The Python Language Reference

Release 3.12.2

Guido van Rossum and the Python development team

fevereiro 07, 2024

Python Software Foundation
Email: docs@python.org

Sumário

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

CAPÍTULO 1

Introdução

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

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

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

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

1.1 Implementações Alternativas

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

As implementações conhecidas são:

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

Jython
: Versão do Python implementado em Java. Esta implementação pode ser usada como linguagem de Script em aplicações Java, ou pode ser usada para criar aplicativos usando as bibliotecas das classes do Java. Também

vem sendo bastante utilizado para criar testes unitários para as bibliotecas do Java. Mais informações podem ser encontradas no the Jython website.

Python for .NET
Essa implementação utiliza de fato a implementação CPython, mas é uma aplicação gerenciada .NET e disponibilizada como uma bibliotecas .NET. Foi desenvolvida por Brian Lloyd. Para obter mais informações, consulte o site do Python for .NET.

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

PyPy
Uma implementação do Python escrita completamente em Python. A mesma suporta vários recursos avançados não encontrados em outras implementações, como suporte sem pilhas e um compilador Just in Time. Um dos objetivos do projeto é incentivar a construção de experimentos com a própria linguagem, facilitando a modificação do interpretador (uma vez que o mesmos está escrito em Python). Informações adicionais estão disponíveis no site do projeto PyPy.

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

1.2 Notação

As descrições de análise léxica e sintaxe usam uma notação de gramática de Formalismo de Backus-Naur (BNF) modificada. Ela usa o seguinte estilo de definição:

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

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

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

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

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

CAPÍTULO 2

Análise léxica

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

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

2.1 Estrutura das linhas

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

2.1.1 Linhas lógicas

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

2.1.2 Linhas físicas

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

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

2.1.3 Comentários

Um comentário inicia com um caracter cerquilha (#) que não é parte de uma string literal, e termina com o fim da linha física. Um comentário significa o fim da linha lógica a menos que regras de junção de linha implicitas sejam invocadas. Comentários são ignorados pela sintaxe.

2.1.4 Declarações de codificação

Se um comentário na primeira ou segunda linha de um script Python corresponde com a expressão regular `coding[=:]\s*([-\w.]+)`, esse comentário é processado com uma declaração de codificação; o primeiro grupo dessa expressão indica a codificação do arquivo do código-fonte. A declaração de codificação deve aparecer em uma linha exclusiva para tal. Se está na segunda linha, a primeira linha também deve ser uma linha somente com comentário. As formas recomendadas de uma declaração de codificação são:

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

que é reconhecido também por GNU Emacs, e

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

que é reconhecido pelo VIM de Bram Moolenaar.

Se nenhuma declaração de codificação é encontrada, a codificação padrão é UTF-8. Adicionalmente, se os primeiros bytes do arquivo são a marca de ordem de byte (BOM) do UTF-8 (`b'\xef\xbb\xbf'`), a codificação de arquivo declarada é UTF-8 (isto é suportado, entre outros, pelo **notepad** da Microsoft).

Se uma codificação é declarada, o nome da codificação deve ser reconhecida pelo Python (veja standard-encodings). A codificação é usada por toda análise léxica, incluindo literais strings, comment and identificadores.

2.1.5 Junção de linha explícita

Duas ou mais linhas físicas podem ser juntadas em linhas lógicas usando o caractere contrabarra (\) da seguinte forma: quando uma linha física termina com uma contrabarra que não é parte da uma literal string ou comentário, ela é juntada com a linha seguinte formando uma única linha lógica, removendo a contrabarra e o caractere de fim de linha seguinte. Por exemplo:

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

Uma linha terminada em uma contrabarra não pode conter um comentário. Uma barra invertida não continua um comentário. Uma contrabarra não continua um token, exceto para strings literais (ou seja, tokens diferentes de strings literais não podem ser divididos em linhas físicas usando uma contrabarra). Uma contrabarra é ilegal em qualquer outro lugar em uma linha fora de uma string literal.

2.1.6 Junção de linha implícita

Expressões entre parênteses, colchetes ou chaves podem ser quebradas em mais de uma linha física sem a necessidade do uso de contrabarras. Por exemplo:

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

Linhas continuadas implicitamente podem conter comentários. O recuo das linhas de continuação não é importante. Linhas de continuação em branco são permitidas. Não há token NEWLINE entre linhas de continuação implícitas. Linhas continuadas implicitamente também podem ocorrer dentro de strings com aspas triplas (veja abaixo); nesse caso, eles não podem conter comentários.

2.1.7 Linhas em branco

Uma linha lógica que contém apenas espaços, tabulações, quebras de página e possivelmente um comentário é ignorada (ou seja, nenhum token NEWLINE é gerado). Durante a entrada interativa de instruções, o tratamento de uma linha em branco pode diferir dependendo da implementação do interpretador. No interpretador interativo padrão, uma linha lógica totalmente em branco (ou seja, uma que não contenha nem mesmo espaço em branco ou um comentário) encerra uma instrução de várias linhas.

2.1.8 Indentação

O espaço em branco (espaços e tabulações) no início de uma linha lógica é usado para calcular o nível de indentação da linha, que por sua vez é usado para determinar o agrupamento de instruções.

As tabulações são substituídas (da esquerda para a direita) por um a oito espaços, de modo que o número total de caracteres até e incluindo a substituição seja um múltiplo de oito (essa é intencionalmente a mesma regra usada pelo Unix). O número total de espaços que precedem o primeiro caractere não em branco determina o recuo da linha. O recuo não pode ser dividido em várias linhas físicas usando contrabarra; o espaço em branco até a primeira contrabarra determina a indentação.

A indentação é rejeitada como inconsistente se um arquivo de origem mistura tabulações e espaços de uma forma que torna o significado dependente do valor de uma tabulação em espaços; uma exceção `TabError` é levantada nesse caso.

Nota de compatibilidade entre plataformas: devido à natureza dos editores de texto em plataformas não-UNIX, não é aconselhável usar uma mistura de espaços e tabulações para o recuo em um único arquivo de origem. Deve-se notar também que diferentes plataformas podem limitar explicitamente o nível máximo de indentação.

Um caractere de quebra de página pode estar presente no início da linha; ele será ignorado para os cálculos de indentação acima. Os caracteres de quebra de página que ocorrem em outro lugar além do espaço em branco inicial têm um efeito indefinido (por exemplo, eles podem redefinir a contagem de espaços para zero).

Os níveis de indentação das linhas consecutivas são usados para gerar tokens INDENT e DEDENT, usando uma pilha, como segue.

Antes da leitura da primeira linha do arquivo, um único zero é colocado na pilha; isso nunca mais será exibido. Os números colocados na pilha sempre aumentarão estritamente de baixo para cima. No início de cada linha lógica, o nível de indentação da linha é comparado ao topo da pilha. Se for igual, nada acontece. Se for maior, ele é colocado na pilha e um token INDENT é gerado. Se for menor, *deve* ser um dos números que aparecem na pilha; todos os números maiores na pilha são retirados e, para cada número retirado, um token DEDENT é gerado. Ao final do arquivo, um token DEDENT é gerado para cada número restante na pilha que seja maior que zero.

Aqui está um exemplo de um trecho de código Python indentado corretamente (embora confuso):

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

O exemplo a seguir mostra vários erros de indentação:

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

(Na verdade, os três primeiros erros são detectados pelo analisador sintático; apenas o último erro é encontrado pelo analisador léxico — o recuo de não corresponde a um nível retirado da pilha.)

2.1.9 Espaços em branco entre tokens

Exceto no início de uma linha lógica ou em string literais, os caracteres de espaço em branco (espaço, tabulação e quebra de página) podem ser usados alternadamente para separar tokens. O espaço em branco é necessário entre dois tokens somente se sua concatenação puder ser interpretada como um token diferente (por exemplo, ab é um token, mas a b são dois tokens).

2.2 Outros tokens

Além de NEWLINE, INDENT e DEDENT, existem as seguintes categorias de tokens: *identificadores*, *palavras-chave*, *literais*, *operadores* e *delimitadores*. Caracteres de espaço em branco (exceto terminadores de linha, discutidos anteriormente) não são tokens, mas servem para delimitar tokens. Onde existe ambiguidade, um token compreende a string mais longa possível que forma um token legal, quando lido da esquerda para a direita.

2.3 Identificadores e palavras-chave

Identificadores (também chamados de *nomes*) são descritos pelas seguintes definições lexicais.

A sintaxe dos identificadores em Python é baseada no anexo do padrão Unicode UAX-31, com elaboração e alterações conforme definido abaixo; veja também **PEP 3131** para mais detalhes.

Dentro do intervalo ASCII (U+0001..U+007F), os caracteres válidos para identificadores são os mesmos de Python 2.x: as letras maiúsculas e minúsculas de `A` até `Z`, o sublinhado _ e, exceto para o primeiro caractere, os dígitos `0` até `9`.

Python 3.0 introduz caracteres adicionais fora do intervalo ASCII (consulte **PEP 3131**). Para esses caracteres, a classificação utiliza a versão do Banco de Dados de Caracteres Unicode incluída no módulo `unicodedata`.

Os identificadores têm comprimento ilimitado. Maiúsculas são diferentes de minúsculas.

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

Os códigos de categoria Unicode mencionados acima significam:

- *Lu* - letras maiúsculas
- *Ll* - letras minúsculas
- *Lt* - letras em titlecase
- *Lm* - letras modificadoras

- *Lo* - outras letras
- *Nl* - letras numéricas
- *Mn* - marcas sem espaçamento
- *Mc* - marcas de combinação de espaçamento
- *Nd* - números decimais
- *Pc* - pontuações de conectores
- *Other_ID_Start* - lista explícita de caracteres em PropList.txt para oferecer suporte à compatibilidade com versões anteriores
- *Other_ID_Continue* - igualmente

Todos os identificadores são convertidos no formato normal NFKC durante a análise; a comparação de identificadores é baseada no NFKC.

Um arquivo HTML não normativo listando todos os caracteres identificadores válidos para Unicode 15.0.0 pode ser encontrado em https://www.unicode.org/Public/15.0.0/ucd/DerivedCoreProperties.txt

2.3.1 Palavras reservadas

Os seguintes identificadores são usados como palavras reservadas, ou *palavras-chave* da linguagem, e não podem ser usados como identificadores comuns. Eles devem ser escritos exatamente como estão escritos aqui:

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

2.3.2 Palavras reservadas contextuais

Novo na versão 3.10.

Alguns identificadores são reservados apenas em contextos específicos. Elas são conhecidas como *palavras reservadas contextuais*. Os identificadores `match`, `case`, `type` e _ podem atuar sintaticamente como palavras reservadas em determinados contextos, mas essa distinção é feita no nível do analisador sintático, não durante a tokenização.

Como palavras reservadas contextuais, seu uso na gramática é possível preservando a compatibilidade com o código existente que usa esses nomes como identificadores.

`match`, `case` e _ são usadas na instrução *`match`*, `type` é usado na instrução *`type`*.

Alterado na versão 3.12: `type` é agora uma palavra reservada contextual.

2.3.3 Classes reservadas de identificadores

Certas classes de identificadores (além de palavras reservadas) possuem significados especiais. Essas classes são identificadas pelos padrões de caracteres de sublinhado iniciais e finais:

_*

Não importado por `from module import *`.

_

Em um padrão `case` de uma instrução *`match`*, _ é uma *palavra reservada contextual* que denota um *curinga*.

Isoladamente, o interpretador interativo disponibiliza o resultado da última avaliação na variável _. (Ele é armazenado no módulo `builtins`, juntamente com funções embutidas como `print`.)

Em outros lugares, _ é um identificador comum. Muitas vezes é usado para nomear itens "especiais", mas não é especial para o Python em si.

Nota: O nome _ é frequentemente usado em conjunto com internacionalização; consulte a documentação do módulo `gettext` para obter mais informações sobre esta convenção.

Também é comumente usado para variáveis não utilizadas.

__*__
Nomes definidos pelo sistema, informalmente conhecidos como nomes "dunder". Esses nomes e suas implementações são definidos pelo interpretador (incluindo a biblioteca padrão). Os nomes de sistema atuais são discutidos na seção *Nomes de métodos especiais* e em outros lugares. Provavelmente mais nomes serão definidos em versões futuras do Python. *Qualquer* uso de nomes __*__, em qualquer contexto, que não siga o uso explicitamente documentado, está sujeito a quebra sem aviso prévio.

__*
Nomes de classes privadas. Os nomes nesta categoria, quando usados no contexto de uma definição de classe, são reescritos para usar uma forma desfigurada para ajudar a evitar conflitos de nomes entre atributos "privados" de classes base e derivadas. Consulte a seção *Identificadores (Nomes)*.

2.4 Literais

Literais são notações para valores constantes de alguns tipos embutidos.

2.4.1 Literais de string e bytes

Literais de string são descritos pelas seguintes definições lexicais:

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

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

Uma restrição sintática não indicada por essas produções é que não são permitidos espaços em branco entre o *stringprefix* ou *bytesprefix* e o restante do literal. O conjunto de caracteres de origem é definido pela declaração de codificação; é UTF-8 se nenhuma declaração de codificação for fornecida no arquivo de origem; veja a seção *Declarações de codificação*.

Em inglês simples: ambos os tipos de literais podem ser colocados entre aspas simples (') ou aspas duplas ("). Eles também podem ser colocados em grupos correspondentes de três aspas simples ou duplas (geralmente chamadas de *strings com aspas triplas*). O caractere de contrabarra (\) é usado para dar um significado especial a caracteres comuns como , que significa 'nova linha' quando escapado (\n). Também pode ser usado para caracteres de escape que, de outra forma, teriam um significado especial, como nova linha, contrabarra ou o caractere de aspas. Veja *sequências de escape* abaixo para exemplos.

Literais de bytes são sempre prefixados com 'b' ou 'B'; eles produzem uma instância do tipo bytes em vez do tipo str. Eles só podem conter caracteres ASCII; bytes com valor numérico igual ou superior a 128 devem ser expressos com escapes.

Literais de string e bytes podem opcionalmente ser prefixados com uma letra 'r' ou 'R'; essas strings são chamadas de strings brutas e tratam as barras invertidas como caracteres literais. Como resultado, em literais de string, os escapes '\U' e '\u' em strings brutas não são tratados de maneira especial. Dado que os literais unicode brutos de Python 2.x se comportam de maneira diferente dos de Python 3.x, não há suporte para a sintaxe 'ur' .

Novo na versão 3.3: O prefixo 'rb' de literais de bytes brutos foi adicionado como sinônimo de 'br'.

Novo na versão 3.3: O suporte para o literal legado unicode (u'value') foi reintroduzido para simplificar a manutenção de bases de código duplas Python 2.x e 3.x. Consulte **PEP 414** para obter mais informações.

Uma string literal com 'f' ou 'F' em seu prefixo é uma string literal formatada; veja *f-strings*. O 'f' pode ser combinado com 'r', mas não com 'b' ou 'u', portanto strings formatadas brutas são possíveis, mas literais de bytes formatados não são.

Em literais com aspas triplas, novas linhas e aspas sem escape são permitidas (e são retidas), exceto que três aspas sem escape em uma linha encerram o literal. (Uma "aspas" é o caractere usado para abrir o literal, ou seja, ' ou ".)

Sequências de escape

A menos que um prefixo 'r' ou 'R' esteja presente, as sequências de escape em literais de string e bytes são interpretadas de acordo com regras semelhantes àquelas usadas pelo Standard C. As sequências de escape reconhecidas são:

Sequência de escape	Significado	Notas
\<newline>	A barra invertida e a nova linha foram ignoradas	(1)
\\	Contrabarra (\)	
\'	Aspas simples (')	
\"	Aspas duplas (")	
\a	ASCII Bell (BEL) - um sinal audível é emitido	
\b	ASCII Backspace (BS) - apaga caractere à esquerda	
\f	ASCII Formfeed (FF) - quebra de página	
\n	ASCII Linefeed (LF) - quebra de linha	
\r	ASCII Carriage Return (CR) - retorno de carro	
\t	ASCII Horizontal Tab (TAB) - tabulação horizontal	
\v	ASCII Vertical Tab (VT) - tabulação vertical	
ooo	Caractere com valor octal *ooo*	(2,4)
\x*hh*	Caractere com valor hexadecimal *hh*	(3,4)

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

Sequência de escape	Significado	Notas
\N{*name*}	Caractere chamado *name* no banco de dados Unicode	(5)
\u*xxxx*	Caractere com valor hexadecimal de 16 bits *xxxx*	(6)
\U*xxxxxxxx*	Caractere com valor hexadecimal de 32 bits *xxxxxxxx*	(7)

Notas:

(1) Uma contrabarra pode ser adicionada ao fim da linha para ignorar a nova linha:

```
>>> 'This string will not include \
... backslashes or newline characters.'
'This string will not include backslashes or newline characters.'
```

O mesmo resultado pode ser obtido usando *strings com aspas triplas*, ou parênteses e *concatenação de literal string*.

(2) Como no padrão C, são aceitos até três dígitos octais.

Alterado na versão 3.11: Escapes octais com valor maior que `0o377` produz uma `DeprecationWarning`.

Alterado na versão 3.12: Escapes octais com valor maior que `0o377` produzem um `SyntaxWarning`. Em uma versão futura do Python eles serão eventualmente um `SyntaxError`.

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

(4) Em um literal de bytes, os escapes hexadecimais e octais denotam o byte com o valor fornecido. Em uma literal de string, esses escapes denotam um caractere Unicode com o valor fornecido.

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

(6) São necessários exatos quatro dígitos hexadecimais.

(7) Qualquer caractere Unicode pode ser codificado desta forma. São necessários exatamente oito dígitos hexadecimais.

Ao contrário do padrão C, todas as sequências de escape não reconhecidas são deixadas inalteradas na string, ou seja, *a contrabarra é deixada no resultado*. (Esse comportamento é útil durante a depuração: se uma sequência de escape for digitada incorretamente, a saída resultante será mais facilmente reconhecida como quebrada.) Também é importante observar que as sequências de escape reconhecidas apenas em literais de string se enquadram na categoria de escapes não reconhecidos para literais de bytes.

Alterado na versão 3.6: Sequências de escape não reconhecidas produzem um `DeprecationWarning`.

Alterado na versão 3.12: Sequências de escape não reconhecidas produzem um `SyntaxWarning`. Em uma versão futura do Python eles serão eventualmente um `SyntaxError`.

Mesmo em um literal bruto, as aspas podem ser escapadas com uma contrabarra, mas a barra invertida permanece no resultado; por exemplo, `r"\""` é uma literal de string válida que consiste em dois caracteres: uma contrabarra e aspas duplas; `r"\"` não é uma literal de string válida (mesmo uma string bruta não pode terminar em um número ímpar de contrabarras). Especificamente, *um literal bruto não pode terminar em uma única contrabarra* (já que a contrabarra escaparia do seguinte caractere de aspas). Observe também que uma única contrabarra seguida por uma nova linha é interpretada como esses dois caracteres como parte do literal, *não* como uma continuação de linha.

2.4.2 Concatenação de literal de string

São permitidos vários literais de strings ou bytes adjacentes (delimitados por espaços em branco), possivelmente usando diferentes convenções de delimitação de strings, e seu significado é o mesmo de sua concatenação. Assim, `"hello" 'world'` é equivalente a `"helloworld"`. Este recurso pode ser usado para reduzir o número de barras invertidas necessárias, para dividir strings longas convenientemente em linhas longas ou até mesmo para adicionar comentários a partes de strings, por exemplo:

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

Observe que esse recurso é definido no nível sintático, mas implementado em tempo de compilação. O operador '+' deve ser usado para concatenar expressões de string em tempo de execução. Observe também que a concatenação literal pode usar diferentes estilos de delimitação de strings para cada componente (mesmo misturando strings brutas e strings com aspas triplas), e literais de string formatados podem ser concatenados com literais de string simples.

[1] https://www.unicode.org/Public/15.0.0/ucd/NameAliases.txt

2.4.3 f-strings

Novo na versão 3.6.

Um *literal de string formatado* ou *f-string* é uma literal de string prefixado com `'f'` ou `'F'`. Essas strings podem conter campos de substituição, que são expressões delimitadas por chaves `{}`. Embora outros literais de string sempre tenham um valor constante, strings formatadas são, na verdade, expressões avaliadas em tempo de execução.

As sequências de escape são decodificadas como em literais de string comuns (exceto quando um literal também é marcado como uma string bruta). Após a decodificação, a gramática do conteúdo da string é:

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

As partes da string fora das chaves são tratadas literalmente, exceto que quaisquer chaves duplas `'{{'` ou `'}}'` são substituídas pela chave única correspondente. Uma única chave de abertura `'{'` marca um campo de substituição, que começa com uma expressão Python. Para exibir o texto da expressão e seu valor após a avaliação (útil na depuração), um sinal de igual `'='` pode ser adicionado após a expressão. Um campo de conversão, introduzido por um ponto de exclamação `'!'`, pode vir a seguir. Um especificador de formato também pode ser anexado, introduzido por dois pontos `':'`. Um campo de substituição termina com uma chave de fechamento `'}'`.

Expressões em literais de string formatadas são tratadas como expressões regulares do Python entre parênteses, com algumas exceções. Uma expressão vazia não é permitida e as expressões *lambda* e de atribuição `:=` devem ser colocadas entre parênteses explícitos. Cada expressão é avaliada no contexto onde o literal de string formatado aparece, na ordem da esquerda para a direita. As expressões de substituição podem conter novas linhas em strings formatadas entre aspas simples e triplas e podem conter comentários. Tudo o que vem depois de um `#` dentro de um campo de substituição é um comentário (até mesmo colchetes e aspas). Nesse caso, os campos de substituição deverão ser fechados em uma linha diferente.

```
>>> f"abc{a # This is a comment }"
... + 3}"
'abc5'
```

Alterado na versão 3.7: Antes do Python 3.7, uma expressão *await* e compreensões contendo uma cláusula *async for* eram ilegais nas expressões em literais de string formatados devido a um problema com a implementação.

Alterado na versão 3.12: Antes do Python 3.12, comentários não eram permitidos dentro de campos de substituição em f-strings.

Quando o sinal de igual `'='` for fornecido, a saída terá o texto da expressão, o `'='` e o valor avaliado. Os espaços após a chave de abertura `'{'`, dentro da expressão e após `'='` são todos preservados na saída. Por padrão, `'='` faz com que `repr()` da expressão seja fornecida, a menos que haja um formato especificado. Quando um formato é especificado, o padrão é o `str()` da expressão, a menos que uma conversão `'!r'` seja declarada.

Novo na versão 3.8: O sinal de igual `'='`.

Se uma conversão for especificada, o resultado da avaliação da expressão será convertido antes da formatação. A conversão `'!s'` chama `str()` no resultado, `'!r'` chama `repr()` e `'!a'` chama `ascii()`.

O resultado é então formatado usando o protocolo `format()`. O especificador de formato é passado para o método *__format__()* da expressão ou resultado da conversão. Uma string vazia é passada quando o especificador de formato é omitido. O resultado formatado é então incluído no valor final de toda a string.

Os especificadores de formato de nível superior podem incluir campos de substituição aninhados. Esses campos aninhados podem incluir seus próprios campos de conversão e especificadores de formato, mas podem não incluir

campos de substituição aninhados mais profundamente. A minilinguagem do especificador de formato é a mesma usada pelo método `str.format()`.

Literais de string formatados podem ser concatenados, mas os campos de substituição não podem ser divididos entre literais.

