulo `trace` contém o seguinte aviso:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 Funções UUencode e UUdecode

The `uu` module contains the following notice:

```
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.
```

(continua na próxima página)

(continuação da página anterior)

```
Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 Chamadas de procedimento remoto XML

O módulo `xmlrpc.client` contém o seguinte aviso:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

O módulo `test.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.3.9 kqueue de seleção

O módulo `select` contém o seguinte aviso para a interface do kqueue:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

O arquivo `Python/pyhash.c` contém a implementação de Marek Majkowski do algoritmo SipHash24 de Dan Bernstein. Contém a seguinte nota:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/
```

(continua na próxima página)

(continuação da página anterior)

```
Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod e dtoa

O arquivo `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 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. O arquivo original, conforme recuperado em 16 de março de 2009, contém os seguintes avisos de direitos autorais e de licenciamento:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                              Apache License
                        Version 2.0, January 2004
                     https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
```

(continua na próxima página)

(continuação da página anterior)

```
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
```

(continua na próxima página)

(continuação da página anterior)

```
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
```

(continua na próxima página)

(continuação da página anterior)

```
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

A extensão `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
```

(continua na próxima página)

(continuação da página anterior)

```
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

A extensão C `_ctypes` subjacente ao módulo `ctypes` é construída usando uma cópia incluída das fontes do libffi, a menos que a construção esteja configurada com `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

A extensão `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.
```

(continua na próxima página)

(continuação da página anterior)

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

A implementação da tabela de hash usada pelo `tracemalloc` é baseada no projeto cfuhash:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

A extensão C `_decimal` subjacente ao módulo `decimal` é construída usando uma cópia incluída da biblioteca libmpdec, a menos que a construção esteja configurada com `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 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.
```

C.3.18 Conjunto de testes C14N do W3C

O conjunto de testes C14N 2.0 no pacote `test` (`Lib/test/xmltestdata/c14n-20/`) foi recuperado do site do W3C em https://www.w3.org/TR/xml-c14n2-testcases/ e é distribuído sob a licença BSD de 3 cláusulas:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original 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 W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work 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,
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.19 Audioop

The audioop module uses the code base in g771.c file of the SoX project. https://sourceforge.net/projects/sox/files/sox/12.17.7/sox-12.17.7.tar.gz

> This source code is a product of Sun Microsystems, Inc. and is provided for unrestricted use. Users may copy or modify this source code without charge.
>
> SUN SOURCE CODE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
>
> Sun source code is provided with no support and without any obligation on the part of Sun Microsystems, Inc. to assist in its use, correction, modification or enhancement.
>
> SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS SOFTWARE OR ANY PART THEREOF.
>
> In no event will Sun Microsystems, Inc. be liable for any lost revenue or profits or other special, indirect and consequential damages, even if Sun has been advised of the possibility of such damages.
>
> Sun Microsystems, Inc. 2550 Garcia Avenue Mountain View, California 94043

C.3.20 asyncio

Partes do módulo `asyncio` são incorporadas do uvloop 0.16, que é distribuído sob a licença MIT:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

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.
```

APÊNDICE D

Direitos autorais

Python e essa documentação é:

Copyright © 2001-2023 Python Software Foundation. Todos os direitos reservados.

Copyright © 2000 BeOpen.com. Todos os direitos reservados.

Copyright © 1995-2000 Corporation for National Research Initiatives. Todos os direitos reservados.

Copyright © 1991-1995 Stichting Mathematisch Centrum. Todos os direitos reservados.

Veja: *História e Licença* para informações completas de licença e permissões.