Alguns exemplos de literais de string formatados:

```
>>> name = "Fred"
>>> f"He said his name is {name!r}."
"He said his name is 'Fred'."
>>> f"He said his name is {repr(name)}."  # repr() is equivalent to !r
"He said his name is 'Fred'."
>>> width = 10
>>> precision = 4
>>> value = decimal.Decimal("12.34567")
>>> f"result: {value:{width}.{precision}}"  # nested fields
'result:      12.35'
>>> today = datetime(year=2017, month=1, day=27)
>>> f"{today:%B %d, %Y}"  # using date format specifier
'January 27, 2017'
>>> f"{today=:%B %d, %Y}" # using date format specifier and debugging
'today=January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
>>> foo = "bar"
>>> f"{ foo = }" # preserves whitespace
" foo = 'bar'"
>>> line = "The mill's closed"
>>> f"{line = }"
'line = "The mill\'s closed"'
>>> f"{line = :20}"
"line = The mill's closed   "
>>> f"{line = !r:20}"
'line = "The mill\'s closed" '
```

É permitido reutilizar o tipo de aspas de f-string externa dentro de um campo de substituição:

```
>>> a = dict(x=2)
>>> f"abc {a["x"]} def"
'abc 2 def'
```

Alterado na versão 3.12: Antes do Python 3.12, a reutilização do mesmo tipo de aspas da f-string externa dentro de um campo de substituição não era possível.

Contrabarras também são permitidas em campos de substituição e são avaliadas da mesma forma que em qualquer outro contexto:

```
>>> a = ["a", "b", "c"]
>>> print(f"List a contains:\n{"\n".join(a)}")
List a contains:
a
b
c
```

Alterado na versão 3.12: Antes do Python 3.12, contrabarras não eram permitidas dentro de um campo de substituição em uma f-string.

Literais de string formatados não podem ser usados como strings de documentação, mesmo que não incluam expressões.

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

Consulte também **PEP 498** para a proposta que adicionou literais de string formatados e `str.format()`, que usa um mecanismo de string de formato relacionado.

2.4.4 Literais numéricos

Existem três tipos de literais numéricos: inteiros, números de ponto flutuante e números imaginários. Não existem literais complexos (números complexos podem ser formados adicionando um número real e um número imaginário).

Observe que os literais numéricos não incluem um sinal; uma frase como `-1` é, na verdade, uma expressão composta pelo operador unário '`-`' e o literal `1`.

2.4.5 Inteiros literais

Literais inteiros são descritos pelas seguintes definições léxicas:

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

Não há limite para o comprimento de literais inteiros além do que pode ser armazenado na memória disponível.

Os sublinhados são ignorados para determinar o valor numérico do literal. Eles podem ser usados para agrupar dígitos para maior legibilidade. Um sublinhado pode ocorrer entre dígitos e após especificadores de base como `0x`.

Observe que não são permitidos zeros à esquerda em um número decimal diferente de zero. Isto é para desambiguação com literais octais de estilo C, que o Python usava antes da versão 3.0.

Alguns exemplos de literais inteiros:

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

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

2.4.6 Literais de ponto flutuante

Literais de ponto flutuante são descritos pelas seguintes definições léxicas:

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

Observe que as partes inteiras e expoentes são sempre interpretadas usando base 10. Por exemplo, `077e010` é válido e representa o mesmo número que `77e10`. O intervalo permitido de literais de ponto flutuante depende da implementação. Assim como em literais inteiros, os sublinhados são permitidos para agrupamento de dígitos.

Alguns exemplos de literais de ponto flutuante:

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

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

2.4.7 Literais imaginários

Os literais imaginários são descritos pelas seguintes definições léxicas:

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

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

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

2.5 Operadores

Os seguintes tokens são operadores:

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

2.6 Delimitadores

Os seguintes tokens servem como delimitadores na gramática:

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

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

Os seguintes caracteres ASCII imprimíveis têm um significado especial como parte de outros tokens ou são significativos para o analisador léxico:

```
'       "       #       \
```

Os seguintes caracteres ASCII imprimíveis não são usados em Python. Sua ocorrência fora de literais de string e comentários é um erro incondicional:

```
$       ?       `
```

CAPÍTULO 3

Modelo de dados

3.1 Objetos, valores e tipos

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

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

Detalhes da implementação do CPython: Para CPython, `id(x)` é o endereço de memória em que `x` está armazenado.

O tipo de um objeto determina as operações que o objeto implementa (por exemplo, "ele tem um tamanho?") e também define os valores possíveis para objetos desse tipo. A função `type()` retorna o tipo de um objeto (que é também um objeto). Como sua identidade, o *tipo* do objeto também é imutável.[1]

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

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

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

[1] Em alguns casos, *é* possível alterar o tipo de um objeto, sob certas condições controladas. No entanto, geralmente não é uma boa ideia, pois pode levar a um comportamento muito estranho se for tratado incorretamente.

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

Some objects contain references to "external" resources such as open files or windows. It is understood that these resources are freed when the object is garbage-collected, but since garbage collection is not guaranteed to happen, such objects also provide an explicit way to release the external resource, usually a `close()` method. Programs are strongly recommended to explicitly close such objects. The '*try*...*finally*' statement and the '*with*' statement provide convenient ways to do this.

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

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

3.2 A hierarquia de tipos padrão

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

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

3.2.1 None

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

3.2.2 NotImplemented

Este tipo possui um único valor. Existe um único objeto com este valor. Este objeto é acessado através do nome embutido `NotImplemented`. Os métodos numéricos e métodos de comparação rica devem retornar esse valor se não implementarem a operação para os operandos fornecidos. (O interpretador tentará então a operação refletida ou alguma outra alternativa, dependendo do operador.) Não deve ser avaliado em um contexto booleano.

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

Alterado na versão 3.9: A avaliação de `NotImplemented` em um contexto booleano foi descontinuado. Embora atualmente seja avaliado como verdadeiro, ele emitirá um `DeprecationWarning`. Ele levantará uma `TypeError` em uma versão futura do Python.

3.2.3 Ellipsis

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

3.2.4 `numbers.Number`

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

As representações de string das classes numéricas, calculadas por *`__repr__()`* e *`__str__()`*, têm as seguintes propriedades:

- Elas são literais numéricos válidos que, quando passados para seu construtor de classe, produzem um objeto com o valor do numérico original.
- A representação está na base 10, quando possível.
- Os zeros à esquerda, possivelmente com exceção de um único zero antes de um ponto decimal, não são mostrados.
- Os zeros à direita, possivelmente com exceção de um único zero após um ponto decimal, não são mostrados.
- Um sinal é mostrado apenas quando o número é negativo.

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

`numbers.Integral`

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

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

Existem dois tipos de inteiros:

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

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

numbers.Real (float)

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

numbers.Complex (complex)

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

3.2.5 Sequências

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

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

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

As sequências são distinguidas de acordo com sua mutabilidade:

Sequências imutáveis

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

Os tipos a seguir são sequências imutáveis:

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

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

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

Sequências mutáveis

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

Nota: Os módulos `collections` e `array` fornecem exemplos adicionais de tipos de sequência mutáveis.

Atualmente, existem dois tipos de sequência mutável intrínseca:

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

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

3.2.6 Tipos de conjuntos

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

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

Atualmente, existem dois tipos de conjuntos intrínsecos:

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

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

3.2.7 Mapeamentos

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

Atualmente, há um único tipo de mapeamento intrínseco:

Dicionários

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

Dicionários preservam a ordem de inserção, o que significa que as chaves serão produzidas na mesma ordem em que foram adicionadas sequencialmente no dicionário. Substituir uma chave existente não altera a ordem, no entanto, remover uma chave e inseri-la novamente irá adicioná-la ao final em vez de manter seu lugar anterior.

Os dicionários são mutáveis; eles podem ser criados pela notação `{...}` (veja a seção *Sintaxes de criação de dicionário*).

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

Alterado na versão 3.7: Dicionários não preservavam a ordem de inserção nas versões do Python anteriores à 3.6. No CPython 3.6, a ordem de inserção foi preservada, mas foi considerada um detalhe de implementação naquela época, em vez de uma garantia da linguagem.

3.2.8 Tipos chamáveis

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

Funções definidas pelo usuário

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

Special read-only attributes

Atributo	Significado
`function.__globals__`	A reference to the `dictionary` that holds the function's *global variables* – the global namespace of the module in which the function was defined.
`function.__closure__`	`None` or a `tuple` of cells that contain bindings for the function's free variables. Um objeto de célula tem o atributo `cell_contents`. Isso pode ser usado para obter o valor da célula, bem como definir o valor.

Special writable attributes

Most of these attributes check the type of the assigned value:

Atributo	Significado
`function.__doc__`	The function's documentation string, or `None` if unavailable. Not inherited by subclasses.
`function.__name__`	The function's name. See also: `__name__` `attributes`.
`function.__qualname__`	The function's *qualified name*. See also: `__qualname__` `attributes`. Novo na versão 3.3.
`function.__module__`	O nome do módulo em que a função foi definida ou `None` se indisponível.
`function.__defaults__`	A `tuple` containing default *parameter* values for those parameters that have defaults, or `None` if no parameters have a default value.
`function.__code__`	The *code object* representing the compiled function body.
`function.__dict__`	The namespace supporting arbitrary function attributes. See also: `__dict__` `attributes`.
`function.__annotations__`	A `dictionary` containing annotations of *parameters*. The keys of the dictionary are the parameter names, and `'return'` for the return annotation, if provided. See also: annotations-howto.
`function.__kwdefaults__`	A `dictionary` containing defaults for keyword-only *parameters*.
`function.__type_params__`	A `tuple` containing the *type parameters* of a *generic function*. Novo na versão 3.12.

Function objects also support getting and setting arbitrary attributes, which can be used, for example, to attach metadata to functions. Regular attribute dot-notation is used to get and set such attributes.

Detalhes da implementação do CPython: CPython's current implementation only supports function attributes on user-defined functions. Function attributes on *built-in functions* may be supported in the future.

Additional information about a function's definition can be retrieved from its *code object* (accessible via the *__code__* attribute).

Métodos de instância

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

Special read-only attributes:

method.**__self__**	Refers to the class instance object to which the method is *bound*
method.**__func__**	Refers to the original *function object*
method.**__doc__**	The method's documentation (same as *method.__func__.__doc__*). A `string` if the original function had a docstring, else `None`.
method.**__name__**	The name of the method (same as *method.__func__.__name__*)
method.**__module__**	The name of the module the method was defined in, or `None` if unavailable.

Methods also support accessing (but not setting) the arbitrary function attributes on the underlying *function object*.

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined *function object* or a `classmethod` object.

When an instance method object is created by retrieving a user-defined *function object* from a class via one of its instances, its *__self__* attribute is the instance, and the method object is said to be *bound*. The new method's *__func__* attribute is the original function object.

When an instance method object is created by retrieving a `classmethod` object from a class or instance, its *__self__* attribute is the class itself, and its *__func__* attribute is the function object underlying the class method.

When an instance method object is called, the underlying function (*__func__*) is called, inserting the class instance (*__self__*) in front of the argument list. For instance, when `C` is a class which contains a definition for a function `f()`, and `x` is an instance of `C`, calling `x.f(1)` is equivalent to calling `C.f(x, 1)`.

When an instance method object is derived from a `classmethod` object, the "class instance" stored in *__self__* will actually be the class itself, so that calling either `x.f(1)` or `C.f(1)` is equivalent to calling `f(C,1)` where `f` is the underlying function.

Note that the transformation from *function object* to instance method object happens each time the attribute is retrieved from the instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local variable. Also notice that this transformation only happens for user-defined functions; other callable objects (and all non-callable objects) are retrieved without transformation. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this *only* happens when the function is an attribute of the class.

Funções geradoras

Uma função ou método que usa a instrução *yield* (veja a seção *A instrução yield*) é chamada de *função geradora*. Tal função, quando chamada, sempre retorna um objeto *iterator* que pode ser usado para executar o corpo da função: chamar o método `iterator.__next__()` do iterador fará com que a função seja executada até que forneça um valor usando a instrução `yield`. Quando a função executa uma instrução *return* ou sai do fim, uma exceção `StopIteration` é levantada e o iterador terá alcançado o fim do conjunto de valores a serem retornados.

Funções de corrotina

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

Funções geradoras assíncronas

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

Chamar o método *aiterator.__anext__* do iterador assíncrono retornará um *aguardável* que, quando aguardado, será executado até fornecer um valor usando a expressão *yield*. Quando a função executa uma instrução vazia *return* ou chega ao final, uma exceção `StopAsyncIteration` é levantada e o iterador assíncrono terá alcançado o final do conjunto de valores a serem produzidos.

Funções embutidas

A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (`math` is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes:

- `__doc__` is the function's documentation string, or `None` if unavailable. See *function.__doc__*.
- `__name__` is the function's name. See *function.__name__*.
- `__self__` is set to `None` (but see the next item).
- `__module__` is the name of the module the function was defined in or `None` if unavailable. See *function.__module__*.

Métodos embutidos

This is really a different disguise of a built-in function, this time containing an object passed to the C function as an implicit extra argument. An example of a built-in method is `alist.append()`, assuming *alist* is a list object. In this case, the special read-only attribute `__self__` is set to the object denoted by *alist*. (The attribute has the same semantics as it does with *other instance methods*.)

Classes

Classes are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override *__new__()*. The arguments of the call are passed to `__new__()` and, in the typical case, to *__init__()* to initialize the new instance.

Instâncias de classes

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

3.2.9 Módulos

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

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

Atributos predefinidos graváveis:

`__name__`
O nome do módulo.

`__doc__`
A string de documentação do módulo, ou `None` se indisponível.

`__file__`
O endereço do caminho do arquivo que o módulo foi carregado, se ele foi carregado a partir de um arquivo. O atributo `__file__` pode estar ausente para certos tipos de módulos, como os módulos C que são estaticamente vinculados ao interpretador. Para extensões de módulos carregadas dinamicamente de uma biblioteca compartilhada, é o endereço do caminho do arquivo da biblioteca compartilhada.

`__annotations__`
Um dicionário contendo *anotações de variável* coletadas durante a execução do corpo do módulo. Para as melhores práticas sobre como trabalhar com `__annotations__`, por favor veja annotations-howto.

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

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

3.2.10 Classes personalizadas

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

When a class attribute reference (for class `C`, say) would yield a class method object, it is transformed into an instance method object whose `__self__` attribute is `C`. When it would yield a `staticmethod` object, it is transformed into the object wrapped by the static method object. See section *Implementando descritores* for another way in which attributes retrieved from a class may differ from those actually contained in its `__dict__`.

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

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

Atributos especiais:

`__name__`
O nome da classe.

__module__
O nome do módulo no qual a classe foi definida.

__dict__
O dicionário contendo o espaço de nomes da classe.

__bases__
Uma tupla contendo a classe base, na ordem de suas ocorrências na lista da classe base.

__doc__
A string de documentação da classe, ou `None` se não definida.

__annotations__
Um dicionário contendo *anotações de variável* coletadas durante a execução do corpo da classe. Para melhores práticas sobre como trabalhar com `__annotations__`, por favor veja annotations-howto.

__type_params__
Uma tupla contendo os *parâmetos de tipos* de uma *classe genérica*.

3.2.11 Instâncias de classe

A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object, it is transformed into an instance method object whose *__self__* attribute is the instance. Static method and class method objects are also transformed; see above under "Classes". See section *Implementando descritores* for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's `__dict__`. If no class attribute is found, and the object's class has a *__getattr__()* method, that is called to satisfy the lookup.

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

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

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

3.2.12 Objetos de E/S (também conhecidos como objetos arquivo)

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

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

3.2.13 Tipos internos

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

Objetos código

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

Special read-only attributes

codeobject.**co_name**	The function name
codeobject.**co_qualname**	The fully qualified function name
codeobject.**co_argcount**	The total number of positional *parameters* (including positional-only parameters and parameters with default values) that the function has
codeobject.**co_posonlyargcount**	The number of positional-only *parameters* (including arguments with default values) that the function has
codeobject.**co_kwonlyargcount**	The number of keyword-only *parameters* (including arguments with default values) that the function has
codeobject.**co_nlocals**	The number of *local variables* used by the function (including parameters)
codeobject.**co_varnames**	A `tuple` containing the names of the local variables in the function (starting with the parameter names)
codeobject.**co_cellvars**	A `tuple` containing the names of *local variables* that are referenced by nested functions inside the function
codeobject.**co_freevars**	A `tuple` containing the names of free variables in the function
codeobject.**co_code**	A string representing the sequence of *bytecode* instructions in the function
codeobject.**co_consts**	A `tuple` containing the literals used by the *bytecode* in the function
codeobject.**co_names**	A `tuple` containing the names used by the *bytecode* in the function
codeobject.**co_filename**	The name of the file from which the code was compiled
codeobject.**co_firstlineno**	The line number of the first line of the function
codeobject.**co_lnotab**	A string encoding the mapping from *bytecode* offsets to line numbers. For details, see the source code of the interpreter. Obsoleto desde a versão 3.12: This attribute of code objects is deprecated, and may be removed in Python 3.14.
codeobject.**co_stacksize**	The required stack size of the code object
codeobject.**co_flags**	An `integer` encoding a number of flags for the interpreter.

The following flag bits are defined for *co_flags*: bit `0x04` is set if the function uses the `*arguments` syntax to accept an arbitrary number of positional arguments; bit `0x08` is set if the function uses the `**keywords` syntax to accept arbitrary keyword arguments; bit `0x20` is set if the function is a generator. See inspect-module-co-flags for details on the semantics of each flags that might be present.

Future feature declarations (`from __future__ import division`) also use bits in *co_flags* to indicate whether a code object was compiled with a particular feature enabled: bit `0x2000` is set if the function was compiled with future division enabled; bits `0x10` and `0x1000` were used in earlier versions of Python.

Other bits in *co_flags* are reserved for internal use.

If a code object represents a function, the first item in *co_consts* is the documentation string of the function, or `None` if undefined.

Methods on code objects

`codeobject.`**`co_positions`**`()`

Returns an iterable over the source code positions of each *bytecode* instruction in the code object.

The iterator returns `tuples` containing the `(start_line, end_line, start_column, end_column)`. The *i-th* tuple corresponds to the position of the source code that compiled to the *i-th* instruction. Column information is 0-indexed utf-8 byte offsets on the given source line.

A informação posicional pode estar ausente. Veja uma lista não-exaustiva de casos onde isso pode acontecer:

- Executando o interpretador com `no_debug_ranges -X`.
- Carregando um arquivo pyc compilado com `no_debug_ranges -X`.
- Tuplas posicionais correspondendo a instruções artificiais.
- Números de linha e coluna que não podem ser representados devido a limitações específicas de implementação.

Quando isso ocorre, alguns ou todos elementos da tupla podem ser `None`.

Novo na versão 3.11.

Nota: Esse recurso requer o armazenamento de posições de coluna no objeto código, o que pode resultar em um pequeno aumento no uso de memória do interpretador e no uso de disco para arquivos Python compilados. Para evitar armazenar as informações extras e/ou desativar a exibição das informações extras de rastreamento, use a opção de linha de comando `no_debug_ranges -X` ou a variável de ambiente `PYTHONNODEBUGRANGES`.

`codeobject.`**`co_lines`**`()`

Returns an iterator that yields information about successive ranges of *bytecode*s. Each item yielded is a `(start, end, lineno) tuple`:

- `start` (an `int`) represents the offset (inclusive) of the start of the *bytecode* range
- `end` (an `int`) represents the offset (exclusive) of the end of the *bytecode* range
- `lineno` is an `int` representing the line number of the *bytecode* range, or `None` if the bytecodes in the given range have no line number

The items yielded will have the following properties:

- The first range yielded will have a `start` of 0.
- The `(start, end)` ranges will be non-decreasing and consecutive. That is, for any pair of `tuples`, the `start` of the second will be equal to the `end` of the first.
- No range will be backwards: `end >= start` for all triples.
- The last `tuple` yielded will have `end` equal to the size of the *bytecode*.

Zero-width ranges, where `start == end`, are allowed. Zero-width ranges are used for lines that are present in the source code, but have been eliminated by the *bytecode* compiler.

Novo na versão 3.10.

Ver também:

PEP 626 - Precise line numbers for debugging and other tools.
The PEP that introduced the `co_lines()` method.

Objetos quadro

Frame objects represent execution frames. They may occur in *traceback objects*, and are also passed to registered trace functions.

Special read-only attributes

`frame.`**`f_back`**	Points to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame
`frame.`**`f_code`**	The *code object* being executed in this frame. Accessing this attribute raises an auditing event `object.__getattr__` with arguments `obj` and `"f_code"`.
`frame.`**`f_locals`**	The dictionary used by the frame to look up *local variables*
`frame.`**`f_globals`**	The dictionary used by the frame to look up *global variables*
`frame.`**`f_builtins`**	The dictionary used by the frame to look up *built-in (intrinsic) names*
`frame.`**`f_lasti`**	The “precise instruction” of the frame object (this is an index into the *bytecode* string of the *code object*)

Special writable attributes

`frame.`**`f_trace`**	If not `None`, this is a function called for various events during code execution (this is used by debuggers). Normally an event is triggered for each new source line (see *`f_trace_lines`*).
`frame.`**`f_trace_lines`**	Set this attribute to `False` to disable triggering a tracing event for each source line.
`frame.`**`f_trace_opcodes`**	Set this attribute to `True` to allow per-opcode events to be requested. Note that this may lead to undefined interpreter behaviour if exceptions raised by the trace function escape to the function being traced.
`frame.`**`f_lineno`**	The current line number of the frame – writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to this attribute.

Frame object methods

Objetos quadro têm suporte a um método:

`frame.`**`clear`**`()`

This method clears all references to *local variables* held by the frame. Also, if the frame belonged to a *generator*, the generator is finalized. This helps break reference cycles involving frame objects (for example when catching an exception and storing its *traceback* for later use).

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

Novo na versão 3.4.

Objetos traceback

Traceback objects represent the stack trace of an exception. A traceback object is implicitly created when an exception occurs, and may also be explicitly created by calling `types.TracebackType`.

Alterado na versão 3.7: Traceback objects can now be explicitly instantiated from Python code.

For implicitly created tracebacks, when the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section *A instrução try*.) It is accessible as the third item of the tuple returned by `sys.exc_info()`, and as the `__traceback__` attribute of the caught exception.

When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

For explicitly created tracebacks, it is up to the creator of the traceback to determine how the *`tb_next`* attributes should be linked to form a full stack trace.

Special read-only attributes:

traceback.**tb_frame**	Points to the execution *frame* of the current level. Accessing this attribute raises an auditing event `object.__getattr__` with arguments `obj` and `"tb_frame"`.
traceback.**tb_lineno**	Gives the line number where the exception occurred
traceback.**tb_lasti**	Indicates the "precise instruction".

The line number and last instruction in the traceback may differ from the line number of its *frame object* if the exception occurred in a *try* statement with no matching except clause or with a *finally* clause.

traceback.**tb_next**

The special writable attribute `tb_next` is the next level in the stack trace (towards the frame where the exception occurred), or `None` if there is no next level.

Alterado na versão 3.7: This attribute is now writable

Objetos slice

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

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

Objetos slice têm suporte a um método:

slice.**indices**(*self*, *length*)

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

Objetos método estático

Objetos método estático fornecem uma forma de transformar objetos função em objetos métodos descritos acima. Um objeto método estático é um invólucro em torno de qualquer outro objeto, comumente um objeto método definido pelo usuário. Quando um objeto método estático é recuperado de uma classe ou de uma instância de classe, o objeto retornado é o objeto encapsulado, do qual não está sujeito a nenhuma transformação adicional. Objetos método estático também são chamáveis. Objetos método estático são criados pelo construtor embutido `staticmethod()`.

Objetos método de classe

A class method object, like a static method object, is a wrapper around another object that alters the way in which that object is retrieved from classes and class instances. The behaviour of class method objects upon such retrieval is described above, under *"instance methods"*. Class method objects are created by the built-in `classmethod()` constructor.

3.3 Nomes de métodos especiais

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

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

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

3.3.1 Personalização básica

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

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

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

Se `__new__()` é chamado durante a construção do objeto e retorna uma instância de *cls*, então o método `__init__()` da nova instância será chamado como `__init__(self[, ...])`, onde *self* é a nova instância e os argumentos restantes são os mesmos que foram passados para o construtor do objeto.

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

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

`object.`**`__init__`**`(`*self*`[, ...])`

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

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

`object.`**`__del__`**`(`*self*`)`

Chamado quando a instância está prestes a ser destruída. Também é chamada de finalizador ou (incorretamente) de destruidor. Se uma classe base tem um método `__del__()`, o método `__del__()` da classe

[2] The `__hash__()`, `__iter__()`, `__reversed__()`, and `__contains__()` methods have special handling for this; others will still raise a `TypeError`, but may do so by relying on the behavior that `None` is not callable.

derivada, se houver, deve chamá-lo explicitamente para garantir a exclusão adequada da parte da classe base da instância.

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

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

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

Detalhes da implementação do CPython: É possível que um ciclo de referência impeça que a contagem de referência de um objeto chegue a zero. Neste caso, mais tarde, o ciclo será detectado e deletado pelo *coletor de lixo cíclico*. Uma causa comum de referências cíclicas é quando uma exceção foi capturada em uma variável local. O locals do quadro então referencia a exceção, que referencia seu próprio traceback, que referencia o locals de todos os quadros capturados no traceback.

Ver também:

Documentação do módulo `gc`.

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

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

`object.`**`__repr__`**`(`*self*`)`

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

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

`object.`**`__str__`**`(`*self*`)`

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

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

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

object.**__bytes__**(*self*)

Chamado por bytes para calcular uma representação de string de bytes de um objeto. Isso deve retornar um objeto `bytes`.

object.**__format__**(*self*, *format_spec*)

Chamado pela função embutida `format()` e, por extensão, avaliação de *literais de string formatadas* e o método `str.format()`, para produzir uma representação de string "formatada" de um objeto. O argumento *format_spec* é uma string que contém uma descrição das opções de formatação desejadas. A interpretação do argumento *format_spec* depende do tipo que implementa *__format__()*, entretanto a maioria das classes delegará a formatação a um dos tipos embutidos ou usará uma sintaxe de opção de formatação semelhante.

Consulte formatspec para uma descrição da sintaxe de formatação padrão.

O valor de retorno deve ser um objeto string.

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

Alterado na versão 3.7: `object.__format__(x, '')` é agora equivalente a `str(x)` em vez de `format(str(x), '')`.

object.**__lt__**(*self*, *other*)

object.**__le__**(*self*, *other*)

object.**__eq__**(*self*, *other*)

object.**__ne__**(*self*, *other*)

object.**__gt__**(*self*, *other*)

object.**__ge__**(*self*, *other*)

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

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

Por padrão, `object` implementa *__eq__()* usando `is`, retornando `NotImplemented` no caso de uma comparação falsa: `True if x is y else NotImplemented`. Para *__ne__()*, por padrão ele delega para *__eq__()* e inverte o resultado a menos que seja `NotImplemented`. Não há outras relações implícitas entre os operadores de comparação ou implementações padrão; por exemplo, o valor verdadeiro de `(x<y or x==y)` não implica `x<=y`. Para gerar operações de ordenação automaticamente a partir de uma única operação raiz, consulte `functools.total_ordering()`.

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

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

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

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

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

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

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

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

Uma classe que sobrescreve *`__eq__()`* e não define *`__hash__()`* terá seu *`__hash__()`* implicitamente definido como `None`. Quando o método *`__hash__()`* de uma classe é `None`, as instâncias da classe levantam uma `TypeError` apropriada quando um programa tenta recuperar seu valor hash, e também será identificado corretamente como não-hasheável ao verificar `isinstance(obj, collections.abc.Hashable)`.

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

Se uma classe que não substitui *`__eq__()`* deseja suprimir o suporte a hash, deve incluir `__hash__ = None` na definição de classe. Uma classe que define seu próprio *`__hash__()`* que levanta explicitamente uma `TypeError` seria incorretamente identificada como hasheável por uma chamada `isinstance(obj, collections.abc.Hashable)`.

Nota: Por padrão, os valores *`__hash__()`* dos objetos str e bytes são "salgados" com um valor aleatório imprevisível. Embora permaneçam constantes em um processo individual do Python, eles não são previsíveis entre invocações repetidas do Python.

This is intended to provide protection against a denial-of-service caused by carefully chosen inputs that exploit the worst case performance of a dict insertion, $O(n^2)$ complexity. See http://ocert.org/advisories/ocert-2011-003.html for details.

Alterar os valores de hash afeta a ordem de iteração dos conjuntos. Python nunca deu garantias sobre essa ordem (e normalmente varia entre compilações de 32 e 64 bits).

Consulte também `PYTHONHASHSEED`.

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

`object.`**`__bool__`**`(`*self*`)`

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

3.3.2 Personalizando o acesso aos atributos

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

`object.`**`__getattr__`**(*self*, *name*)

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

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

`object.`**`__getattribute__`**(*self*, *name*)

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

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

Levanta um evento de auditoria `object.__getattr__` com argumentos `obj`, `name`.

`object.`**`__setattr__`**(*self*, *name*, *value*)

Chamado quando se tenta efetuar uma atribuição de atributos. Esse método é chamado em vez do mecanismo normal (ou seja, armazena o valor no dicionário da instância). *name* é o nome do atributo, *value* é o valor a ser atribuído a ele.

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

Levanta um evento de auditoria `object.__setattr__` com argumentos `obj`, `name`, `value`.

`object.`**`__delattr__`**(*self*, *name*)

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

Levanta um evento de auditoria `object.__delattr__` com argumentos `obj`, `name`.

`object.`**`__dir__`**(*self*)

Chamado quando `dir()` é chamado com o objeto como argumento. Uma sequência deve ser retornada. `dir()` converte a sequência retornada em uma lista e a ordena.

Personalizando acesso a atributos de módulos

Os nomes especiais `__getattr__` e `__dir__` também podem ser usados para personalizar o acesso aos atributos dos módulos. A função `__getattr__` no nível do módulo deve aceitar um argumento que é o nome de um atributo e retornar o valor calculado ou levantar uma exceção `AttributeError`. Se um atributo não for encontrado em um objeto de módulo por meio da pesquisa normal, por exemplo *object.__getattribute__()*, então `__getattr__` é pesquisado no módulo `__dict__` antes de levantar `AttributeError`. Se encontrado, ele é chamado com o nome do atributo e o resultado é retornado.

A função `__dir__` não deve aceitar nenhum argumento e retorna uma sequência de strings que representa os nomes acessíveis no módulo. Se presente, esta função substitui a pesquisa padrão `dir()` em um módulo.

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

```
import sys
from types import ModuleType

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

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

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

Nota: Definir `__getattr__` no módulo e configurar o `__class__` do módulo só afeta as pesquisas feitas usando a sintaxe de acesso ao atributo – acessar diretamente os globais do módulo (seja por código dentro do módulo, ou por meio de uma referência ao dicionário global do módulo) não tem efeito.

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

Novo na versão 3.7: Atributos de módulo `__getattr__` e `__dir__`.

Ver também:

PEP 562 - __getattr__ e __dir__ de módulo
Descreve as funções `__getattr__` e `__dir__` nos módulos.

Implementando descritores

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

`object.`**`__get__`**`(`*self*`,` *instance*`,` *owner=None*`)`

Chamado para obter o atributo da classe proprietária (acesso ao atributo da classe) ou de uma instância dessa classe (acesso ao atributo da instância). O argumento opcional *owner* é a classe proprietária, enquanto *instance* é a instância pela qual o atributo foi acessado, ou `None` quando o atributo é acessado por meio de *owner*.

Este método deve retornar o valor do atributo calculado ou levantar uma exceção `AttributeError`.

PEP 252 especifica que *__get__()* é um chamável com um ou dois argumentos. Os próprios descritores embutidos do Python implementam esta especificação; no entanto, é provável que algumas ferramentas de terceiros tenham descritores que requerem ambos os argumentos. A implementação de *__getattribute__()* do próprio Python sempre passa em ambos os argumentos sejam eles requeridos ou não.

object.**__set__**(*self*, *instance*, *value*)

Chamado para definir o atributo em uma instância *instance* da classe proprietária para um novo valor, *value*.

Observe que adicionar *__set__()* ou *__delete__()* altera o tipo de descritor para um "descritor de dados". Consulte *Invocando descritores* para mais detalhes.

object.**__delete__**(*self*, *instance*)

Chamado para excluir o atributo em uma instância *instance* da classe proprietária.

Instâncias de descritores também podem ter o atributo `__objclass__` presente:

object.**__objclass__**

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

Invocando descritores

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

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

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

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

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

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

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

Ligação de super
Uma pesquisa pontilhada, ou *dotted lookup*, como `super(A, a).x` procura `a.__class__.__mro__` por uma classe base `B` seguindo `A` e então retorna `B.__dict__['x'].__get__(a, A)`. Se não for um descritor, `x` é retornado inalterado.

Para ligações de instâncias, a precedência de invocação do descritor depende de quais métodos do descritor são definidos. Um descritor pode definir qualquer combinação de *__get__()*, *__set__()* e *__delete__()*. Se ele não definir `__get__()`, então acessar o atributo retornará o próprio objeto descritor, a menos que haja um valor no dicionário de instância do objeto. Se o descritor define `__set__()` e/ou `__delete__()`, é um descritor de dados; se não definir nenhum, é um descritor sem dados. Normalmente, os descritores de dados definem `__get__()` e `__set__()`, enquanto os descritores sem dados têm apenas o método `__get__()`. Descritores de dados com `__get__()` e `__set__()` (e/ou `__delete__()`) definidos sempre substituem uma redefinição em um dicionário de instância. Em contraste, descritores sem dados podem ser substituídos por instâncias.

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

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

__slots__

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

O espaço economizado com o uso de `__dict__` pode ser significativo. A velocidade de pesquisa de atributos também pode ser significativamente melhorada.

`object.`**`__slots__`**

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

Observações ao uso de *__slots__*:

- Ao herdar de uma classe sem *__slots__*, os atributos `__dict__` e *__weakref__* das instâncias sempre estarão acessíveis.
- Sem uma variável `__dict__`, as instâncias não podem ser atribuídas a novas variáveis não listadas na definição *__slots__*. As tentativas de atribuir a um nome de variável não listado levantam `AttributeError`. Se a atribuição dinâmica de novas variáveis for desejada, então adicione `'__dict__'` à sequência de strings na declaração de *__slots__*.
- Sem uma variável *__weakref__* para cada instância, as classes que definem *__slots__* não suportam `referências fracas` para suas instâncias. Se for necessário um suporte de referência fraca, adicione `'__weakref__'` à sequência de strings na declaração *__slots__*.
- *__slots__* são implementados no nível de classe criando *descritores* para cada nome de variável. Como resultado, os atributos de classe não podem ser usados para definir valores padrão para variáveis de instância definidas por *__slots__*; caso contrário, o atributo de classe substituiria a atribuição do descritor.
- A ação de uma declaração *__slots__* se limita à classe em que é definida. *__slots__* declarados em uma classe base estão disponíveis nas subclasses. No entanto, as subclasses receberão um `__dict__` e *__weakref__* a menos que também definam *__slots__* (que deve conter apenas nomes de quaisquer slots *adicionais*).
- Se uma classe define um slot também definido em uma classe base, a variável de instância definida pelo slot da classe base fica inacessível (exceto por recuperar seu descritor diretamente da classe base). Isso torna o significado do programa indefinido. No futuro, uma verificação pode ser adicionada para evitar isso.
- `TypeError` será levantada se *__slots__* não vazios forem definidos para uma classe derivada de um tipo embutido `"variable-length"` como `int`, `bytes` e `tuple`.
- Qualquer *iterável* não string pode ser atribuído a *__slots__*.
- Se um `dicionário` for usado para atribuir *__slots__*, as chaves do dicionário serão usadas como os nomes dos slots. Os valores do dicionário podem ser usados para fornecer strings de documentação (docstrings) por atributo que serão reconhecidos por `inspect.getdoc()` e exibidos na saída de `help()`.
- Atribuição de `__class__` funciona apenas se ambas as classes têm o mesmo *__slots__*.
- A herança múltipla com várias classes bases com slots pode ser usada, mas apenas uma classe base tem permissão para ter atributos criados por slots (as outras classes bases devem ter layouts de slots vazios) – violações levantam `TypeError`.
- Se um *iterador* for usado para *__slots__*, um *descritor* é criado para cada um dos valores do iterador. No entanto, o atributo *__slots__* será um iterador vazio.

3.3.3 Personalizando a criação de classe

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

classmethod `object.__init_subclass__`(*cls*)

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

Keyword arguments which are given to a new class are passed to the parent class's `__init_subclass__`. For compatibility with other classes using `__init_subclass__`, one should take out the needed keyword arguments and pass the others over to the base class, as in:

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

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

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

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

Novo na versão 3.6.

Quando uma classe é criada, `type.__new__()` verifica as variáveis de classe e faz chamadas a funções de retorno (callback) para aqueles com um gancho *__set_name__()*.

`object.__set_name__`(*self*, *owner*, *name*)

Chamado automaticamente no momento em que a classe proprietária *owner* é criada. O objeto foi atribuído a *name* nessa classe:

```
class A:
    x = C()  # Automatically calls: x.__set_name__(A, 'x')
```

Se a variável de classe for atribuída após a criação da classe, *__set_name__()* não será chamado automaticamente. Se necessário, *__set_name__()* pode ser chamado diretamente:

```
class A:
   pass

c = C()
A.x = c                  # The hook is not called
c.__set_name__(A, 'x')   # Manually invoke the hook
```

Consulte *Criando o objeto classe* para mais detalhes.

Novo na versão 3.6.

Metaclasses

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

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

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

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

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

- entradas de MRO são resolvidas;
- a metaclasse apropriada é determinada;
- o espaço de nomes da classe é preparada;
- o corpo da classe é executado;
- o objeto da classe é criado.

Resolvendo entradas de MRO

`object.`**`__mro_entries__`**(*self*, *bases*)

Se uma classe base que aparece em uma definição de classe não é uma instância de `type`, então um método `__mro_entries__()` é procurado na base. Se um método `__mro_entries__()` é encontrado, a base é substituída pelo resultado de uma chamada para `__mro_entries__()` ao criar a classe. O método é chamado com a tupla de bases original passada como parâmetro *bases*, e deve retornar uma tupla de classes que serão usadas no lugar da base. A tupla retornada pode estar vazia: nesses casos, a base original é ignorada.

Ver também:

`types.resolve_bases()`
Dinamicamente resolve bases que não são instâncias de `type`.

`types.get_original_bases()`
Recupera as "bases originais" de uma classe antes das modificações feitas por *`__mro_entries__()`*.

PEP 560
Suporte básico para módulo typing e tipos genéricos.

Determinando a metaclasse apropriada

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

- se nenhuma classe base e nenhuma metaclasse explícita forem fornecidas, então `type()` é usada;
- se uma metaclasse explícita é fornecida e *não* é uma instância de `type()`, então ela é usada diretamente como a metaclasse;
- se uma instância de `type()` é fornecida como a metaclasse explícita, ou classes bases são definidas, então a metaclasse mais derivada é usada.

A metaclasse mais derivada é selecionada a partir da metaclasse explicitamente especificada (se houver) e das metaclasses (ou seja, `type(cls)`) de todas as classes bases especificadas. A metaclasse mais derivada é aquela que é um subtipo de *todas* essas metaclasses candidatas. Se nenhuma das metaclasses candidatas atender a esse critério, a definição de classe falhará com `TypeError`.

Preparando o espaço de nomes da classe

Uma vez identificada a metaclasse apropriada, o espaço de nomes da classe é preparado. Se a metaclasse tiver um atributo `__prepare__`, ela será chamada como `namespace = metaclass.__prepare__(name, bases, **kwds)` (onde os argumentos nomeados adicionais, se houver, vêm da definição de classe). O método `__prepare__` deve ser implementado como um `classmethod`. O espaço de nomes retornado por `__prepare__` é passado para `__new__`, mas quando o objeto classe final é criado, o espaço de nomes é copiado para um novo `dict`.

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

Ver também:

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

Executando o corpo da classe

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

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

Criando o objeto classe

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

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

Detalhes da implementação do CPython: No CPython 3.6 e posterior, a célula `__class__` é passada para a metaclasse como uma entrada de `__classcell__` no espaço de nomes da classe. Se estiver presente, deve

ser propagado até a chamada a `type.__new__` para que a classe seja inicializada corretamente. Não fazer isso resultará em um `RuntimeError` no Python 3.8.

Quando usada a metaclasse padrão `type`, ou qualquer metaclasse que chame `type.__new__`, as seguintes etapas de personalização adicionais são executadas depois da criação do objeto classe:

1) O método `type.__new__` coleta todos os atributos no espaço de nomes da classe que definem um método *`__set_name__()`*;
2) Esses métodos `__set_name__` são chamados com a classe sendo definida e o nome atribuído para este atributo específico;
3) O gancho *`__init_subclass__()`* é chamado na classe base imediata da nova classe em sua ordem de resolução de método.

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

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

Ver também:

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

Usos para metaclasses

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

3.3.4 Personalizando verificações de instância e subclasse

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

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

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

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

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

Ver também:

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

3.3.5 Emulando tipos genéricos

Quando estiver usando *anotações de tipo*, é frequentemente útil *parametrizar* um *tipo genérico* usando a notação de colchetes do Python. Por exemplo, a anotação `list[int]` pode ser usada para indicar uma `list` em que todos os seus elementos são do tipo `int`.

Ver também:

PEP 484 - Dicas de tipo
Apresenta a estrutura do Python para anotações de tipo

Tipos Generic Alias
Documentação de objetos que representam classes genéricas parametrizadas

Generics, genéricos definidos pelo usuário e `typing.Generic`
Documentação sobre como implementar classes genéricas que podem ser parametrizadas em tempo de execução e compreendidas por verificadores de tipo estático

Uma classe pode *geralmente* ser parametrizada somente se ela define o método de classe especial `__class_getitem__()`.

classmethod `object.__class_getitem__`(*cls*, *key*)

Retorna um objeto que representa a especialização de uma classe genérica por argumentos de tipo encontrados em *key*.

Quando definido em uma classe, `__class_getitem__()` é automaticamente um método de classe. Assim, não é necessário que seja decorado com `@classmethod` quando de sua definição.

O propósito de *__class_getitem__*

O propósito de *`__class_getitem__()`* é permitir a parametrização em tempo de execução de classes genéricas da biblioteca padrão, a fim de aplicar mais facilmente *dicas de tipo* a essas classes.

Para implementar classes genéricas personalizadas que podem ser parametrizadas em tempo de execução e compreendidas por verificadores de tipo estáticos, os usuários devem herdar de uma classe da biblioteca padrão que já implementa *`__class_getitem__()`*, ou herdar de `typing.Generic`, que possui sua própria implementação de `__class_getitem__()`.

Implementações personalizadas de *`__class_getitem__()`* em classes definidas fora da biblioteca padrão podem não ser compreendidas por verificadores de tipo de terceiros, como o mypy. O uso de `__class_getitem__()` em qualquer classe para fins diferentes de dicas de tipo é desencorajado.

__class_getitem__ versus *__getitem__*

Normalmente, a *subscription* de um objeto usando colchetes chamará o método de instância *`__getitem__()`* definido na classe do objeto. No entanto, se o objeto sendo subscrito for ele mesmo uma classe, o método de classe *`__class_getitem__()`* pode ser chamado em seu lugar. `__class_getitem__()` deve retornar um objeto GenericAlias se estiver devidamente definido.

Apresentado com a *expressão* `obj[x]`, o interpretador de Python segue algo parecido com o seguinte processo para decidir se *`__getitem__()`* ou *`__class_getitem__()`* deve ser chamado:

```
from inspect import isclass

def subscribe(obj, x):
    """Return the result of the expression 'obj[x]'"""

    class_of_obj = type(obj)

    # If the class of obj defines __getitem__,
    # call class_of_obj.__getitem__(obj, x)
```

(continua na próxima página)

(continuação da página anterior)

```
    if hasattr(class_of_obj, '__getitem__'):
        return class_of_obj.__getitem__(obj, x)

    # Else, if obj is a class and defines __class_getitem__,
    # call obj.__class_getitem__(x)
    elif isclass(obj) and hasattr(obj, '__class_getitem__'):
        return obj.__class_getitem__(x)

    # Else, raise an exception
    else:
        raise TypeError(
            f"'{class_of_obj.__name__}' object is not subscriptable"
        )
```

Em Python, todas as classes são elas mesmas instâncias de outras classes. A classe de uma classe é conhecida como *metaclasse* dessa classe, e a maioria das classes tem a classe `type` como sua metaclasse. `type` não define *`__getitem__()`*, o que significa que expressões como `list[int]`, `dict[str, float]` e `tuple[str, bytes]` resultam em chamadas para *`__class_getitem__()`*:

```
>>> # list has class "type" as its metaclass, like most classes:
>>> type(list)
<class 'type'>
>>> type(dict) == type(list) == type(tuple) == type(str) == type(bytes)
True
>>> # "list[int]" calls "list.__class_getitem__(int)"
>>> list[int]
list[int]
>>> # list.__class_getitem__ returns a GenericAlias object:
>>> type(list[int])
<class 'types.GenericAlias'>
```

No entanto, se uma classe tiver uma metaclasse personalizada que define *`__getitem__()`*, subscrever a classe pode resultar em comportamento diferente. Um exemplo disso pode ser encontrado no módulo `enum`:

```
>>> from enum import Enum
>>> class Menu(Enum):
...     """A breakfast menu"""
...     SPAM = 'spam'
...     BACON = 'bacon'
...
>>> # Enum classes have a custom metaclass:
>>> type(Menu)
<class 'enum.EnumMeta'>
>>> # EnumMeta defines __getitem__,
>>> # so __class_getitem__ is not called,
>>> # and the result is not a GenericAlias object:
>>> Menu['SPAM']
<Menu.SPAM: 'spam'>
>>> type(Menu['SPAM'])
<enum 'Menu'>
```

Ver também:

PEP 560 - Suporte básico para módulo typing e tipos genéricos
Introduz *`__class_getitem__()`*, e define quando uma *subscrição* resulta na chamada de `__class_getitem__()` em vez de *`__getitem__()`*

3.3.6 Emulando objetos chamáveis

`object.__call__`(*self*[, *args...*])

Chamado quando a instância é "chamada" como uma função; se este método for definido, `x(arg1, arg2, ...)` basicamente traduz para `type(x).__call__(x, arg1, ...)`.

3.3.7 Emulando de tipos contêineres

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

`object.__len__`(*self*)

Called to implement the built-in function `len()`. Should return the length of the object, an integer >= 0. Also, an object that doesn't define a *`__bool__()`* method and whose `__len__()` method returns zero is considered to be false in a Boolean context.

Detalhes da implementação do CPython: In CPython, the length is required to be at most `sys.maxsize`. If the length is larger than `sys.maxsize` some features (such as `len()`) may raise `OverflowError`. To prevent raising `OverflowError` by truth value testing, an object must define a *`__bool__()`* method.

`object.__length_hint__`(*self*)

Chamado para implementar `operator.length_hint()`. Deve retornar um comprimento estimado para o objeto (que pode ser maior ou menor que o comprimento real). O comprimento deve ser um inteiro >= 0. O valor de retorno também pode ser `NotImplemented`, que é tratado da mesma forma como se o método `__length_hint__` não existisse. Este método é puramente uma otimização e nunca é necessário para a correção.

Novo na versão 3.4.

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

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

é traduzida com

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

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

`object.__getitem__`(*self*, *key*)

Called to implement evaluation of `self[key]`. For *sequence* types, the accepted keys should be integers. Optionally, they may support `slice` objects as well. Negative index support is also optional. If *key* is of an

inappropriate type, `TypeError` may be raised; if *key* is a value outside the set of indexes for the sequence (after any special interpretation of negative values), `IndexError` should be raised. For *mapping* types, if *key* is missing (not in the container), `KeyError` should be raised.

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

Nota: When *subscripting* a *class*, the special class method *__class_getitem__()* may be called instead of `__getitem__()`. See *__class_getitem__ versus __getitem__* for more details.

`object.`**`__setitem__`**(*self*, *key*, *value*)

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

`object.`**`__delitem__`**(*self*, *key*)

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

`object.`**`__missing__`**(*self*, *key*)

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

`object.`**`__iter__`**(*self*)

This method is called when an *iterator* is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container.

`object.`**`__reversed__`**(*self*)

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

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

Os operadores de teste de associação (*in* e *not in*) são normalmente implementados como uma iteração através de um contêiner. No entanto, os objetos contêiner podem fornecer o seguinte método especial com uma implementação mais eficiente, que também não requer que o objeto seja iterável.

`object.`**`__contains__`**(*self*, *item*)

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

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

3.3.8 Emulando tipos numéricos

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

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

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). For instance, to evaluate the expression `x + y`, where *x* is an instance of a class that has an *`__add__()`* method, `type(x).__add__(x, y)` is called. The *`__divmod__()`* method should be the equivalent to using *`__floordiv__()`* and *`__mod__()`*; it should not be related to *`__truediv__()`*. Note that *`__pow__()`* should be defined to accept an optional third argument if the ternary version of the built-in `pow()` function is to be supported.

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

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

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation[3] and the operands are of different types.[4] For instance, to evaluate the expression `x - y`, where *y* is an instance of a class that has an *`__rsub__()`* method, `type(y).__rsub__(y, x)` is called if `type(x).__sub__(x, y)` returns *NotImplemented*.

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

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

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

Nota: Se o tipo do operando direito for uma subclasse do tipo do operando esquerdo e essa subclasse fornecer uma implementação diferente do método refletido para a operação, este método será chamado antes do método não refletido do operando esquerdo. Esse comportamento permite que as subclasses substituam as operações de seus ancestrais.

`object.`**`__iadd__`**(*self*, *other*)

`object.`**`__isub__`**(*self*, *other*)

`object.`**`__imul__`**(*self*, *other*)

`object.`**`__imatmul__`**(*self*, *other*)

`object.`**`__itruediv__`**(*self*, *other*)

`object.`**`__ifloordiv__`**(*self*, *other*)

`object.`**`__imod__`**(*self*, *other*)

`object.`**`__ipow__`**(*self*, *other*[, *modulo*])

`object.`**`__ilshift__`**(*self*, *other*)

`object.`**`__irshift__`**(*self*, *other*)

`object.`**`__iand__`**(*self*, *other*)

`object.`**`__ixor__`**(*self*, *other*)

`object.`**`__ior__`**(*self*, *other*)

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

`object.`**`__neg__`**(*self*)

`object.`**`__pos__`**(*self*)

`object.`**`__abs__`**(*self*)

`object.`**`__invert__`**(*self*)

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

`object.`**`__complex__`**(*self*)

`object.`**`__int__`**(*self*)

`object.`**`__float__`**(*self*)

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

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

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

Se *`__int__()`*, *`__float__()`* e *`__complex__()`* não estiverem definidos, funções embutidas correspondentes `int()`, `float()` e `complex()` recorre a *`__index__()`*.

`object.`**`__round__`**(*self*[, *ndigits*])

`object.`**`__trunc__`**(*self*)

`object.`**`__floor__`**(*self*)

object.**__ceil__**(*self*)

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

The built-in function `int()` falls back to *__trunc__()* if neither *__int__()* nor *__index__()* is defined.

Alterado na versão 3.11: The delegation of `int()` to *__trunc__()* is deprecated.

3.3.9 Gerenciadores de contexto da instrução with

Um *gerenciador de contexto* é um objeto que define o contexto de tempo de execução a ser estabelecido ao executar uma instrução *with*. O gerenciador de contexto lida com a entrada e a saída do contexto de tempo de execução desejado para a execução do bloco de código. Os gerenciadores de contexto são normalmente invocados usando a instrução `with` (descrita na seção *The with statement*), mas também podem ser usados invocando diretamente seus métodos.

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

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

object.**__enter__**(*self*)

Insere o contexto de tempo de execução relacionado a este objeto. A instrução *with* vinculará o valor de retorno deste método ao(s) alvo(s) especificado(s) na cláusula `as` da instrução, se houver.

object.**__exit__**(*self*, *exc_type*, *exc_value*, *traceback*)

Sai do contexto de tempo de execução relacionado a este objeto. Os parâmetros descrevem a exceção que fez com que o contexto fosse encerrado. Se o contexto foi encerrado sem exceção, todos os três argumentos serão `None`.

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

Note that *__exit__()* methods should not reraise the passed-in exception; this is the caller's responsibility.

Ver também:

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

3.3.10 Customizando argumentos posicionais na classe correspondência de padrão

When using a class name in a pattern, positional arguments in the pattern are not allowed by default, i.e. `case MyClass(x, y)` is typically invalid without special support in `MyClass`. To be able to use that kind of pattern, the class needs to define a *__match_args__* attribute.

object.**__match_args__**

Essa variável de classe pode ser atribuída a uma tupla de strings. Quando essa classe é usada em uma classe padrão com argumentos posicionais, cada argumento posicional será convertido para um argumento nomeado, usando correspondência de valor em *__match_args__* como palavra reservada. A ausência desse atributo é equivalente a defini-lo como `()`

Por exemplo, se `MyClass.__match_args__` é `("left", "center", "right")` significa que `case MyClass(x, y)` é equivalente a `case MyClass(left=x, center=y)`. Note que o número de argumentos no padrão deve ser menor ou igual ao número de elementos em *__match_args__*; caso seja maior, a tentativa de correspondência de padrão irá levantar uma `TypeError`.

Novo na versão 3.10.

Ver também:

PEP 634 - Correspondência de Padrão Estrutural
A especificação para a instrução Python `match`

3.3.11 Emulating buffer types

The buffer protocol provides a way for Python objects to expose efficient access to a low-level memory array. This protocol is implemented by builtin types such as `bytes` and `memoryview`, and third-party libraries may define additional buffer types.

While buffer types are usually implemented in C, it is also possible to implement the protocol in Python.

`object.__buffer__`(*self*, *flags*)

Called when a buffer is requested from *self* (for example, by the `memoryview` constructor). The *flags* argument is an integer representing the kind of buffer requested, affecting for example whether the returned buffer is read-only or writable. `inspect.BufferFlags` provides a convenient way to interpret the flags. The method must return a `memoryview` object.

`object.__release_buffer__`(*self*, *buffer*)

Called when a buffer is no longer needed. The *buffer* argument is a `memoryview` object that was previously returned by *__buffer__()*. The method must release any resources associated with the buffer. This method should return `None`. Buffer objects that do not need to perform any cleanup are not required to implement this method.

Novo na versão 3.12.

Ver também:

PEP 688 - Making the buffer protocol accessible in Python
Introduces the Python `__buffer__` and `__release_buffer__` methods.

`collections.abc.Buffer`
ABC for buffer types.

3.3.12 Pesquisa de método especial

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

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

The rationale behind this behaviour lies with a number of special methods such as *__hash__()* and *__repr__()* that are implemented by all objects, including type objects. If the implicit lookup of these methods used the conventional lookup process, they would fail when invoked on the type object itself:

```
>>> 1 .__hash__() == hash(1)
True
>>> int.__hash__() == hash(int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor '__hash__' of 'int' object needs an argument
```

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

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

In addition to bypassing any instance attributes in the interest of correctness, implicit special method lookup generally also bypasses the *__getattribute__()* method even of the object's metaclass:

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

Bypassing the *__getattribute__()* machinery in this fashion provides significant scope for speed optimisations within the interpreter, at the cost of some flexibility in the handling of special methods (the special method *must* be set on the class object itself in order to be consistently invoked by the interpreter).

3.4 Corrotinas

3.4.1 Objetos aguardáveis

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

Nota: The *generator iterator* objects returned from generators decorated with `types.coroutine()` are also awaitable, but they do not implement *__await__()*.

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

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

Nota: The language doesn't place any restriction on the type or value of the objects yielded by the iterator returned by `__await__`, as this is specific to the implementation of the asynchronous execution framework (e.g. `asyncio`) that will be managing the *awaitable* object.

Novo na versão 3.5.

Ver também:

PEP 492 para informações adicionais sobre objetos aguardáveis.

3.4.2 Objetos corrotina

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

As corrotinas também têm os métodos listados abaixo, que são análogos aos dos geradores (ver *Métodos de iterador gerador*). No entanto, ao contrário dos geradores, as corrotinas não suportam diretamente a iteração.

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

`coroutine.`**`send`**`(`*value*`)`

Starts or resumes execution of the coroutine. If *value* is `None`, this is equivalent to advancing the iterator returned by `__await__()`. If *value* is not `None`, this method delegates to the `send()` method of the iterator that caused the coroutine to suspend. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the `__await__()` return value, described above.

`coroutine.`**`throw`**`(`*value*`)`

`coroutine.`**`throw`**`(`*type*[, *value*[, *traceback*]]`)`

Raises the specified exception in the coroutine. This method delegates to the `throw()` method of the iterator that caused the coroutine to suspend, if it has such a method. Otherwise, the exception is raised at the suspension point. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the `__await__()` return value, described above. If the exception is not caught in the coroutine, it propagates back to the caller.

Alterado na versão 3.12: A segunda assinatura (tipo[, valor[, traceback]]]) foi descontinuada e pode ser removida em uma versão futura do Python.

`coroutine.`**`close`**`()`

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

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

3.4.3 Iteradores assíncronos

Um *iterador assíncrono* pode chamar código assíncrono em seu método `__anext__`.

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

`object.`**`__aiter__`**`(`*self*`)`

Deve retornar um objeto *iterador assíncrono*.

`object.`**`__anext__`**`(`*self*`)`

Deve retornar um *aguardável* resultando em um próximo valor do iterador. Deve levantar um erro `StopAsyncIteration` quando a iteração terminar.

Um exemplo de objeto iterável assíncrono:

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

(continua na próxima página)

(continuação da página anterior)

```
    def __aiter__(self):
        return self

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

Novo na versão 3.5.

Alterado na versão 3.7: Prior to Python 3.7, `__aiter__()` could return an *awaitable* that would resolve to an *asynchronous iterator*.

Starting with Python 3.7, `__aiter__()` must return an asynchronous iterator object. Returning anything else will result in a `TypeError` error.

3.4.4 Gerenciadores de contexto assíncronos

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

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

`object.`**`__aenter__`**(*self*)

Semantically similar to `__enter__()`, the only difference being that it must return an *awaitable*.

`object.`**`__aexit__`**(*self*, *exc_type*, *exc_value*, *traceback*)

Semantically similar to `__exit__()`, the only difference being that it must return an *awaitable*.

Um exemplo de uma classe gerenciadora de contexto assíncrona:

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

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

Novo na versão 3.5.

CAPÍTULO 4

Modelo de execução

4.1 Estrutura de um programa

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

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

4.2 Nomeação e ligação

4.2.1 Ligação de nomes

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

As seguintes construções ligam nomes:

- parâmetros formais para funções,
- definições de classe,
- definições de função,
- expressões de atribuição,
- *alvos* que são identificadores se ocorrerem em uma atribuição:
  - cabeçalho de laço `for`,
  - depois de `as` em uma instrução `with`, cláusula `except`, cláusula `except*` ou no padrão as na correspondência de padrões estruturais,
  - em um padrão de captura na correspondência de padrões estruturais

- instruções *import*.
- instruções *type*.
- *listas de parâmetros de tipo*.

A instrução `import` no formato `from ... import *` liga todos os nomes definidos no módulo importado, exceto aqueles que começam com um sublinhado. Este formulário só pode ser usado no nível do módulo.

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

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

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

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

4.2.2 Resolução de nomes

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

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

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

Se a operação de ligação de nomes ocorre dentro de um bloco de código, todos os usos do nome dentro do bloco são tratadas como referências para o bloco atual. Isso pode. Isso pode levar a erros quando um nome é usado em um bloco antes de ser vinculado. Esta regra é sutil. Python carece de declarações e permite que as operações de ligação de nomes ocorram em qualquer lugar dentro de um bloco de código. As variáveis locais de um bloco de código podem ser determinadas pela varredura de todo o texto do bloco para operações de ligação de nome. Veja the FAQ entry on UnboundLocalError para exemplos.

Se a instrução *global* ocorrer dentro de um bloco, todos os usos dos nomes especificados na instrução referem-se às ligações desses nomes no espaço de nomes de nível superior. Os nomes são resolvidos no espaço de nomes de nível superior pesquisando o espaço de nomes global, ou seja, o espaço de nomes do módulo que contém o bloco de código, e o espaço de nomes interno, o espaço de nomes do módulo `builtins`. O espaço de nomes global é pesquisado primeiro. Se os nomes não forem encontrados lá, o espaço de nomes interno será pesquisado. A instrução `global` deve preceder todos os usos dos nomes listados.

A instrução *global* tem o mesmo escopo que uma operação de ligação de nome no mesmo bloco. Se o escopo mais próximo de uma variável livre contiver uma instrução global, a variável livre será tratada como global.

A instrução *nonlocal* faz com que os nomes correspondentes se refiram a variáveis previamente vinculadas no escopo da função delimitadora mais próxima. A exceção `SyntaxError` é levantada em tempo de compilação se o nome fornecido não existir em nenhum escopo de função delimitador. *Parâmetros de tipo* não podem ser vinculadas novamente com a instrução `nonlocal`.

O espaço de nomes de um módulo é criado automaticamente na primeira vez que um módulo é importado. O módulo principal de um script é sempre chamado de `__main__`.

Blocos de definição de classe e argumentos para `exec()` e `eval()` são especiais no contexto de resolução de nome. Uma definição de classe é uma instrução executável que pode usar e definir nomes. Essas referências seguem as regras normais para resolução de nome, com exceção de que variáveis locais não vinculadas são pesquisadas no espaço de nomes global global. O espaço de nomes global da definição de classe se torna o dicionário de atributos da classe. O

escopo dos nomes definidos em um bloco de classe é limitado ao bloco de classe; ele não se estende aos blocos de código de métodos. Isso inclui compreensões e expressões geradoras, mas não inclui *escopos de anotação*, que têm acesso a seus escopos de classe delimitadores. Isso significa que o seguinte falhará:

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

Porém, o seguinte vai funcionar:

```
class A:
    type Alias = Nested
    class Nested: pass

print(A.Alias.__value__)  # <type 'A.Nested'>
```

4.2.3 Escopos de anotação

As instruções `type` e *listas de parâmetros de tipo* introduzem *escopos de anotação*, que se comportam principalmente como escopos de função, mas com algumas exceções discutidas abaixo. *Anotações* atualmente não usam escopos de anotação, mas espera-se que elas usem escopos de anotação no Python 3.13 quando **PEP 649** for implementada.

Os escopos de anotação são usados nos seguintes contextos:

- Listas de parâmetros de tipo para *apelidos de tipo genérico*.
- Listas de parâmetros de tipo para *funções genéricas*. As anotações de uma função genérica são executadas dentro do escopo de anotação, mas seus padrões e decoradores não.
- Listas de parâmetros de tipo para *classes genéricas*. As classes base e argumentos nomeados de uma classe genérica são executadas dentro do escopo de anotação, mas seus decoradores não.
- Os limites e restrições para variáveis de tipo (*avaliadas de forma preguiçosa*).
- O valor dos apelidos de tipo (*avaliado de forma preguiçosa*).

Escopos de anotação diferenciam-se de escopos de função nas seguintes formas:

- Os escopos de anotação têm acesso ao espaço de nomes da classe delimitadora. Se um escopo de anotação estiver imediatamente dentro de um escopo de classe ou dentro de outro escopo de anotação que esteja imediatamente dentro de um escopo de classe, o código no escopo de anotação poderá usar nomes definidos no escopo de classe como se fosse executado diretamente no corpo da classe. Isto contrasta com funções regulares definidas dentro de classes, que não podem acessar nomes definidos no escopo da classe.
- Expressões em escopos de anotação não podem conter expressões `yield`, `yield from`, `await` ou `:= 1`. (Essas expressões são permitidas em outros escopos contidos no escopo de anotação.)
- Nomes definidos em escopos de anotação não podem ser vinculados novamente com instruções `nonlocal` em escopos internos. Isso inclui apenas parâmetros de tipo, pois nenhum outro elemento sintático que pode aparecer nos escopos de anotação pode introduzir novos nomes.
- Embora os escopos de anotação tenham um nome interno, esse nome não é refletido no *__qualname__* dos objetos definidos dentro do escopo. Em vez disso, o `__qualname__` de tais objetos é como se o objeto fosse definido no escopo delimitador.

Novo na versão 3.12: Escopos de anotação foram introduzidos no Python 3.12 como parte da **PEP 695**.

4.2.4 Avaliação preguiçosa

Os valores dos apelidos de tipo criados através da instrução `type` são *avaliados preguiçosamente*. O mesmo se aplica aos limites e restrições das variáveis de tipo criadas através da *sintaxe do parâmetros de tipo*. Isso significa que eles não são avaliados quando o apelido de tipo ou a variável de tipo é criado. Em vez disso, eles são avaliados apenas quando isso é necessário para resolver um acesso de atributo.

Exemplo:

```
>>> type Alias = 1/0
>>> Alias.__value__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> def func[T: 1/0](): pass
>>> T = func.__type_params__[0]
>>> T.__bound__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

Aqui a exceção é levantada apenas quando o atributo `__value__` do apelido de tipo ou o atributo `__bound__` da variável de tipo é acessado.

Esse comportamento é útil principalmente para referências a tipos que ainda não foram definidos quando o alias de tipo ou variável de tipo é criado. Por exemplo, a avaliação lenta permite a criação de apelidos de tipo mutuamente recursivos:

```
from typing import Literal

type SimpleExpr = int | Parenthesized
type Parenthesized = tuple[Literal["("], Expr, Literal[")"]]
type Expr = SimpleExpr | tuple[SimpleExpr, Literal["+", "-"], Expr]
```

Valores avaliados preguiçosamente são avaliados em *escopo de anotação*, o que significa que os nomes que aparecem dentro do valor avaliado lentamente são pesquisados como se fossem usados no escopo imediatamente envolvente.

Novo na versão 3.12.

4.2.5 Builtins e execução restrita

Detalhes da implementação do CPython: Os usuários não devem tocar em `__builtins__`; é estritamente um detalhe de implementação. Usuários que desejam substituir valores no espaço de nomes interno devem `import` o módulo `builtins` e modificar seus atributos apropriadamente.

O espaço de nomes builtins associado com a execução de um bloco de código é encontrado procurando o nome `__builtins__` em seu espaço de nomes global; este deve ser um dicionário ou um módulo (no último caso, o dicionário do módulo é usado). Por padrão, quando no módulo `__main__`, `__builtins__` é o módulo embutido `builtins`; quando em qualquer outro módulo, `__builtins__` é um apelido para o dicionário do próprio módulo `builtins`.

4.2.6 Interação com recursos dinâmicos

A resolução de nome de variáveis livres ocorre em tempo de execução, não em tempo de compilação. Isso significa que o código a seguir imprimirá 42:

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

As funções `eval()` e `exec()` não têm acesso ao ambiente completo para resolução de nome. Os nomes podem ser resolvidos nos espaços de nomes locais e globais do chamador. Variáveis livres não são resolvidas no espaço de nomes mais próximo, mas no espaço de nomes global.[1] As funções `exec()` e `eval()` possuem argumentos opcionais para substituir o espaço de nomes global e local. Se apenas um espaço de nomes for especificado, ele será usado para ambos.

4.3 Exceções

As exceções são um meio de romper o fluxo normal de controle de um bloco de código para tratar erros ou outras condições excepcionais. Uma exceção é *levantada* no ponto em que o erro é detectado; ele pode ser *tratado* pelo bloco de código circundante ou por qualquer bloco de código que invocou direta ou indiretamente o bloco de código onde ocorreu o erro.

O interpretador Python levanta uma exceção quando detecta um erro em tempo de execução (como divisão por zero). Um programa Python também pode levantar explicitamente uma exceção com a instrução *`raise`*. Os tratadores de exceção são especificados com a instrução *`try`* ... *`except`*. A cláusula *`finally`* de tal declaração pode ser usada para especificar o código de limpeza que não trata a exceção, mas é executado se uma exceção ocorreu ou não no código anterior.

Python usa o modelo de "terminação" da manipulação de erros: um manipulador de exceção pode descobrir o que aconteceu e continuar a execução em um nível externo, mas não pode reparar a causa do erro e tentar novamente a operação com falha (exceto reinserindo a parte incorreta de código de cima).

Quando uma exceção não é manipulada, o interpretador encerra a execução do programa ou retorna ao seu laço principal interativo. Em ambos os casos, ele exeibe um traceback (situação da pilha de execução), exceto quando a exceção é `SystemExit`.

As exceções são identificadas por instâncias de classe. A cláusula *`except`* é selecionada dependendo da classe da instância: ela deve referenciar a classe da instância ou uma *classe base não-virtual* dela. A instância pode ser recebida pelo manipulador e pode conter informações adicionais sobre a condição excepcional.

Nota: As mensagens de exceção não fazem parte da API do Python. Seu conteúdo pode mudar de uma versão do Python para outra sem aviso e não deve ser invocado pelo código que será executado em várias versões do interpretador.

Veja também a descrição da declaração *`try`* na seção *A instrução try* e a instrução *`raise`* na seção *A instrução raise*.

[1] Essa limitação ocorre porque o código executado por essas operações não está disponível no momento em que o módulo é compilado.

CAPÍTULO 5

O sistema de importação

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

A instrução `import` combina duas operações; ela procura o módulo nomeado e vincula os resultados dessa pesquisa a um nome no escopo local. A operação de busca da instrução `import` é definida como uma chamada para a função `__import__()`, com os argumentos apropriados. O valor de retorno de `__import__()` é usado para executar a operação de ligação de nome da instrução `import`. Veja a instrução `import` para os detalhes exatos da operação de ligação desse nome.

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

Quando uma instrução `import` é executada, a função embutida padrão `__import__()` é chamada. Outros mecanismos para chamar o sistema de importação (como `importlib.import_module()`) podem optar por ignorar `__import__()` e usar suas próprias soluções para implementar a semântica de importação.

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

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

[1] See `types.ModuleType`.

5.1 importlib

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

5.2 Pacotes

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

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

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

Todo módulo tem um nome. Nomes de subpacotes são separados do nome do pacote por um ponto, semelhante à sintaxe de acesso aos atributos padrão do Python. Assim pode ter um pacote chamado `email`, que por sua vez tem um subpacote chamado `email.mime` e um módulo dentro dele chamado `email.mime.text`.

5.2.1 Pacotes regulares

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

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

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

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

5.2.2 Pacotes de espaço de nomes

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

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

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

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

5.3 Caminho de busca

Para iniciar a busca, Python precisa do nome *completo* do módulo (ou pacote, mas para o propósito dessa exposição, não há diferença) que se quer importar. Esse nome vem de vários argumentos passados para o comando *import*, ou dos parâmetros das funções `importlib.import_module()` ou `__import__()`.

Esse nome será usado em várias fases da busca do import, e pode ser um nome com pontos, para um submódulo, ex. `foo.bar.baz`. Nesse caso, Python primeiro tenta importar `foo`, depois `foo.bar`, e finalmente `foo.bar.baz`. Se algum dos imports intermediários falhar, uma exceção `ModuleNotFoundError` é levantada.

5.3.1 Caches de módulos

A primeira checagem durante a busca do import é feita num dicionário chamado `sys.modules`. Esse mapeamento serve como um cache de todos os módulos que já foram importados previamente, incluindo os caminhos intermediários. Se `foo.bar.baz` foi previamente importado, `sys.modules` conterá entradas para `foo`, `foo.bar`, and `foo.bar.baz`. Cada chave terá como valor um objeto módulo correspondente.

Durante o import, o nome do módulo é procurado em `sys.modules` e, se estiver presente, o valor associado é o módulo que satisfaz o import, e o processo termina. Entretanto, se o valor é `None`, uma exceção `ModuleNotFoundError` é levantada. Se o nome do módulo não foi encontrado, Python continuará a busca.

É possível alterar `sys.modules`. Apagar uma chave pode não destruir o objeto módulo associado (outros módulos podem manter referências para ele), mas a entrada do cache será invalidada para o nome daquele módulo, fazendo Python executar nova busca no próximo import. Pode ser atribuído `None` para a chave, forçando que o próximo import do módulo resulte numa exceção `ModuleNotFoundError`.

No entanto, tenha cuidado, pois se você mantiver uma referência para o objeto módulo, invalidar sua entrada de cache em `sys.modules` e, em seguida, reimportar do módulo nomeado, os dois módulo objetos *não* serão os mesmos. Por outro lado, o `importlib.reload()` reutilizará o *mesmo* objeto módulo e simplesmente reinicializará o conteúdo do módulo executando novamente o código do módulo.

5.3.2 Buscadores e carregadores

Se o módulo nomeado não for encontrado em `sys.modules`, então o protocolo de importação do Python é invocado para buscar e carregar o módulo. Este protocolo consiste em dois objetos conceituais, *finders* e *loaders*. O trabalho de um buscador é determinar se ele pode buscar o módulo nomeado usando qualquer estratégia que ele conheça. Objetos que implementam ambas essas interfaces são referenciadas como *importers* - elas retornam a si mesmas, quando elas encontram a aquela eles podem carregar o módulo requisitado.

Python inclui um número de buscadores e carregadores padrões. O primeiro sabe como localizar módulos embutidos, e o segundo sabe como localizar módulos congelados. Um terceiro buscador padrão procura um *import path* por módulos. O *import path* é uma lista de localizações que podem nomear caminhos de sistema de arquivo ou arquivos zip. Ele também pode ser extendido para buscar por qualquer recurso localizável, tais como aqueles identificados por URLs.

O maquinário de importação é extensível, então novos buscadores podem ser adicionados para extender o alcance e o escopo de buscar módulos.

Buscadores na verdade não carregam módulos. Se eles podem encontrar o módulo nomeado, eles retornam um *module spec*, um encapsulamento da informação relacionada a importação do módulo, a qual o maquinário de importação então usa quando o módulo é carregado.

As seguintes seções descrevem o protocolo para buscadores e carregadores em mais detalhes, incluindo como você pode criar e registrar novos para extender o maquinário de importação.

Alterado na versão 3.4: Em versões anteriores do Python, buscadores retornavam *carregadores* diretamente, enquanto hoje eles retornam especificações de um módulo, o qual *contém* carregadores. Carregadores ainda são usados durante a importação, mas possuem menos responsabilidades.

5.3.3 Ganchos de importação

O maquinário de importação é desenhado para ser extensível; o mecanismo primário para isso são os *ganchos de importação*. Existem dois tipos de ganchos de importação: *meta ganchos* e *ganchos de importação de caminho*.

Meta ganchos são chamados no início do processo de importação, antes que qualquer outro processo de importação tenha ocorrido, que não seja busca de cache de `sys.modules`. Isso permite que meta ganchos substituam processamento de `sys.path`, módulos congelados ou mesmo módulos embutidos. Meta ganchos são registrados adicionando novos objetos buscadores a `sys.meta_path`, conforme descrito abaixo.

Ganchos de importação de caminhos são chamados como parte do processamento de `sys.path` (ou `package.__path__`), no ponto onde é encontrado o item do caminho associado. Ganchos de importação de caminho são registrados adicionando novas chamadas para `sys.path_hooks`, conforme descrito abaixo.

5.3.4 O meta caminho

Quando o módulo nomeado não é encontrado em `sys.modules`, Python em seguida busca `sys.meta_path`, o qual contém uma lista de objetos localizador de metacaminho. Esses buscadores são consultados a fim de verificar se eles sabem como manipular o módulo nomeado. Os localizadores de metacaminho devem implementar um método chamado `find_spec()`, o qual recebe tres argumentos: um nome, um caminho de importação, e (opcionalmente) um módulo alvo. O localizador de metacaminho pode usar qualquer estratégia que ele quiser para determinar se ele pode manipular o módulo nomeado ou não.

Se o buscador de meta caminho souber como tratar o módulo nomeado, ele retorna um objeto com especificações. Se ele não puder tratar o módulo nomeado, ele retorna `None`. Se o processamento de `sys.meta_path` alcançar o fim da sua lista sem retornar uma especificação, então `ModuleNotFoundError` é levantado. Qualquer outras exceções levantadas são simplesmente propagadas para cima, abortando o processo de importação.

O método `find_spec()` dos buscadores de meta caminhos é chamado com dois ou tres argumentos. O primeiro é o nome totalmente qualificado do módulo sendo importado, por exemplo `foo.bar.baz`. O segundo argumento é o caminho de entradas para usar para a busca do módulo. Para módulos de alto-nível, o segundo argumento é `None`, mas para sub-módulos ou sub-pacotes, o segundo argumento é o valor do atributo `__path__` do pacote pai. Se o atributo `__path__` apropriado não puder ser acessado, um `ModuleNotFoundError` é levantado. O terceiro

argumento é um objeto módulo existente que será o alvo do carregamento posteriormente. O sistema de importação passa um módulo alvo apenas durante o recarregamento.

O meta caminho pode ser percorrido múltiplas vezes para uma requisição de importação individual. Por exemplo, assumindo que nenhum dos módulos envolvidos já tenha sido cacheado, importar"foo.bar.baz" irá primeiro executar uma importação de alto-nível, chamando `mpf.find_spec("foo", None, None)` em cada buscador de meta caminho (`mpf`). Depois que `foo` foi importado, `foo.bar` será importado percorrendo o meta caminho uma segunda vez, chamando"mpf.find_spec("foo.bar", foo.__path__, None)". Uma vez que `foo.bar` tenha sido importado, a travessia final irá chamar `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)`.

Alguns buscadores de meta caminho apenas suportam importações de alto-nível. Estes importadores sempre retornarão `None` quando qualquer coisa diferente de `None` for passada como o segundo argumento.

O `sys.meta_path` padrão do Python possui tres buscadores de meta caminhos, um que sabe como importar módulos embutidos, um que sabe como importar módulos congelados, e outro que sabe como importar módulos de um *import path* (ex: o *path based finder*).

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

Alterado na versão 3.10: O uso de `find_module()` pelo sistema de importação agora levanta `ImportWarning`.

Alterado na versão 3.12: `find_module()` has been removed. Use `find_spec()` instead.

5.4 Carregando

Se e quando uma especificação de módulo for encontrada, o maquinário de importação irá usá-lo (e o carregador que ele contém) durante o carregamento do módulo. Aqui está uma aproximação do que acontece durante a etapa de carregamento de uma importação:

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

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

Perceba os seguintes detalhes:

- Se houver um objeto módulo existente com o nome fornecido em `sys.modules`, a importação já tera retornado ele.
- O módulo irá existir em `sys.modules` antes do carregador executar o código do módulo. Isso é crucial porque o código do módulo pode (direta ou indiretamente) importar a si mesmo; adicioná-lo a `sys.modules` antecipadamente previne recursão infinita no pior caso e múltiplos carregamentos no melhor caso.
- Se o carregamento falhar, o módulo com falha – e apenas o módulo com falha – é removido de `sys.modules`. Qualquer módulo já presente no cache de `sys.modules`, e qualquer módulo que tenha sido carregado com sucesso como um efeito colateral, deve permanecer no cache. Isso contrasta com recarregamento, onde mesmo o módulo com falha é mantido em `sys.modules`.
- Depois que o módulo é criado, mas antes da execução, o maquinário de importação define os atributos de módulo relacionados a importação ("_init_module_attrs" no exemplo de pseudo-código acima), assim como foi resumido em *uma seção posterior*.
- Execução de módulo é o momento chave do carregamento, no qual o espaço de nomes do módulo é populado. Execução é inteiramente delegada para o carregador, o qual pode decidir o que será populado e como.
- O módulo criado durante o carregamento e passado para exec_module() pode não ser aquele retornado ao final da importação[2].

Alterado na versão 3.4: O sistema de importação tem tomado conta das responsabilidades padrões dos carregadores. Essas responsabilidades eram anteriormente executadas pelo método `importlib.abc.Loader.load_module()`.

5.4.1 Loaders

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

Loaders must satisfy the following requirements:

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

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

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

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

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

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

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

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

- If there is an existing module object with the given name in `sys.modules`, the loader must use that existing module. (Otherwise, `importlib.reload()` will not work correctly.) If the named module does not exist in `sys.modules`, the loader must create a new module object and add it to `sys.modules`.
- The module *must* exist in `sys.modules` before the loader executes the module code, to prevent unbounded recursion or multiple loading.
- If loading fails, the loader must remove any modules it has inserted into `sys.modules`, but it must remove **only** the failing module(s), and only if the loader itself has loaded the module(s) explicitly.

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

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

Alterado na versão 3.10: Use of `load_module()` will raise `ImportWarning`.

5.4.2 Submódulos

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

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

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

```
from .foo import Foo
```

then executing the following puts name bindings for `foo` and `Foo` in the `spam` module:

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

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

5.4.3 Module spec

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

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

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

Novo na versão 3.4.

5.4.4 Import-related module attributes

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

It is **strongly** recommended that you rely on *__spec__* and its attributes instead of any of the other individual attributes listed below.

__name__

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

__loader__

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

It is **strongly** recommended that you rely on *__spec__* instead of this attribute.

Alterado na versão 3.12: The value of __loader__ is expected to be the same as __spec__.loader. The use of __loader__ is deprecated and slated for removal in Python 3.14.

__package__

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

This attribute is used instead of __name__ to calculate explicit relative imports for main modules, as defined in **PEP 366**.

It is **strongly** recommended that you rely on *__spec__* instead of this attribute.

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

Alterado na versão 3.10: ImportWarning is raised if import falls back to __package__ instead of parent.

Alterado na versão 3.12: Raise DeprecationWarning instead of ImportWarning when falling back to __package__.

__spec__

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

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

Novo na versão 3.4.

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

__path__

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

Non-package modules should not have a __path__ attribute.

__file__

__cached__

__file__ is optional (if set, value must be a string). It indicates the pathname of the file from which the module was loaded (if loaded from a file), or the pathname of the shared library file for extension modules loaded dynamically from a shared library. It might be missing for certain types of modules, such as C modules

that are statically linked into the interpreter, and the import system may opt to leave it unset if it has no semantic meaning (e.g. a module loaded from a database).

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

Note that `__cached__` may be set even if `__file__` is not set. However, that scenario is quite atypical. Ultimately, the loader is what makes use of the module spec provided by the finder (from which `__file__` and `__cached__` are derived). So if a loader can load from a cached module but otherwise does not load from a file, that atypical scenario may be appropriate.

It is **strongly** recommended that you rely on *`__spec__`* instead of `__cached__`.

5.4.5 module.__path__

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

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

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

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

5.4.6 Module reprs

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

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

Here are the exact rules used:

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

Alterado na versão 3.12: Use of `module_repr()`, having been deprecated since Python 3.4, was removed in Python 3.12 and is no longer called during the resolution of a module's repr.

5.4.7 Cached bytecode invalidation

Before Python loads cached bytecode from a `.pyc` file, it checks whether the cache is up-to-date with the source `.py` file. By default, Python does this by storing the source's last-modified timestamp and size in the cache file when writing it. At runtime, the import system then validates the cache file by checking the stored metadata in the cache file against the source's metadata.

Python also supports "hash-based" cache files, which store a hash of the source file's contents rather than its metadata. There are two variants of hash-based `.pyc` files: checked and unchecked. For checked hash-based `.pyc` files, Python validates the cache file by hashing the source file and comparing the resulting hash with the hash in the cache file. If a checked hash-based cache file is found to be invalid, Python regenerates it and writes a new checked hash-based cache file. For unchecked hash-based `.pyc` files, Python simply assumes the cache file is valid if it exists. Hash-based `.pyc` files validation behavior may be overridden with the `--check-hash-based-pycs` flag.

Alterado na versão 3.7: Added hash-based `.pyc` files. Previously, Python only supported timestamp-based invalidation of bytecode caches.

5.5 The Path Based Finder

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

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

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

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

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

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

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

5.5.1 Path entry finders

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

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

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

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

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

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

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

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

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

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

5.5.2 Path entry finder protocol

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

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

To indicate to the import machinery that the spec represents a namespace *portion*, the path entry finder sets `submodule_search_locations` to a list containing the portion.

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

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

`find_loader()` takes one argument, the fully qualified name of the module being imported. `find_loader()` returns a 2-tuple where the first item is the loader and the second item is a namespace *portion*.

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

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

Alterado na versão 3.10: Calls to `find_module()` and `find_loader()` by the import system will raise `ImportWarning`.

Alterado na versão 3.12: `find_module()` and `find_loader()` have been removed.

5.6 Replacing the standard import system

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

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

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

5.7 Package Relative Imports

Relative imports use leading dots. A single leading dot indicates a relative import, starting with the current package. Two or more leading dots indicate a relative import to the parent(s) of the current package, one level per dot after the first. For example, given the following package layout:

```
package/
    __init__.py
    subpackage1/
        __init__.py
```

(continua na próxima página)

(continuação da página anterior)

```
        moduleX.py
        moduleY.py
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

In either `subpackage1/moduleX.py` or `subpackage1/__init__.py`, the following are valid relative imports:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
```

Absolute imports may use either the `import <>` or `from <> import <>` syntax, but relative imports may only use the second form; the reason for this is that:

```
import XXX.YYY.ZZZ
```

should expose `XXX.YYY.ZZZ` as a usable expression, but .moduleY is not a valid expression.

5.8 Special considerations for __main__

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

5.8.1 __main__.__spec__

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

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

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

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

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

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

5.9 Referências

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

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

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

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

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

PEP 338 defines executing modules as scripts.

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

CAPÍTULO 6

Expressões

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

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

```
name  ::=  othername
```

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

6.1 Conversões aritméticas

Quando uma descrição de um operador aritmético abaixo usa a frase "os argumentos numéricos são convertidos em um tipo comum", isso significa que a implementação do operador para tipos embutidos funciona da seguinte maneira:

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

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

6.2 Átomos

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

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

6.2.1 Identificadores (Nomes)

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

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

Mangling de nome privado: Quando um identificador que ocorre textualmente em uma definição de classe começa com dois ou mais caracteres de sublinhado e não termina em dois ou mais sublinhados, ele é considerado um *nome privado* dessa classe. Os nomes privados são transformados em um formato mais longo antes que o código seja gerado para eles. A transformação insere o nome da classe, com sublinhados à esquerda removidos e um único sublinhado inserido na frente do nome. Por exemplo, o identificador `__spam` que ocorre em uma classe chamada `Ham` será transformado em `_Ham__spam`. Essa transformação é independente do contexto sintático em que o identificador é usado. Se o nome transformado for extremamente longo (mais de 255 caracteres), poderá ocorrer truncamento definido pela implementação. Se o nome da classe consistir apenas em sublinhados, nenhuma transformação será feita.

6.2.2 Literais

Python oferece suporte a strings e bytes literais e vários literais numéricos:

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

A avaliação de um literal produz um objeto do tipo fornecido (string, bytes, inteiro, número de ponto flutuante, número complexo) com o valor fornecido. O valor pode ser aproximado no caso de ponto flutuante e literais imaginários (complexos). Veja a seção *Literais* para detalhes.

Todos os literais correspondem a tipos de dados imutáveis e, portanto, a identidade do objeto é menos importante que seu valor. Múltiplas avaliações de literais com o mesmo valor (seja a mesma ocorrência no texto do programa ou uma ocorrência diferente) podem obter o mesmo objeto ou um objeto diferente com o mesmo valor.

6.2.3 Formas de parênteses

Um formulário entre parênteses é uma lista de expressões opcional entre parênteses:

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

Uma lista de expressões entre parênteses produz tudo o que aquela lista de expressões produz: se a lista contiver pelo menos uma vírgula, ela produzirá uma tupla; caso contrário, produz a única expressão que compõe a lista de expressões.

Um par de parênteses vazio produz um objeto de tupla vazio. Como as tuplas são imutáveis, aplicam-se as mesmas regras dos literais (isto é, duas ocorrências da tupla vazia podem ou não produzir o mesmo objeto).

Observe que as tuplas não são formadas pelos parênteses, mas sim pelo uso da vírgula. A exceção é a tupla vazia, para a qual os parênteses *são* obrigatórios – permitir "nada" sem parênteses em expressões causaria ambiguidades e permitiria que erros de digitação comuns passassem sem serem detectados.

6.2.4 Sintaxe de criação de listas, conjuntos e dicionários

Para construir uma lista, um conjunto ou um dicionário, o Python fornece uma sintaxe especial chamada "sintaxes de criação" (em inglês, *displays*), cada uma delas em dois tipos:

- o conteúdo do contêiner é listado explicitamente ou
- eles são calculados por meio de um conjunto de instruções de laço e filtragem, chamado de *compreensão*.

Elementos de sintaxe comuns para compreensões são:

```
comprehension  ::=  assignment_expression comp_for
comp_for       ::=  ["async"] "for" target_list "in" or_test [comp_iter]
comp_iter      ::=  comp_for | comp_if
comp_if        ::=  "if" or_test [comp_iter]
```

A compreensão consiste em uma única expressão seguida por pelo menos uma cláusula `for` e zero ou mais cláusulas `for` ou `if`. Neste caso, os elementos do novo contêiner são aqueles que seriam produzidos considerando cada uma das cláusulas `for` ou `if` de um bloco, aninhando da esquerda para a direita, e avaliando a expressão para produzir um elemento cada vez que o bloco mais interno é alcançado.

No entanto, além da expressão iterável na cláusula `for` mais à esquerda, a compreensão é executada em um escopo aninhado implicitamente separado. Isso garante que os nomes atribuídos na lista de destino não "vazem" para o escopo delimitador.

A expressão iterável na cláusula `for` mais à esquerda é avaliada diretamente no escopo envolvente e então passada como um argumento para o escopo aninhado implicitamente. Cláusulas `for` subsequentes e qualquer condição de filtro na cláusula `for` mais à esquerda não podem ser avaliadas no escopo delimitador, pois podem depender dos valores obtidos do iterável mais à esquerda. Por exemplo: `[x*y for x in range(10) for y in range(x, x+10)]`.

Para garantir que a compreensão sempre resulte em um contêiner do tipo apropriado, as expressões `yield` e `yield from` são proibidas no escopo aninhado implicitamente.

Desde o Python 3.6, em uma função *`async def`*, uma cláusula `async for` pode ser usada para iterar sobre um *iterador assíncrono*. Uma compreensão em uma função `async def` pode consistir em uma cláusula `for` ou `async for` seguindo a expressão principal, pode conter `for` ou cláusulas `async for`, e também pode usar expressões *`await`*. Se uma compreensão contém cláusulas `async for` ou expressões `await` ou outras compreensões assíncronas, ela é chamada de *compreensão assíncrona*. Uma compreensão assíncrona pode suspender a execução da função de corrotina na qual ela aparece. Veja também a **PEP 530**.

Novo na versão 3.6: Compreensões assíncronas foram introduzidas.

Alterado na versão 3.8: `yield` e `yield from` proibidos no escopo aninhado implícito.

Alterado na versão 3.11: Compreensões assíncronas agora são permitidas dentro de compreensões em funções assíncronas. As compreensões externas tornam-se implicitamente assíncronas.

6.2.5 Sintaxes de criação de lista

Uma sintaxe de criação de lista é uma série possivelmente vazia de expressões entre colchetes:

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

Uma sintaxe de criação de lista produz um novo objeto de lista, sendo o conteúdo especificado por uma lista de expressões ou uma compreensão. Quando uma lista de expressões separadas por vírgulas é fornecida, seus elementos são avaliados da esquerda para a direita e colocados no objeto de lista nessa ordem. Quando uma compreensão é fornecida, a lista é construída a partir dos elementos resultantes da compreensão.

6.2.6 Sintaxes de criação de conjunto

Uma sintaxe de criação definida é denotada por chaves e distinguível de sintaxes de criação de dicionário pela falta de caractere de dois pontos separando chaves e valores:

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

Uma sintaxe de criação de conjunto produz um novo objeto de conjunto mutável, sendo o conteúdo especificado por uma sequência de expressões ou uma compreensão. Quando uma lista de expressões separadas por vírgula é fornecida, seus elementos são avaliados da esquerda para a direita e adicionados ao objeto definido. Quando uma compreensão é fornecida, o conjunto é construído a partir dos elementos resultantes da compreensão.

Um conjunto vazio não pode ser construído com `{}`; este literal constrói um dicionário vazio.

6.2.7 Sintaxes de criação de dicionário

Uma sintaxe de criação de dicionário é uma série possivelmente vazia de itens de dicionário (pares chave/valor) envolto entre chaves:

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

Uma sintaxe de criação de dicionário produz um novo objeto dicionário.

Se for fornecida uma sequência separada por vírgulas de itens de dicionário, eles são avaliados da esquerda para a direita para definir as entradas do dicionário: cada objeto chave é usado como uma chave no dicionário para armazenar o valor correspondente. Isso significa que você pode especificar a mesma chave várias vezes na lista de itens de dicionário, e o valor final do dicionário para essa chave será o último dado.

Um asterisco duplo `**` denota *desempacotamento do dicionário*. Seu operando deve ser um *mapeamento*. Cada item de mapeamento é adicionado ao novo dicionário. Os valores posteriores substituem os valores já definidos por itens de dicionário anteriores e desempacotamentos de dicionário anteriores.

Novo na versão 3.5: Descompactando em sintaxes de criação de dicionário, originalmente proposto pela **PEP 448**.

Uma compreensão de dict, em contraste com as compreensões de lista e conjunto, precisa de duas expressões separadas por dois pontos, seguidas pelas cláusulas usuais “for” e “if”. Quando a compreensão é executada, os elementos chave e valor resultantes são inseridos no novo dicionário na ordem em que são produzidos.

Restrições nos tipos de valores de chave são listadas anteriormente na seção *A hierarquia de tipos padrão*. (Para resumir, o tipo de chave deve ser *hasheável*, que exclui todos os objetos mutáveis.) Não são detectadas colisões entre chaves duplicadas; o último valor (textualmente mais à direita na sintaxe de criação) armazenado para um determinado valor de chave prevalece.

Alterado na versão 3.8: Antes do Python 3.8, em compreensões de dict, a ordem de avaliação de chave e valor não era bem definida. No CPython, o valor foi avaliado antes da chave. A partir de 3.8, a chave é avaliada antes do valor, conforme proposto pela **PEP 572**.

6.2.8 Expressões geradoras

Uma expressão geradora é uma notação geradora compacta entre parênteses:

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

Uma expressão geradora produz um novo objeto gerador. Sua sintaxe é a mesma das compreensões, exceto pelo fato de estar entre parênteses em vez de colchetes ou chaves.

As variáveis usadas na expressão geradora são avaliadas lentamente quando o método *__next__()* é chamado para o objeto gerador (da mesma forma que os geradores normais). No entanto, a expressão iterável na cláusula `for` mais à esquerda é avaliada imediatamente, de modo que um erro produzido por ela será emitido no ponto em que a expressão do gerador é definida, em vez de no ponto em que o primeiro valor é recuperado. Cláusulas `for` subsequentes e qualquer condição de filtro na cláusula `for` mais à esquerda não podem ser avaliadas no escopo delimitador, pois podem depender dos valores obtidos do iterável mais à esquerda. Por exemplo: `(x*y for x in range(10) for y in range(x, x+10))`.

Os parênteses podem ser omitidos em chamadas com apenas um argumento. Veja a seção *Chamadas* para detalhes.

Para evitar interferir com a operação esperada da própria expressão geradora, as expressões `yield` e `yield from` são proibidas no gerador definido implicitamente.

Se uma expressão geradora contém cláusulas `async for` ou expressões *await*, ela é chamada de *expressão geradora assíncrona*. Uma expressão geradora assíncrona retorna um novo objeto gerador assíncrono, que é um iterador assíncrono (consulte *Iteradores assíncronos*).

Novo na versão 3.6: Expressões geradoras assíncronas foram introduzidas.

Alterado na versão 3.7: Antes do Python 3.7, as expressões geradoras assíncronas só podiam aparecer em corrotinas *async def*. A partir da versão 3.7, qualquer função pode usar expressões geradoras assíncronas.

Alterado na versão 3.8: `yield` e `yield from` proibidos no escopo aninhado implícito.

6.2.9 Expressões yield

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

A expressão yield é usada ao definir uma função *generadora* ou uma função *geradora assíncrona* e, portanto, só pode ser usada no corpo de uma definição de função. Usar uma expressão yield no corpo de uma função faz com que essa função seja uma função geradora, e usá-la no corpo de uma função *async def* faz com que essa função de corrotina seja uma função geradora assíncrona. Por exemplo:

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

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

Devido a seus efeitos colaterais no escopo recipiente, as expressões `yield` não são permitidas como parte dos escopos definidos implicitamente usados para implementar compreensões e expressões geradoras.

Alterado na versão 3.8: Expressões yield proibidas nos escopos aninhados implicitamente usados para implementar compreensões e expressões geradoras.

As funções geradoras são descritas abaixo, enquanto as funções geradoras assíncronas são descritas separadamente na seção *Funções geradoras assíncronas*

Quando uma função geradora é chamada, ela retorna um iterador conhecido como gerador. Esse gerador então controla a execução da função geradora. A execução começa quando um dos métodos do gerador é chamado. Nesse

momento, a execução segue para a primeira expressão yield, onde é suspensa novamente, retornando o valor de *expression_list* ao chamador do gerador, ou None se *expression_list* é omitido. Por suspenso, queremos dizer que todo o estado local é retido, incluindo as chamadas atuais de variáveis locais, o ponteiro de instrução, a pilha de avaliação interna e o estado de qualquer tratamento de exceção. Quando a execução é retomada chamando um dos métodos do gerador, a função pode prosseguir exatamente como se a expressão yield fosse apenas outra chamada externa. O valor da expressão yield após a retomada depende do método que retomou a execução. Se *__next__()* for usado (tipicamente através de uma *for* ou do next() embutido) então o resultado será None. Caso contrário, se *send()* for usado, o resultado será o valor passado para esse método.

Tudo isso torna as funções geradoras bastante semelhantes às corrotinas; cedem múltiplas vezes, possuem mais de um ponto de entrada e sua execução pode ser suspensa. A única diferença é que uma função geradora não pode controlar onde a execução deve continuar após o seu rendimento; o controle é sempre transferido para o chamador do gerador.

Expressões yield são permitidas em qualquer lugar em uma construção *try*. Se o gerador não for retomado antes de ser finalizado (ao atingir uma contagem de referências zero ou ao ser coletado como lixo), o método *close()* do iterador de gerador será chamado, permitindo que quaisquer cláusulas *finally* pendentes sejam executadas.

Quando yield from <expr> é usado, a expressão fornecida deve ser iterável. Os valores produzidos pela iteração desse iterável são passados diretamente para o chamador dos métodos do gerador atual. Quaisquer valores passados com *send()* e quaisquer exceções passadas com *throw()* são passados para o iterador subjacente se ele tiver os métodos apropriados. Se este não for o caso, então *send()* irá levantar AttributeError ou TypeError, enquanto *throw()* irá apenas levantar a exceção passada imediatamente.

Quando o iterador subjacente estiver completo, o atributo value da instância StopIteration gerada torna-se o valor da expressão yield. Ele pode ser definido explicitamente ao levantar StopIteration ou automaticamente quando o subiterador é um gerador (retornando um valor do subgerador).

Alterado na versão 3.3: Adicionado yield from <expr> para delegar o fluxo de controle a um subiterador.

Os parênteses podem ser omitidos quando a expressão yield é a única expressão no lado direito de uma instrução de atribuição.

Ver também:

PEP 255 - Geradores simples
A proposta para adicionar geradores e a instrução *yield* ao Python.

PEP 342 - Corrotinas via Geradores Aprimorados
A proposta de aprimorar a API e a sintaxe dos geradores, tornando-os utilizáveis como simples corrotinas.

PEP 380 - Sintaxe para Delegar a um Subgerador
A proposta de introduzir a sintaxe *yield_from*, facilitando a delegação a subgeradores.

PEP 525 - Geradores assíncronos
A proposta que se expandiu em **PEP 492** adicionando recursos de gerador a funções de corrotina.

Métodos de iterador gerador

Esta subseção descreve os métodos de um iterador gerador. Eles podem ser usados para controlar a execução de uma função geradora.

Observe que chamar qualquer um dos métodos do gerador abaixo quando o gerador já estiver em execução levanta uma exceção ValueError.

generator.**__next__**()

Inicia a execução de uma função geradora ou a retoma na última expressão yield executada. Quando uma função geradora é retomada com um método *__next__()*, a expressão yield atual sempre é avaliada como None. A execução então continua para a próxima expressão yield, onde o gerador é suspenso novamente, e o valor de *expression_list* é retornado para o chamador de *__next__()*. Se o gerador sair sem produzir outro valor, uma exceção StopIteration será levantada.

Este método é normalmente chamado implicitamente, por exemplo por um laço *for*, ou pela função embutida next().

generator.**send**(*value*)

Retoma a execução e "envia" um valor para a função geradora. O argumento *value* torna-se o resultado da expressão yield atual. O método *send()* retorna o próximo valor gerado pelo gerador, ou levanta StopIteration se o gerador sair sem produzir outro valor. Quando *send()* é chamado para iniciar o gerador, ele deve ser chamado com None como argumento, porque não há nenhuma expressão yield que possa receber o valor.

generator.**throw**(*value*)

generator.**throw**(*type*[, *value*[, *traceback*]])

Levanta uma exceção no ponto em que o gerador foi pausado e retorna o próximo valor gerado pela função geradora. Se o gerador sair sem gerar outro valor, uma exceção StopIteration será levantada. Se a função geradora não detectar a exceção passada ou levanta uma exceção diferente, essa exceção se propagará para o chamador.

Em uso típico, isso é chamado com uma única instância de exceção semelhante à forma como a palavra reservada *raise* é usada.

Para compatibilidade com versões anteriores, no entanto, a segunda assinatura é suportada, seguindo uma convenção de versões mais antigas do Python. O argumento *type* deve ser uma classe de exceção e *value* deve ser uma instância de exceção. Se o *valor* não for fornecido, o construtor *tipo* será chamado para obter uma instância. Se *traceback* for fornecido, ele será definido na exceção, caso contrário, qualquer atributo __traceback__ existente armazenado em *value* poderá ser limpo.

Alterado na versão 3.12: A segunda assinatura (tipo[, valor[, traceback]]) foi descontinuada e pode ser removida em uma versão futura do Python.

generator.**close**()

Levanta GeneratorExit no ponto onde a função geradora foi pausada. Se a função geradora sair normalmente, já estiver fechada ou levantar GeneratorExit (por não capturar a exceção), "close" retornará ao seu chamador. Se o gerador produzir um valor, um RuntimeError é levantado. Se o gerador levantar qualquer outra exceção, ela será propagada para o chamador. *close()* não faz nada se o gerador já saiu devido a uma exceção ou saída normal.

Exemplos

Aqui está um exemplo simples que demonstra o comportamento de geradores e funções geradoras:

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

Para exemplos usando `yield from`, consulte a pep-380 em "O que há de novo no Python."

Funções geradoras assíncronas

A presença de uma expressão yield em uma função ou método definido usando a *`async def`* define ainda mais a função como uma função *geradora assíncrona*.

Quando uma função geradora assíncrona é chamada, ela retorna um iterador assíncrono conhecido como objeto gerador assíncrono. Esse objeto controla a execução da função geradora. Um objeto gerador assíncrono é normalmente usado em uma instrução *`async for`* em uma função de corrotina de forma análoga a como um objeto gerador seria usado em uma instrução *`for`*.

A chamada de um dos métodos do gerador assíncrono retorna um objeto *aguardável*, e a execução começa quando esse objeto é aguardado. Nesse momento, a execução prossegue até a primeira expressão yield, onde é suspensa novamente, retornando o valor de *`expression_list`* para a corrotina em aguardo. Assim como ocorre com um gerador, a suspensão significa que todo o estado local é mantido, inclusive as ligações atuais das variáveis locais, o ponteiro de instruções, a pilha de avaliação interna e o estado de qualquer tratamento de exceção. Quando a execução é retomada, aguardando o próximo objeto retornado pelos métodos do gerador assíncrono, a função pode prosseguir exatamente como se a expressão de rendimento fosse apenas outra chamada externa. O valor da expressão yield após a retomada depende do método que retomou a execução. Se *`__anext__()`* for usado, o resultado será `None`. Caso contrário, se *`asend()`* for usado, o resultado será o valor passado para esse método.

Se um gerador assíncrono encerrar mais cedo por *`break`*, pela tarefa que fez sua chamada ser cancelada ou por outras exceções, o código de limpeza assíncrona do gerador será executado e possivelmente levantará alguma exceção ou acessará as variáveis de contexto em um contexto inesperado – talvez após o tempo de vida das tarefas das quais ele depende, ou durante o laço de eventos de encerramento quando o gancho de coleta de lixo do gerador assíncrono for chamado. Para prevenir isso, o chamador deve encerrar explicitamente o gerador assíncrono chamando o método *`aclose()`* para finalizar o gerador e, por fim, desconectá-lo do laço de eventos.

Em uma função geradora assíncrona, expressões de yield são permitidas em qualquer lugar em uma construção *`try`*. No entanto, se um gerador assíncrono não for retomado antes de ser finalizado (alcançando uma contagem de referência zero ou sendo coletado pelo coletor de lixo), então uma expressão de yield dentro de um construção `try` pode resultar em uma falha na execução das cláusulas pendentes de *`finally`*. Nesse caso, é responsabilidade do laço de eventos ou escalonador que executa o gerador assíncrono chamar o método *`aclose()`* do gerador iterador assíncrono e executar o objeto corrotina resultante, permitindo assim que quaisquer cláusulas pendentes de `finally` sejam executadas.

Para cuidar da finalização após o término do laço de eventos, um laço de eventos deve definir uma função *finalizer* que recebe um gerador assíncrono e provavelmente chama *`aclose()`* e executa a corrotina. Este *finalizer* pode ser registrado chamando `sys.set_asyncgen_hooks()`. Quando iterado pela primeira vez, um gerador assíncrono armazenará o *finalizer* registrado para ser chamado na finalização. Para um exemplo de referência de um método *finalizer*, consulte a implementação de `asyncio.Loop.shutdown_asyncgens` em Lib/asyncio/base_events.py.

O expressão `yield from <expr>` é um erro de sintaxe quando usado em uma função geradora assíncrona.

Métodos geradores-iteradores assíncronos

Esta subseção descreve os métodos de um iterador gerador assíncrono, que são usados para controlar a execução de uma função geradora.

coroutine agen.**__anext__**()

Retorna um objeto aguardável que, quando executado, começa a executar o gerador assíncrono ou o retoma na última expressão yield executada. Quando uma função geradora assíncrona é retomada com o método *`__anext__()`*, a expressão yield atual sempre avalia para `None` no objeto aguardável retornado, que, quando executado, continuará para a próxima expressão yield. O valor de *`expression_list`* da expressão yield é o valor da exceção `StopIteration` levantada pela corrotina em conclusão. Se o gerador assíncrono sair sem produzir outro valor, o objeto aguardável em vez disso levanta uma exceção `StopAsyncIteration`, sinalizando que a iteração assíncrona foi concluída.

Este método é normalmente chamado implicitamente por um laço *`async for`*.

coroutine agen.**asend**(*value*)

Retorna um objeto aguardável que, quando executado, retoma a execução do gerador assíncrono. Assim como o método *send()* para um gerador, isso "envia" um valor para a função geradora assíncrona, e o argumento *value* se torna o resultado da expressão de yield atual. O objeto aguardável retornado pelo método *asend()* retornará o próximo valor produzido pelo gerador como o valor da exceção `StopIteration` levantada, ou lança `StopAsyncIteration` se o gerador assíncrono sair sem produzir outro valor. Quando *asend()* é chamado para iniciar o gerador assíncrono, ele deve ser chamado com `None` como argumento, pois não há expressão yield que possa receber o valor.

coroutine agen.**athrow**(*value*)

coroutine agen.**athrow**(*type*[, *value*[, *traceback*]])

Retorna um objeto aguardável que gera uma exceção do tipo `type` no ponto em que o gerador assíncrono foi pausado, e retorna o próximo valor produzido pela função geradora como o valor da exceção `StopIteration` levantada. Se o gerador assíncrono terminar sem produzir outro valor, uma exceção `StopAsyncIteration` é levantada pelo objeto aguardável. Se a função geradora não capturar a exceção passada ou gerar uma exceção diferente, então quando o objeto aguardável for executado, essa exceção se propagará para o chamador do objeto aguardável.

Alterado na versão 3.12: A segunda assinatura (tipo[, valor[, traceback]]) foi descontinuada e pode ser removida em uma versão futura do Python.

coroutine agen.**aclose**()

Retorna um objeto aguardável que, quando executado, levantará uma `GeneratorExit` na função geradora assíncrona no ponto em que foi pausada. Se a função geradora assíncrona sair de forma normal, se estiver já estiver fechada ou levantar `GeneratorExit` (não capturando a exceção), então o objeto aguardável retornado levantará uma exceção `StopIteration`. Quaisquer outros objetos aguardáveis retornados por chamadas subsequentes à função geradora assíncrona levantarão uma exceção `StopAsyncIteration`. Se a função geradora assíncrona levantar um valor, um `RuntimeError` será lançado pelo objeto aguardável. Se a função geradora assíncrona levantar qualquer outra exceção, ela será propagada para o chamador do objeto aguardável. Se a função geradora assíncrona já tiver saído devido a uma exceção ou saída normal, então chamadas posteriores ao método *aclose()* retornarão um objeto aguardável que não faz nada.

6.3 Primárias

Primárias representam as operações mais fortemente vinculadas da linguagem. Sua sintaxe é:

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

6.3.1 Referências de atributo

Uma referência de atributo é um primário seguido de um ponto e um nome.

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

A primária deve avaliar para um objeto de um tipo que tem suporte a referências de atributo, o que a maioria dos objetos faz. Este objeto é então solicitado a produzir o atributo cujo nome é o identificador. O tipo e o valor produzido são determinados pelo objeto. Várias avaliações da mesma referência de atributo podem produzir diferentes objetos.

Esta produção pode ser personalizada substituindo o método *__getattribute__()* ou o método *__getattr__()*. O método `__getattribute__()` é chamado primeiro e retorna um valor ou levanta uma `AttributeError` se o atributo não estiver disponível.

Se for levantada uma `AttributeError` e o objeto tiver um método `__getattr__()`, esse método será chamado como alternativa.

6.3.2 Subscrições

A subscrição de uma instância de uma classe de *classe de contêiner* geralmente selecionará um elemento do contêiner. A subscrição de uma *classe genérica* geralmente retornará um objeto GenericAlias.

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

When an object is subscripted, the interpreter will evaluate the primary and the expression list.

The primary must evaluate to an object that supports subscription. An object may support subscription through defining one or both of `__getitem__()` and `__class_getitem__()`. When the primary is subscripted, the evaluated result of the expression list will be passed to one of these methods. For more details on when `__class_getitem__` is called instead of `__getitem__`, see *__class_getitem__ versus __getitem__*.

If the expression list contains at least one comma, it will evaluate to a `tuple` containing the items of the expression list. Otherwise, the expression list will evaluate to the value of the list's sole member.

For built-in objects, there are two types of objects that support subscription via `__getitem__()`:

1. Mappings. If the primary is a *mapping*, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the subscription selects the value in the mapping that corresponds to that key. An example of a builtin mapping class is the `dict` class.
2. Sequences. If the primary is a *sequence*, the expression list must evaluate to an `int` or a `slice` (as discussed in the following section). Examples of builtin sequence classes include the `str`, `list` and `tuple` classes.

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

A `string` is a special kind of sequence whose items are *characters*. A character is not a separate data type but a string of exactly one character.

6.3.3 Fatiamentos

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

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

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

The semantics for a slicing are as follows. The primary is indexed (using the same `__getitem__()` method as normal subscription) with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of a proper slice is a slice object (see section *A hierarquia de tipos padrão*) whose `start`, `stop` and `step` attributes are the

values of the expressions given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

6.3.4 Chamadas

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

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

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

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

If keyword arguments are present, they are first converted to positional arguments, as follows. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot (if the identifier is the same as the first formal parameter name, the first slot is used, and so on). If the slot is already filled, a `TypeError` exception is raised. Otherwise, the argument is placed in the slot, filling it (even if the expression is `None`, it fills the slot). When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value from the function definition. (Default values are calculated, once, when the function is defined; thus, a mutable object such as a list or dictionary used as default value will be shared by all calls that don't specify an argument value for the corresponding slot; this should usually be avoided.) If there are any unfilled slots for which no default value is specified, a `TypeError` exception is raised. Otherwise, the list of filled slots is used as the argument list for the call.

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

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

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

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

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

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

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

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

When `**expression` is used, each key in this mapping must be a string. Each value from the mapping is assigned to the first formal parameter eligible for keyword assignment whose name is equal to the key. A key need not be a Python identifier (e.g. `"max-temp °F"` is acceptable, although it will not match any formal parameter that could be declared). If there is no match to a formal parameter the key-value pair is collected by the `**` parameter, if there is one, or if there is not, a `TypeError` exception is raised.

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

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

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

Se for—

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

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

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

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

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

6.4 Expressão await

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

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

Novo na versão 3.5.

6.5 O operador de potência

O operador de potência vincula-se com mais força do que os operadores unários à sua esquerda; ele se vincula com menos força do que os operadores unários à sua direita. A sintaxe é:

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

Assim, em uma sequência sem parênteses de operadores de potência e unários, os operadores são avaliados da direita para a esquerda (isso não restringe a ordem de avaliação dos operandos): `-1**2` resulta em `-1` .

O operador de potência tem a mesma semântica que a função embutida `pow()`, quando chamado com dois argumentos: ele produz seu argumento esquerdo elevado à potência de seu argumento direito. Os argumentos numéricos são primeiro convertidos em um tipo comum e o resultado é desse tipo.

For int operands, the result has the same type as the operands unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`.

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

This operation can be customized using the special *__pow__()* method.

6.6 Unary arithmetic and bitwise operations

All unary arithmetic and bitwise operations have the same priority:

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

The unary - (minus) operator yields the negation of its numeric argument; the operation can be overridden with the *__neg__()* special method.

The unary + (plus) operator yields its numeric argument unchanged; the operation can be overridden with the *__pos__()* special method.

The unary ~ (invert) operator yields the bitwise inversion of its integer argument. The bitwise inversion of `x` is defined as `-(x+1)`. It only applies to integral numbers or to custom objects that override the *__invert__()* special method.

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

6.7 Binary arithmetic operations

As operações aritméticas binárias possuem os níveis de prioridade convencionais. Observe que algumas dessas operações também se aplicam a determinados tipos não numéricos. Além do operador potência, existem apenas dois níveis, um para operadores multiplicativos e outro para operadores aditivos:

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

O operador `*` (multiplicação) produz o produto de seus argumentos. Os argumentos devem ser números ou um argumento deve ser um número inteiro e o outro deve ser uma sequência. No primeiro caso, os números são convertidos para um tipo comum e depois multiplicados. Neste último caso, é realizada a repetição da sequência; um fator de repetição negativo produz uma sequência vazia.

This operation can be customized using the special *__mul__()* and *__rmul__()* methods.

O operador `@` (arroba) deve ser usado para multiplicação de matrizes. Nenhum tipo embutido do Python implementa este operador.

Novo na versão 3.5.

Os operadores `/` (divisão) e `//` (divisão pelo piso) produzem o quociente de seus argumentos. Os argumentos numéricos são primeiro convertidos em um tipo comum. A divisão de inteiros produz um ponto flutuante, enquanto a divisão pelo piso de inteiros resulta em um inteiro; o resultado é o da divisão matemática com a função 'floor' aplicada ao resultado. A divisão por zero levanta a exceção `ZeroDivisionError`.

This operation can be customized using the special *__truediv__()* and *__floordiv__()* methods.

O operador `%` (módulo) produz o restante da divisão do primeiro argumento pelo segundo. Os argumentos numéricos são primeiro convertidos em um tipo comum. Um argumento zero à direita levanta a exceção `ZeroDivisionError`. Os argumentos podem ser números de ponto flutuante, por exemplo, `3.14%0.7` é igual a `0.34` (já que `3.14` é igual a `4*0.7 + 0.34`.) O operador módulo sempre produz um resultado com o mesmo sinal do seu segundo operando (ou zero); o valor absoluto do resultado é estritamente menor que o valor absoluto do segundo operando[1].

Os operadores de divisão pelo piso e módulo são conectados pela seguinte identidade: `x == (x//y)*y + (x%y)`. A divisão pelo piso e o módulo também estão conectados com a função embutida `divmod()`: `divmod(x, y) == (x//y, x%y)`.[2].

Além de realizar a operação de módulo em números, o operador `%` também é sobrecarregado por objetos string para realizar a formatação de string no estilo antigo (também conhecida como interpolação). A sintaxe para formatação de string é descrita na Referência da Biblioteca Python, seção old-string-formatting.

The *modulo* operation can be customized using the special *__mod__()* method.

O operador de divisão pelo piso, o operador de módulo e a função `divmod()` não são definidos para números complexos. Em vez disso, converta para um número de ponto flutuante usando a função `abs()` se apropriado.

O operador + (adição) produz a soma de seus argumentos. Os argumentos devem ser números ou sequências do mesmo tipo. No primeiro caso, os números são convertidos para um tipo comum e depois somados. Neste último caso, as sequências são concatenadas.

This operation can be customized using the special *__add__()* and *__radd__()* methods.

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

[2] If x is very close to an exact integer multiple of y, it's possible for `x//y` to be one larger than `(x-x%y)//y` due to rounding. In such cases, Python returns the latter result, in order to preserve that `divmod(x,y)[0] * y + x % y` be very close to `x`.

O operador − (subtração) produz a diferença de seus argumentos. Os argumentos numéricos são primeiro convertidos em um tipo comum.

This operation can be customized using the special *__sub__()* method.

6.8 Shifting operations

The shifting operations have lower priority than the arithmetic operations:

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

Esses operadores aceitam números inteiros como argumentos. Eles deslocam o primeiro argumento para a esquerda ou para a direita pelo número de bits fornecido pelo segundo argumento.

This operation can be customized using the special *__lshift__()* and *__rshift__()* methods.

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

6.9 Operações binárias bit a bit

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

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

The `&` operator yields the bitwise AND of its arguments, which must be integers or one of them must be a custom object overriding *__and__()* or *__rand__()* special methods.

The ^ operator yields the bitwise XOR (exclusive OR) of its arguments, which must be integers or one of them must be a custom object overriding *__xor__()* or *__rxor__()* special methods.

The `|` operator yields the bitwise (inclusive) OR of its arguments, which must be integers or one of them must be a custom object overriding *__or__()* or *__ror__()* special methods.

6.10 Comparações

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

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

Comparisons yield boolean values: `True` or `False`. Custom *rich comparison methods* may return non-boolean values. In this case Python will call `bool()` on such value in boolean contexts.

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

Formalmente, se *a*, *b*, *c*, …, *y*, *z* são expressões e *op1*, *op2*, …, *opN* são operadores de comparação, então `a op1 b`

`op2 c ... y opN z` é equivalente a `a op1 b e b op2 c e ... y opN z`, exceto que cada expressão é avaliada no máximo uma vez.

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

6.10.1 Comparações de valor

Os operadores <, >, ==, >=, <= e `!=` comparam os valores de dois objetos. Os objetos não precisam ser do mesmo tipo.

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

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

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

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

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

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

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

  The not-a-number values `float('NaN')` and `decimal.Decimal('NaN')` are special. Any ordered comparison of a number to a not-a-number value is false. A counter-intuitive implication is that not-a-number values are not equal to themselves. For example, if `x = float('NaN')`, `3 < x`, `x < 3` and `x == x` are all false, while `x != x` is true. This behavior is compliant with IEEE 754.

- `None` and `NotImplemented` are singletons. **PEP 8** advises that comparisons for singletons should always be done with `is` or `is not`, never the equality operators.

- Binary sequences (instances of `bytes` or `bytearray`) can be compared within and across their types. They compare lexicographically using the numeric values of their elements.

- Strings (instances of `str`) compare lexicographically using the numerical Unicode code points (the result of the built-in function `ord()`) of their characters.[3]

  Strings and binary sequences cannot be directly compared.

[3] The Unicode standard distinguishes between *code points* (e.g. U+0041) and *abstract characters* (e.g. "LATIN CAPITAL LETTER A"). While most abstract characters in Unicode are only represented using one code point, there is a number of abstract characters that can in addition be represented using a sequence of more than one code point. For example, the abstract character "LATIN CAPITAL LETTER C WITH CEDILLA" can be represented as a single *precomposed character* at code position U+00C7, or as a sequence of a *base character* at code position U+0043 (LATIN CAPITAL LETTER C), followed by a *combining character* at code position U+0327 (COMBINING CEDILLA).

The comparison operators on strings compare at the level of Unicode code points. This may be counter-intuitive to humans. For example, `"\u00C7" == "\u0043\u0327"` is `False`, even though both strings represent the same abstract character "LATIN CAPITAL LETTER C WITH CEDILLA".

To compare strings at the level of abstract characters (that is, in a way intuitive to humans), use `unicodedata.normalize()`.

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

  Sequences compare lexicographically using comparison of corresponding elements. The built-in containers typically assume identical objects are equal to themselves. That lets them bypass equality tests for identical objects to improve performance and to maintain their internal invariants.

  Lexicographical comparison between built-in collections works as follows:

  – For two collections to compare equal, they must be of the same type, have the same length, and each pair of corresponding elements must compare equal (for example, `[1,2] == (1,2)` is false because the type is not the same).

  – Collections that support order comparison are ordered the same as their first unequal elements (for example, `[1,2,x] <= [1,2,y]` has the same value as `x <= y`). If a corresponding element does not exist, the shorter collection is ordered first (for example, `[1,2] < [1,2,3]` is true).

- Mappings (instances of `dict`) compare equal if and only if they have equal `(key, value)` pairs. Equality comparison of the keys and values enforces reflexivity.

  Order comparisons (<, >, <=, and >=) raise `TypeError`.

- Sets (instances of `set` or `frozenset`) can be compared within and across their types.

  They define order comparison operators to mean subset and superset tests. Those relations do not define total orderings (for example, the two sets `{1,2}` and `{2,3}` are not equal, nor subsets of one another, nor supersets of one another). Accordingly, sets are not appropriate arguments for functions which depend on total ordering (for example, `min()`, `max()`, and `sorted()` produce undefined results given a list of sets as inputs).

  Comparison of sets enforces reflexivity of its elements.

- Most other built-in types have no comparison methods implemented, so they inherit the default comparison behavior.

User-defined classes that customize their comparison behavior should follow some consistency rules, if possible:

- Equality comparison should be reflexive. In other words, identical objects should compare equal:

  `x is y` implies `x == y`

- Comparison should be symmetric. In other words, the following expressions should have the same result:

  `x == y` and `y == x`

  `x != y` and `y != x`

  `x < y` and `y > x`

  `x <= y` and `y >= x`

- Comparison should be transitive. The following (non-exhaustive) examples illustrate that:

  `x > y and y > z` implies `x > z`

  `x < y and y <= z` implies `x < z`

- Inverse comparison should result in the boolean negation. In other words, the following expressions should have the same result:

  `x == y` and `not x != y`

  `x < y` and `not x >= y` (for total ordering)

  `x > y` and `not x <= y` (for total ordering)

  The last two expressions apply to totally ordered collections (e.g. to sequences, but not to sets or mappings). See also the `total_ordering()` decorator.

- The `hash()` result should be consistent with equality. Objects that are equal should either have the same hash value, or be marked as unhashable.

Python does not enforce these consistency rules. In fact, the not-a-number values are an example for not following these rules.

6.10.2 Membership test operations

The operators *in* and *not in* test for membership. `x in s` evaluates to `True` if *x* is a member of *s*, and `False` otherwise. `x not in s` returns the negation of `x in s`. All built-in sequences and set types support this as well as dictionary, for which `in` tests whether the dictionary has a given key. For container types such as list, tuple, set, frozenset, dict, or collections.deque, the expression `x in y` is equivalent to `any(x is e or x == e for e in y)`.

For the string and bytes types, `x in y` is `True` if and only if *x* is a substring of *y*. An equivalent test is `y.find(x) != -1`. Empty strings are always considered to be a substring of any other string, so `"" in "abc"` will return `True`.

For user-defined classes which define the *__contains__()* method, `x in y` returns `True` if `y.__contains__(x)` returns a true value, and `False` otherwise.

For user-defined classes which do not define *__contains__()* but do define *__iter__()*, `x in y` is `True` if some value `z`, for which the expression `x is z or x == z` is true, is produced while iterating over `y`. If an exception is raised during the iteration, it is as if *in* raised that exception.

Lastly, the old-style iteration protocol is tried: if a class defines *__getitem__()*, `x in y` is `True` if and only if there is a non-negative integer index *i* such that `x is y[i] or x == y[i]`, and no lower integer index raises the `IndexError` exception. (If any other exception is raised, it is as if *in* raised that exception).

The operator *not in* is defined to have the inverse truth value of *in*.

6.10.3 Comparações de identidade

The operators *is* and *is not* test for an object's identity: `x is y` is true if and only if *x* and *y* are the same object. An Object's identity is determined using the `id()` function. `x is not y` yields the inverse truth value.[4]

6.11 Operações booleanas