Índice

Não alfabético

...., 75
2to3, 75
>>>, 75
__future__, 81
__slots__, 88

A

aguardável, 77
ambiente virtual, 91
anotação, 75
anotação de função, 80
anotação de variável, 90
apelido de tipo, 90
API provisória, 87
argumento, 76
    difference from parameter, 12
argumento nomeado, 83
argumento posicional, 87
arquivo binário, 77
arquivo texto, 89
atributo, 76

B

BDFL, 77
bytecode, 77

C

caminho de importação, 82
carregador, 84
chamável, 77
classe, 78
classe base abstrata, 75
classe estilo novo, 86
codificação da localidade, 84
codificador de texto, 89
coleta de lixo, 81
compreensão de conjunto, 89
compreensão de dicionário, 79
compreensão de lista, 84
contagem de referências, 88
contíguo, 78
contíguo C, 78
contíguo Fortran, 78
corrotina, 78
CPython, 78

D

decorador, 78
descritor, 79
desligamento do interpretador, 83
despacho único, 89
dica de tipo, 90
dicionário, 79
divisão pelo piso, 80
docstring, 79

E

EAFP, 79
entrada de caminho, 87
escopo aninhado, 86
espaço de nomes, 85
especial
    método, 89
expressão, 79
expressão geradora, 81

F

f-string, 80
fatia, 89
função, 80
função chave, 83
função de corrotina, 78
função de retorno, 77
função genérica, 81

G

gancho de entrada de caminho, 87
gerador, 81
gerador assíncrono, 76
gerenciador de contexto, 78
gerenciador de contexto assíncrono, 76
GIL, 81

H

hasheável, 82

I

IDLE, 82
imortal, 82
importação, 82
importador, 82
imutável, 82
instrução, 89
interativo, 82
interpretado, 82
iterador, 83
iterador assíncrono, 76
iterador gerador, 81
iterador gerador assíncrono, 76
Iterável, 83
iterável assíncrono, 76

L

lambda, 83
LBYL, 84
lista, 84
localizador, 80
localizador baseado no caminho, 87
localizador de entrada de caminho, 87
localizador de metacaminho, 84

M

mágico
    método, 84
mapeamento, 84
máquina virtual, 91
metaclasse, 84
método, 85
    especial, 89
    mágico, 84
método especial, 89
método mágico, 84
módulo, 85
módulo de extensão, 79
módulo spec, 85
MRO, 85
mutável, 85

N

nome qualificado, 88
novas linhas universais, 90
número complexo, 78

O

objeto, 86
objeto arquivo, 80
objeto arquivo ou similar, 80
objeto byte ou similar, 77
objeto caminho ou similar, 87
ordem de resolução de métodos, 85

P

pacote, 86
pacote de espaço de nomes, 85
pacote provisório, 87
pacote regular, 88
parâmetro, 86
    difference from argument, 12
PATH, 52
PEP, 87
porção, 87
Propostas de Melhorias do Python
    PEP 1, 87
    PEP 5, 5
    PEP 8, 8, 33
    PEP 238, 80
    PEP 278, 90
    PEP 302, 84
    PEP 343, 78
    PEP 362, 76, 86
    PEP 373, 5
    PEP 387, 3
    PEP 411, 87
    PEP 420, 85, 87
    PEP 443, 81
    PEP 483, 81
    PEP 484, 75, 80, 81, 90, 91
    PEP 492, 7678
    PEP 498, 80
    PEP 519, 87
    PEP 525, 76
    PEP 526, 75, 91
    PEP 572, 41
    PEP 585, 81
    PEP 602, 4
    PEP 683, 82
    PEP 3116, 90
    PEP 3147, 36
    PEP 3155, 88
pyc baseado em hash, 82
Python 3000, 87
PYTHONDONTWRITEBYTECODE, 36
Pythônico, 88

R

referência emprestada, 77
referência forte, 89

S

sequência, 88
string entre aspas triplas, 90
suavemente descontinuado, 89

T

tipagem pato, 79
tipo, 90
tipo genérico, 81
tratador de erros e codificação do sistema de arquivos, 80
trava global do interpretador, 81
tupla nomeada, 85

V

variável de ambiente
    PATH, 52
    PYTHONDONTWRITEBYTECODE, 36
variável de classe, 78
variável de contexto, 78
verificador de tipo estático, 89
visão de dicionário, 79

Z

Zen do Python, 91