```
or_test   ::=  and_test | or_test "or" and_test
and_test  ::=  not_test | and_test "and" not_test
not_test  ::=  comparison | "not" not_test
```

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. User-defined objects can customize their truth value by providing a *__bool__()* method.

The operator *not* yields `True` if its argument is false, `False` otherwise.

The expression `x and y` first evaluates *x*; if *x* is false, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

The expression `x or y` first evaluates *x*; if *x* is true, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

Note that neither *and* nor *or* restrict the value and type they return to `False` and `True`, but rather return the last evaluated argument. This is sometimes useful, e.g., if `s` is a string that should be replaced by a default value if it is empty, the expression `s or 'foo'` yields the desired value. Because *not* has to create a new value, it returns a boolean value regardless of the type of its argument (for example, `not 'foo'` produces `False` rather than `''`.)

[4] Due to automatic garbage-collection, free lists, and the dynamic nature of descriptors, you may notice seemingly unusual behaviour in certain uses of the *is* operator, like those involving comparisons between instance methods, or constants. Check their documentation for more info.

6.12 Expressões de atribuição

```
assignment_expression  ::=  [identifier ":="] expression
```

An assignment expression (sometimes also called a "named expression" or "walrus") assigns an *expression* to an *identifier*, while also returning the value of the *expression*.

One common use case is when handling matched regular expressions:

```
if matching := pattern.search(data):
    do_something(matching)
```

Or, when processing a file stream in chunks:

```
while chunk := file.read(9000):
    process(chunk)
```

Assignment expressions must be surrounded by parentheses when used as expression statements and when used as sub-expressions in slicing, conditional, lambda, keyword-argument, and comprehension-if expressions and in `assert`, `with`, and `assignment` statements. In all other places where they can be used, parentheses are not required, including in `if` and `while` statements.

Novo na versão 3.8: See **PEP 572** for more details about assignment expressions.

6.13 Expressões condicionais

```
conditional_expression  ::=  or_test ["if" or_test "else" expression]
expression              ::=  conditional_expression | lambda_expr
```

Conditional expressions (sometimes called a "ternary operator") have the lowest priority of all Python operations.

The expression `x if C else y` first evaluates the condition, *C* rather than *x*. If *C* is true, *x* is evaluated and its value is returned; otherwise, *y* is evaluated and its value is returned.

See **PEP 308** for more details about conditional expressions.

6.14 Lambdas

```
lambda_expr  ::=  "lambda" [parameter_list] ":" expression
```

Lambda expressions (sometimes called lambda forms) are used to create anonymous functions. The expression `lambda parameters: expression` yields a function object. The unnamed object behaves like a function object defined with:

```
def <lambda>(parameters):
    return expression
```

See section *Definições de função* for the syntax of parameter lists. Note that functions created with lambda expressions cannot contain statements or annotations.

6.15 Listas de expressões

```
expression_list      ::=   expression ("," expression)* [","]
starred_list         ::=   starred_item ("," starred_item)* [","]
starred_expression   ::=   expression | (starred_item ",")* [starred_item]
starred_item         ::=   assignment_expression | "*" or_expr
```

Except when part of a list or set display, an expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

An asterisk `*` denotes *iterable unpacking*. Its operand must be an *iterable*. The iterable is expanded into a sequence of items, which are included in the new tuple, list, or set, at the site of the unpacking.

Novo na versão 3.5: Iterable unpacking in expression lists, originally proposed by **PEP 448**.

A trailing comma is required only to create a one-item tuple, such as `1,`; it is optional in all other cases. A single expression without a trailing comma doesn't create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: `()`.)

6.16 Ordem de avaliação

Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

In the following lines, expressions will be evaluated in the arithmetic order of their suffixes:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```

6.17 Precedência de operadores

The following table summarizes the operator precedence in Python, from highest precedence (most binding) to lowest precedence (least binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation and conditional expressions, which group from right to left).

Note that comparisons, membership tests, and identity tests, all have the same precedence and have a left-to-right chaining feature as described in the *Comparações* section.

Operador	Descrição
`(expressions...)`, `[expressions...]`, `{key: value...}`, `{expressions...}`	Binding or parenthesized expression, list display, dictionary display, set display
`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	Subscription, slicing, call, attribute reference
`await x`	Expressão await
`**`	Exponenciação[5]
`+x`, `-x`, `~x`	Positive, negative, bitwise NOT
`*`, `@`, `/`, `//`, `%`	Multiplication, matrix multiplication, division, floor division, remainder[6]
`+`, `-`	Addition and subtraction
`<<`, `>>`	Shifts
`&`	E (AND) bit a bit
`^`	OU EXCLUSIVO (XOR) bit a bit
`|`	OR bit a bit
`in`, *`not in`*, *`is`*, *`is not`*, `<`, `<=`, `>`, `>=`, `!=`, `==`	Comparisons, including membership tests and identity tests
`not x`	Booleano NEGAÇÃO (NOT)
`and`	Booleano E (AND)
`or`	Booleano OU (OR)
`if` – `else`	Expressão condicional
`lambda`	Expressão lambda
`:=`	Expressão de atribuição

[5] The power operator `**` binds less tightly than an arithmetic or bitwise unary operator on its right, that is, `2**-1` is `0.5`.
[6] The `%` operator is also used for string formatting; the same precedence applies.

CAPÍTULO 7

Instruções simples

Uma instrução simples consiste uma única linha lógica. Várias instruções simples podem ocorrer em uma única linha separada por ponto e vírgula. A sintaxe para instruções simples é:

```
simple_stmt  ::=  expression_stmt
                | assert_stmt
                | assignment_stmt
                | augmented_assignment_stmt
                | annotated_assignment_stmt
                | pass_stmt
                | del_stmt
                | return_stmt
                | yield_stmt
                | raise_stmt
                | break_stmt
                | continue_stmt
                | import_stmt
                | future_stmt
                | global_stmt
                | nonlocal_stmt
                | type_stmt
```

7.1 Instruções de expressão

As instruções de expressão são usadas (principalmente interativamente) para calcular e escrever um valor, ou (geralmente) para chamar um procedimento (uma função que não retorna nenhum resultado significativo; em Python, os procedimentos retornam o valor `None`). Outros usos de instruções de expressão são permitidos e ocasionalmente úteis. A sintaxe para uma instrução de expressão é:

```
expression_stmt  ::=  starred_expression
```

Uma instrução de expressão avalia a lista de expressões (que pode ser uma única expressão).

No modo interativo, se o valor não for `None`, ele será convertido em uma string usando a função embutida `repr()`

e a string resultante será gravada na saída padrão em uma linha sozinha (exceto se o resultado é `None`, de modo que as chamadas de procedimento não causam nenhuma saída.)

7.2 Instruções de atribuição

As instruções de atribuição são usadas para (re)vincular nomes a valores e modificar atributos ou itens de objetos mutáveis:

```
assignment_stmt  ::=  (target_list "=")+ (starred_expression | yield_expression)
target_list      ::=  target ("," target)* [","]
target           ::=  identifier
                      | "(" [target_list] ")"
                      | "[" [target_list] "]"
                      | attributeref
                      | subscription
                      | slicing
                      | "*" target
```

(Veja a seção *Primárias* para as definições de sintaxe de *attributeref*, *subscription* e *slicing*.)

Uma instrução de atribuição avalia a lista de expressões (lembre-se de que pode ser uma única expressão ou uma lista separada por vírgulas, a última produzindo uma tupla) e atribui o único objeto resultante a cada uma das listas alvos, da esquerda para a direita.

A atribuição é definida recursivamente dependendo da forma do alvo (lista). Quando um alvo faz parte de um objeto mutável (uma referência de atributo, assinatura ou divisão), o objeto mutável deve, em última análise, executar a atribuição e decidir sobre sua validade e pode levantar uma exceção se a atribuição for inaceitável. As regras observadas pelos vários tipos e as exceções levantadas são dadas com a definição dos tipos de objetos (ver seção *A hierarquia de tipos padrão*).

A atribuição de um objeto a uma lista alvo, opcionalmente entre parênteses ou colchetes, é definida recursivamente da maneira a seguir.

- Se a lista alvo contiver um único alvo sem vírgula à direita, opcionalmente entre parênteses, o objeto será atribuído a esse alvo.
- Senão:
  - Se a lista alvo contiver um alvo prefixado com um asterisco, chamado de alvo "com estrela" *(starred)*: o objeto deve ser um iterável com pelo menos tantos itens quantos os alvos na lista alvo, menos um. Os primeiros itens do iterável são atribuídos, da esquerda para a direita, aos alvos antes do alvo com estrela. Os itens finais do iterável são atribuídos aos alvos após o alvo com estrela. Uma lista dos itens restantes no iterável é então atribuída ao alvo com estrela (a lista pode estar vazia).
  - Senão: o objeto deve ser um iterável com o mesmo número de itens que existem alvos na lista alvos, e os itens são atribuídos, da esquerda para a direita, aos alvos correspondentes.

A atribuição de um objeto a um único alvo é definida recursivamente da maneira a seguir.

- Se o alvo for um identificador (nome):
  - Se o nome não ocorrer em uma instrução `global` ou `nonlocal` no bloco de código atual: o nome está vinculado ao objeto no espaço de nomes local atual.
  - Caso contrário: o nome é vinculado ao objeto no espaço de nomes global global ou no espaço de nomes global externo determinado por `nonlocal`, respectivamente.

  O nome é vinculado novamente se já estiver vinculado. Isso pode fazer com que a contagem de referências para o objeto anteriormente vinculado ao nome chegue a zero, fazendo com que o objeto seja desalocado e seu destrutor (se houver) seja chamado.

- Se o alvo for uma referência de atributo: a expressão primária na referência é avaliada. Deve produzir um objeto com atributos atribuíveis; se este não for o caso, a exceção `TypeError` é levanta. Esse objeto é então solicitado a atribuir o objeto atribuído ao atributo fornecido; se não puder executar a atribuição, ele levanta uma exceção (geralmente, mas não necessariamente `AttributeError`).

  Nota: Se o objeto for uma instância de classe e a referência de atributo ocorrer em ambos os lados do operador de atribuição, a expressão do lado direito, `a.x` pode acessar um atributo de instância ou (se não existir nenhum atributo de instância) uma classe atributo. O alvo do lado esquerdo `a.x` é sempre definido como um atributo de instância, criando-o se necessário. Assim, as duas ocorrências de `a.x` não necessariamente se referem ao mesmo atributo: se a expressão do lado direito se refere a um atributo de classe, o lado esquerdo cria um novo atributo de instância como alvo da atribuição:

  ```
  class Cls:
      x = 3             # class variable
  inst = Cls()
  inst.x = inst.x + 1   # writes inst.x as 4 leaving Cls.x as 3
  ```

  Esta descrição não se aplica necessariamente aos atributos do descritor, como propriedades criadas com `property()`.

- Se o alvo for uma assinatura: a expressão primária na referência é avaliada. Deve produzir um objeto de sequência mutável (como uma lista) ou um objeto de mapeamento (como um dicionário). Em seguida, a expressão subscrito é avaliada.

  Se o primário for um objeto de sequência mutável (como uma lista), o subscrito deverá produzir um inteiro. Se for negativo, o comprimento da sequência é adicionado a ela. O valor resultante deve ser um inteiro não negativo menor que o comprimento da sequência, e a sequência é solicitada a atribuir o objeto atribuído ao seu item com esse índice. Se o índice estiver fora do intervalo, a exceção `IndexError` será levantada (a atribuição a uma sequência subscrita não pode adicionar novos itens a uma lista).

  Se o primário for um objeto de mapeamento (como um dicionário), o subscrito deve ter um tipo compatível com o tipo de chave do mapeamento, e o mapeamento é solicitado a criar um par chave/valore que mapeia o subscrito para o objeto atribuído. Isso pode substituir um par de chave/valor existente pelo mesmo valor de chave ou inserir um novo par de chave/valor (se não existir nenhuma chave com o mesmo valor).

  For user-defined objects, the *`__setitem__()`* method is called with appropriate arguments.

- Se o alvo for um fatiamento: a expressão primária na referência é avaliada. Deve produzir um objeto de sequência mutável (como uma lista). O objeto atribuído deve ser um objeto de sequência do mesmo tipo. Em seguida, as expressões de limite inferior e superior são avaliadas, na medida em que estiverem presentes; os padrões são zero e o comprimento da sequência. Os limites devem ser avaliados como inteiros. Se um dos limites for negativo, o comprimento da sequência será adicionado a ele. Os limites resultantes são cortados para ficarem entre zero e o comprimento da sequência, inclusive. Finalmente, o objeto de sequência é solicitado a substituir a fatia pelos itens da sequência atribuída. O comprimento da fatia pode ser diferente do comprimento da sequência atribuída, alterando assim o comprimento da sequência alvo, se a sequência alvo permitir.

Detalhes da implementação do CPython: Na implementação atual, a sintaxe dos alvos é considerada a mesma das expressões e a sintaxe inválida é rejeitada durante a fase de geração do código, causando mensagens de erro menos detalhadas.

Embora a definição de atribuição implique que as sobreposições entre o lado esquerdo e o lado direito sejam "simultâneas" (por exemplo, `a, b = b, a` troca duas variáveis), sobreposições *dentro* da coleção de variáveis atribuídas ocorrem da esquerda para a direita, às vezes resultando em confusão. Por exemplo, o programa a seguir imprime `[0, 2]`:

```
x = [0, 1]
i = 0
i, x[i] = 1, 2          # i is updated, then x[i] is updated
print(x)
```

Ver também:

PEP 3132 - Descompactação de Iterável Estendida
A especificação para o recurso `*target`.

7.2.1 Instruções de atribuição aumentada

A atribuição aumentada é a combinação, em uma única instrução, de uma operação binária e uma instrução de atribuição:

```
augmented_assignment_stmt  ::=  augtarget augop (expression_list | yield_expression)
augtarget                  ::=  identifier | attributeref | subscription | slicing
augop                      ::=  "+=" | "-=" | "*=" | "@=" | "/=" | "//=" | "%=" | "**="
                                | ">>=" | "<<=" | "&=" | "^=" | "|="
```

(Veja a seção *Primárias* para as definições de sintaxe dos últimos três símbolos.)

Uma atribuição aumentada avalia o alvo (que, diferentemente das instruções de atribuição normais, não pode ser um descompactação) e a lista de expressões, executa a operação binária específica para o tipo de atribuição nos dois operandos e atribui o resultado ao alvo original. O alvo é avaliado apenas uma vez.

Uma expressão de atribuição aumentada como `x += 1` pode ser reescrita como `x = x + 1` para obter um efeito semelhante, mas não exatamente igual. Na versão aumentada, `x` é avaliado apenas uma vez. Além disso, quando possível, a operação real é executada *no local*, o que significa que, em vez de criar um novo objeto e atribuí-lo ao alvo, o objeto antigo é modificado.

Ao contrário das atribuições normais, as atribuições aumentadas avaliam o lado esquerdo *antes* de avaliar o lado direito. Por exemplo, `a[i] += f(x)` primeiro procura `a[i]`, então avalia `f(x)` e executa a adição e, por último, escreve o resultado de volta para `a[i]`.

Com exceção da atribuição a tuplas e vários alvos em uma única instrução, a atribuição feita por instruções de atribuição aumentada é tratada da mesma maneira que atribuições normais. Da mesma forma, com exceção do possível comportamento *in-place*, a operação binária executada por atribuição aumentada é a mesma que as operações binárias normais.

Para alvos que são referências de atributos, a mesma *advertência sobre atributos de classe e instância* se aplica a atribuições regulares.

7.2.2 instruções de atribuição anotado

A atribuição de *anotação* é a combinação, em uma única instrução, de uma anotação de variável ou atributo e uma instrução de atribuição opcional:

```
annotated_assignment_stmt  ::=  augtarget ":" expression
                                ["=" (starred_expression | yield_expression)]
```

A diferença para as *Instruções de atribuição* normal é que apenas um único alvo é permitido.

Para nomes simples como alvos de atribuição, se no escopo de classe ou módulo, as anotações são avaliadas e armazenadas em uma classe especial ou atributo de módulo `__annotations__` que é um mapeamento de dicionário de nomes de variáveis (desconfigurados se privados) para anotações avaliadas. Este atributo é gravável e é criado automaticamente no início da execução do corpo da classe ou módulo, se as anotações forem encontradas estaticamente.

Para expressões como alvos de atribuição, as anotações são avaliadas se estiverem no escopo da classe ou do módulo, mas não armazenadas.

Se um nome for anotado em um escopo de função, esse nome será local para esse escopo. As anotações nunca são avaliadas e armazenadas em escopos de função.

If the right hand side is present, an annotated assignment performs the actual assignment before evaluating annotations (where applicable). If the right hand side is not present for an expression target, then the interpreter evaluates the target except for the last *__setitem__()* or *__setattr__()* call.

Ver também:

PEP 526 - Sintaxe para Anotações de Variáveis
A proposta que adicionou sintaxe para anotar os tipos de variáveis (incluindo variáveis de classe e variáveis de instância), em vez de expressá-las por meio de comentários.

PEP 484 - Dicas de tipo
A proposta que adicionou o módulo `typing` para fornecer uma sintaxe padrão para anotações de tipo que podem ser usadas em ferramentas de análise estática e IDEs.

Alterado na versão 3.8: Agora, as atribuições anotadas permitem as mesmas expressões no lado direito que as atribuições regulares. Anteriormente, algumas expressões (como expressões de tupla sem parênteses) causavam um erro de sintaxe.

7.3 A instrução `assert`

As instruções assert são uma maneira conveniente de inserir asserções de depuração em um programa:

```
assert_stmt  ::=  "assert" expression ["," expression]
```

A forma simples, `assert expression`, é equivalente a

```
if __debug__:
    if not expression: raise AssertionError
```

A forma estendida, `assert expression1, expression2`, é equivalente a

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

Essas equivalências assumem que `__debug__` e `AssertionError` referem-se às variáveis embutidas com esses nomes. Na implementação atual, a variável embutida `__debug__` é `True` em circunstâncias normais, `False` quando a otimização é solicitada (opção de linha de comando `-O`). O gerador de código atual não emite código para uma instrução assert quando a otimização é solicitada em tempo de compilação. Observe que não é necessário incluir o código-fonte da expressão que falhou na mensagem de erro; ele será exibido como parte do stack trace (situação da pilha de execução).

Atribuições a `__debug__` são ilegais. O valor da variável embutida é determinado quando o interpretador é iniciado.

7.4 A instrução `pass`

```
pass_stmt  ::=  "pass"
```

`pass` é uma operação nula — quando é executada, nada acontece. É útil como um espaço reservado quando uma instrução é necessária sintaticamente, mas nenhum código precisa ser executado, por exemplo:

```
def f(arg): pass    # a function that does nothing (yet)

class C: pass       # a class with no methods (yet)
```

7.5 A instrução `del`

```
del_stmt  ::=  "del" target_list
```

A exclusão é definida recursivamente de maneira muito semelhante à maneira como a atribuição é definida. Em vez de explicar em detalhes, aqui estão algumas dicas.

A exclusão de uma lista alvo exclui recursivamente cada alvo, da esquerda para a direita.

A exclusão de um nome remove a ligação desse nome do espaço de nomes global local ou global, dependendo se o nome ocorre em uma instrução *global* no mesmo bloco de código. Se o nome for desvinculado, uma exceção `NameError` será levantada.

A exclusão de referências de atributos, assinaturas e fatias é passada para o objeto principal envolvido; a exclusão de um fatiamento é em geral equivalente à atribuição de uma fatia vazia do tipo certo (mas mesmo isso é determinado pelo objeto fatiado).

Alterado na versão 3.2: Anteriormente, era ilegal excluir um nome do espaço de nomes local se ele ocorresse como uma variável livre em um bloco aninhado.

7.6 A instrução `return`

```
return_stmt  ::=  "return" [expression_list]
```

return só pode ocorrer sintaticamente aninhado em uma definição de função, não em uma definição de classe aninhada.

Se uma lista de expressões estiver presente, ela será avaliada, caso contrário, `None` será substituído.

return deixa a chamada da função atual com a lista de expressões (ou `None`) como valor de retorno.

Quando *return* passa o controle de uma instrução *try* com uma cláusula *finally*, essa cláusula `finally` é executada antes de realmente sair da função.

Em uma função geradora, a instrução *return* indica que o gerador está pronto e fará com que `StopIteration` seja gerado. O valor retornado (se houver) é usado como argumento para construir `StopIteration` e se torna o atributo `StopIteration.value`.

Em uma função de gerador assíncrono, uma instrução *return* vazia indica que o gerador assíncrono está pronto e fará com que `StopAsyncIteration` seja gerado. Uma instrução `return` não vazia é um erro de sintaxe em uma função de gerador assíncrono.

7.7 A instrução `yield`

```
yield_stmt  ::=  yield_expression
```

Uma instrução *yield* é semanticamente equivalente a uma *expressão yield*. A instrução yield pode ser usada para omitir os parênteses que, de outra forma, seriam necessários na instrução de expressão yield equivalente. Por exemplo, as instruções yield

```
yield <expr>
yield from <expr>
```

são equivalentes às instruções de expressão yield

```
(yield <expr>)
(yield from <expr>)
```

Expressões e instruções yield são usadas apenas ao definir uma função *geradora* e são usadas apenas no corpo da função geradora. Usar yield em uma definição de função é suficiente para fazer com que essa definição crie uma função geradora em vez de uma função normal.

Para detalhes completos da semântica `yield`, consulte a seção *Expressões yield*.

7.8 A instrução `raise`

```
raise_stmt  ::=  "raise" [expression ["from" expression]]
```

Se nenhuma expressão estiver presente, `raise` reativa a exceção que está sendo tratada no momento, que também é conhecida como *exceção ativa*. Se não houver uma exceção ativa no momento, uma exceção `RuntimeError` é levantada indicando que isso é um erro.

Caso contrário, `raise` avalia a primeira expressão como o objeto de exceção. Deve ser uma subclasse ou uma instância de `BaseException`. Se for uma classe, a instância de exceção será obtida quando necessário instanciando a classe sem argumentos.

O *tipo* da exceção é a classe da instância de exceção, o *valor* é a própria instância.

A traceback object is normally created automatically when an exception is raised and attached to it as the `__traceback__` attribute. You can create an exception and set your own traceback in one step using the `with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so:

```
raise Exception("foo occurred").with_traceback(tracebackobj)
```

The `from` clause is used for exception chaining: if given, the second *expression* must be another exception class or instance. If the second expression is an exception instance, it will be attached to the raised exception as the `__cause__` attribute (which is writable). If the expression is an exception class, the class will be instantiated and the resulting exception instance will be attached to the raised exception as the `__cause__` attribute. If the raised exception is not handled, both exceptions will be printed:

```
>>> try:
...     print(1 / 0)
... except Exception as exc:
...     raise RuntimeError("Something bad happened") from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("Something bad happened") from exc
RuntimeError: Something bad happened
```

A similar mechanism works implicitly if a new exception is raised when an exception is already being handled. An exception may be handled when an `except` or `finally` clause, or a `with` statement, is used. The previous exception is then attached as the new exception's `__context__` attribute:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
```

(continua na próxima página)

(continuação da página anterior)

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("Something bad happened")
RuntimeError: Something bad happened
```

Exception chaining can be explicitly suppressed by specifying `None` in the `from` clause:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Informações adicionais sobre exceções podem ser encontradas na seção *Exceções*, e informações sobre como lidar com exceções estão na seção *A instrução try*.

Alterado na versão 3.3: `None` agora é permitido como `Y` em `raise X from Y`.

Novo na versão 3.3: The `__suppress_context__` attribute to suppress automatic display of the exception context.

Alterado na versão 3.11: Se o traceback da exceção ativa for modificado em uma cláusula `except`, uma instrução `raise` subsequente levantará novamente a exceção com o traceback modificado. Anteriormente, a exceção era levantada novamente com o traceback que tinha quando foi capturada.

7.9 A instrução `break`

```
break_stmt  ::=  "break"
```

`break` só pode ocorrer sintaticamente aninhado em um laço `for` ou `while`, mas não aninhado em uma função ou definição de classe dentro desse laço.

Ele termina o laço de fechamento mais próximo, pulando a cláusula opcional `else` se o laço tiver uma.

Se um laço `for` é encerrado por `break`, o alvo de controle do laço mantém seu valor atual.

Quando `break` passa o controle de uma instrução `try` com uma cláusula `finally`, essa cláusula `finally` é executada antes de realmente sair do laço.

7.10 A instrução `continue`

```
continue_stmt   ::=   "continue"
```

continue só pode ocorrer sintaticamente aninhado em um laço *for* ou *while*, mas não aninhado em uma função ou definição de classe dentro desse laço. Ele continua com o próximo ciclo do laço de fechamento mais próximo.

Quando *continue* passa o controle de uma instrução *try* com uma cláusula *finally*, essa cláusula `finally` é executada antes realmente iniciar o próximo ciclo do laço.

7.11 A instrução `import`

```
import_stmt      ::=   "import" module ["as" identifier] ("," module ["as" identifier])*
                       | "from" relative_module "import" identifier ["as" identifier]
                       ("," identifier ["as" identifier])*
                       | "from" relative_module "import" "(" identifier ["as" identifier
                       ("," identifier ["as" identifier])* [","] ")"
                       | "from" relative_module "import" "*"
module           ::=   (identifier ".")* identifier
relative_module  ::=   "."* module | "."+
```

A instrução de importação básica (sem cláusula *from*) é executada em duas etapas:

1. encontra um módulo, carregando e inicializando-o se necessário
2. define um nome ou nomes no espaço de nomes local para o escopo onde ocorre a instrução *import*.

Quando a instrução contém várias cláusulas (separadas por vírgulas), as duas etapas são executadas separadamente para cada cláusula, como se as cláusulas tivessem sido separadas em instruções de importação individuais.

Os detalhes da primeira etapa, encontrar e carregar módulos, estão descritos com mais detalhes na seção sobre o *sistema de importação*, que também descreve os vários tipos de pacotes e módulos que podem ser importados, bem como todos os os ganchos que podem ser usados para personalizar o sistema de importação. Observe que falhas nesta etapa podem indicar que o módulo não pôde ser localizado *ou* que ocorreu um erro durante a inicialização do módulo, o que inclui a execução do código do módulo.

Se o módulo solicitado for recuperado com sucesso, ele será disponibilizado no espaço de nomes local de três maneiras:

- Se o nome do módulo é seguido pela palavra reservada `as`, o nome a seguir é vinculado diretamente ao módulo importado.
- Se nenhum outro nome for especificado e o módulo que está sendo importado for um módulo de nível superior, o nome do módulo será vinculado ao espaço de nomes local como uma referência ao módulo importado
- Se o módulo que está sendo importado *não* for um módulo de nível superior, o nome do pacote de nível superior que contém o módulo será vinculado ao espaço de nomes local como uma referência ao pacote de nível superior. O módulo importado deve ser acessado usando seu nome completo e não diretamente

O formulário *from* usa um processo um pouco mais complexo:

1. encontra o módulo especificado na cláusula *from*, carregando e inicializando-o se necessário;
2. para cada um dos identificadores especificados nas cláusulas *import*:
    1. verifica se o módulo importado tem um atributo com esse nome
    2. caso contrário, tenta importar um submódulo com esse nome e verifica o módulo importado novamente para esse atributo
    3. se o atributo não for encontrado, a exceção `ImportError` é levantada.

4. caso contrário, uma referência a esse valor é armazenada no espaço de nomes local, usando o nome na cláusula `as` se estiver presente, caso contrário, usando o nome do atributo

Exemplos:

```
import foo                 # foo imported and bound locally
import foo.bar.baz         # foo, foo.bar, and foo.bar.baz imported, foo bound␣
↪locally
import foo.bar.baz as fbb  # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz␣
↪bound as fbb
from foo.bar import baz    # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz␣
↪bound as baz
from foo import attr       # foo imported and foo.attr bound as attr
```

Se a lista de identificadores for substituída por uma estrela (`'*'`), todos os nomes públicos definidos no módulo serão vinculados ao espaço de nomes local para o escopo onde ocorre a instrução *import*.

Os *nomes públicos* definidos por um módulo são determinados verificando o espaço de nomes do módulo para uma variável chamada `__all__`; se definido, deve ser uma sequência de strings que são nomes definidos ou importados por esse módulo. Os nomes dados em `__all__` são todos considerados públicos e devem existir. Se `__all__` não estiver definido, o conjunto de nomes públicos inclui todos os nomes encontrados no espaço de nomes do módulo que não começam com um caractere sublinhado (`'_'`). `__all__` deve conter toda a API pública. Destina-se a evitar a exportação acidental de itens que não fazem parte da API (como módulos de biblioteca que foram importados e usados no módulo).

A forma curinga de importação — `from module import *` — só é permitida no nível do módulo. Tentar usá-lo em definições de classe ou função irá levantar uma `SyntaxError`.

Ao especificar qual módulo importar, você não precisa especificar o nome absoluto do módulo. Quando um módulo ou pacote está contido em outro pacote, é possível fazer uma importação relativa dentro do mesmo pacote superior sem precisar mencionar o nome do pacote. Usando pontos iniciais no módulo ou pacote especificado após *from* você pode especificar quão alto percorrer a hierarquia de pacotes atual sem especificar nomes exatos. Um ponto inicial significa o pacote atual onde o módulo que faz a importação existe. Dois pontos significam um nível de pacote acima. Três pontos são dois níveis acima, etc. Então, se você executar `from . import mod` de um módulo no pacote `pkg` então você acabará importando o `pkg.mod`. Se você executar `from ..subpkg2 import mod` de dentro de `pkg.subpkg1` você irá importar `pkg.subpkg2.mod`. A especificação para importações relativas está contida na seção *Package Relative Imports*.

`importlib.import_module()` é fornecida para dar suporte a aplicações que determinam dinamicamente os módulos a serem carregados.

Levanta um evento de auditoria `import` com argumentos `module`, `filename`, `sys.path`, `sys.meta_path`, `sys.path_hooks`.

7.11.1 Instruções future

Uma *instrução future* é uma diretiva para o compilador de que um determinado módulo deve ser compilado usando sintaxe ou semântica que estará disponível em uma versão futura especificada do Python, onde o recurso se tornará padrão.

A instrução future destina-se a facilitar a migração para versões futuras do Python que introduzem alterações incompatíveis na linguagem. Ele permite o uso dos novos recursos por módulo antes do lançamento em que o recurso se torna padrão.

```
future_stmt   ::=   "from" "__future__" "import" feature ["as" identifier]
                    ("," feature ["as" identifier])*
                    | "from" "__future__" "import" "(" feature ["as" identifier]
                    ("," feature ["as" identifier])* [","] ")"
feature       ::=   identifier
```

Uma instrução future deve aparecer perto do topo do módulo. As únicas linhas que podem aparecer antes de uma instrução future são:

- o módulo docstring (se houver),
- omentários,
- linhas vazias e
- outras instruções future.

O único recurso que requer o uso da instrução future é `annotations` (veja **PEP 563**).

Todos os recursos históricos habilitados pela instrução future ainda são reconhecidos pelo Python 3. A lista inclui `absolute_import`, `division`, `generators`, `generator_stop`, `unicode_literals`, `print_function`, `nested_scopes` e `with_statement`. Eles são todos redundantes porque estão sempre habilitados e mantidos apenas para compatibilidade com versões anteriores.

Uma instrução future é reconhecida e tratada especialmente em tempo de compilação: as alterações na semântica das construções principais são frequentemente implementadas gerando código diferente. Pode até ser o caso de um novo recurso introduzir uma nova sintaxe incompatível (como uma nova palavra reservada), caso em que o compilador pode precisar analisar o módulo de maneira diferente. Tais decisões não podem ser adiadas até o tempo de execução.

Para qualquer versão, o compilador sabe quais nomes de recursos foram definidos e levanta um erro em tempo de compilação se uma instrução future contiver um recurso desconhecido.

A semântica do tempo de execução direto é a mesma de qualquer instrução de importação: existe um módulo padrão `__future__`, descrito posteriormente, e será importado da maneira usual no momento em que a instrução future for executada.

A semântica interessante do tempo de execução depende do recurso específico ativado pela instrução future.

Observe que não há nada de especial sobre a instrução:

```
import __future__ [as name]
```

Essa não é uma instrução future; é uma instrução de importação comum sem nenhuma semântica especial ou restrições de sintaxe.

Code compiled by calls to the built-in functions `exec()` and `compile()` that occur in a module `M` containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can be controlled by optional arguments to `compile()` — see the documentation of that function for details.

Uma instrução future tipada digitada em um prompt do interpretador interativo terá efeito no restante da sessão do interpretador. Se um interpretador for iniciado com a opção `-i`, for passado um nome de script para ser executado e o script incluir uma instrução future, ela entrará em vigor na sessão interativa iniciada após a execução do script.

Ver também:

PEP 236 - De volta ao __future__
A proposta original para o mecanismo do __future__.

7.12 A instrução `global`

```
global_stmt  ::=  "global" identifier ("," identifier)*
```

A instrução *`global`* é uma declaração que vale para todo o bloco de código atual. Isso significa que os identificadores listados devem ser interpretados como globais. Seria impossível atribuir a uma variável global sem `global`, embora variáveis livres possam se referir a globais sem serem declaradas globais.

Nomes listados em uma instrução *`global`* não devem ser usados no mesmo bloco de código que precede textualmente a instrução `global`.

Os nomes listados em uma instrução *`global`* não devem ser definidos como parâmetros formais, ou como alvos em instruções *`with`* ou cláusulas *`except`*, ou em uma lista alvo *`for`*, definição de *`class`*, definição de função,

instrução *import* ou anotação de variável.

Detalhes da implementação do CPython: A implementação atual não impõe algumas dessas restrições, mas os programas não devem abusar dessa liberdade, pois implementações future podem aplicá-las ou alterar silenciosamente o significado do programa.

Nota do programador: *global* é uma diretiva para o analisador sintático. Aplica-se apenas ao código analisado ao mesmo tempo que a instrução `global`. Em particular, uma instrução `global` contida em uma string ou objeto código fornecido à função embutida `exec()` não afeta o bloco de código *contendo* a chamada da função e o código contido em tal uma string não é afetada por instruções `global` no código que contém a chamada da função. O mesmo se aplica às funções `eval()` e `compile()`.

7.13 A instrução `nonlocal`

```
nonlocal_stmt  ::=  "nonlocal" identifier ("," identifier)*
```

A instrução *nonlocal* faz com que os identificadores listados se refiram a variáveis vinculadas anteriormente no escopo mais próximo, excluindo globais. Isso é importante porque o comportamento padrão para ligação é pesquisar primeiro o espaço de nomes local. A instrução permite que o código encapsulado ligue novamente variáveis fora do escopo local além do escopo global (módulo).

Os nomes listados em uma instrução *nonlocal*, diferentemente daqueles listados em uma instrução *global*, devem se referir a associações preexistentes em um escopo delimitador (o escopo no qual uma nova associação deve ser criada não pode ser determinado inequivocamente).

Os nomes listados em uma instrução *nonlocal* não devem colidir com ligações preexistentes no escopo local.

Ver também:

PEP 3104 - Acesso a nomes em escopos externos
A especificação para a instrução *nonlocal*.

7.14 A instrução `type`

```
type_stmt  ::=  'type' identifier [type_params] "=" expression
```

A instrução `type` declara um apelido de tipo, que é uma instância de `typing.TypeAliasType`.

Por exemplo, a instrução a seguir cria um apelido de tipo:

```
type Point = tuple[float, float]
```

Este código é aproximadamente equivalente a:

```
annotation-def VALUE_OF_Point():
    return tuple[float, float]
Point = typing.TypeAliasType("Point", VALUE_OF_Point())
```

`annotation-def` indica um *escopo de anotação*, que se comporta principalmente como uma função, mas com diversas pequenas diferenças.

O valor do apelido de tipo é avaliado no escopo de anotação. Ele não é avaliado quando o apelido de tipo é criado, mas somente quando o valor é acessado através do atributo `__value__` do apelido de tipo (veja *Avaliação preguiçosa*). Isso permite que o apelido de tipo se refira a nomes que ainda não estão definidos.

Apelidos de tipo podem se tornar genéricos adicionando uma *lista de parâmetros de tipo* após o nome. Veja *Generic type aliases* para mais.

`type` é uma *palavra reservada contextual*.

Novo na versão 3.12.

Ver também:

PEP 695 - Sintaxe de parâmetros de tipo
Introduziu a instrução `type` e sintaxe para classes e funções genéricas.

CAPÍTULO 8

Instruções compostas

Instruções compostas contém (grupos de) outras instruções; Elas afetam ou controlam a execução dessas outras instruções de alguma maneira. Em geral, instruções compostas abrangem múltiplas linhas, no entanto em algumas manifestações simples uma instrução composta inteira pode estar contida em uma linha.

As instruções *if*, *while* e *for* implementam construções tradicionais de controle do fluxo de execução. *try* especifica tratadores de exceção e/ou código de limpeza para uma instrução ou grupo de instruções, enquanto a palavra reservada *with* permite a execução de código de inicialização e finalização em volta de um bloco de código. Definições de função e classe também são sintaticamente instruções compostas.

Uma instrução composta consiste em uma ou mais "cláusulas". Uma cláusula consiste em um cabeçalho e um "conjunto". Os cabeçalhos das cláusulas de uma instrução composta específica estão todos no mesmo nível de indentação. Cada cabeçalho de cláusula começa com uma palavra reservada de identificação exclusiva e termina com dois pontos. Um conjunto é um grupo de instruções controladas por uma cláusula. Um conjunto pode ser uma ou mais instruções simples separadas por ponto e vírgula na mesma linha do cabeçalho, após os dois pontos do cabeçalho, ou pode ser uma ou mais instruções indentadas nas linhas subsequentes. Somente a última forma de conjunto pode conter instruções compostas aninhadas; o seguinte é ilegal, principalmente porque não ficaria claro a qual cláusula *if* a seguinte cláusula *else* pertenceria:

```
if test1: if test2: print(x)
```

Observe também que o ponto e vírgula é mais vinculado que os dois pontos neste contexto, de modo que no exemplo a seguir, todas ou nenhuma das chamadas `print()` são executadas:

```
if x < y < z: print(x); print(y); print(z)
```

Resumindo:

```
compound_stmt   ::=   if_stmt
                    | while_stmt
                    | for_stmt
                    | try_stmt
                    | with_stmt
                    | match_stmt
                    | funcdef
                    | classdef
                    | async_with_stmt
                    | async_for_stmt
```

```
                    | async_funcdef
suite           ::=   stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement       ::=   stmt_list NEWLINE | compound_stmt
stmt_list       ::=   simple_stmt (";" simple_stmt)* [";"]
```

Note que instruções sempre terminam em uma `NEWLINE` possivelmente seguida por uma `DEDENT`. Note também que cláusulas de continuação sempre começam com uma palavra reservada que não pode iniciar uma instrução, desta forma não há ambiguidades (o problema do "*else* pendurado" é resolvido em Python obrigando que instruções *if* aninhadas tenham indentação)

A formatação das regras de gramática nas próximas seções põe cada cláusula em uma linha separada para as tornar mais claras.

8.1 A instrução `if`

A instrução *if* é usada para execução condicional:

```
if_stmt  ::=   "if" assignment_expression ":" suite
               ("elif" assignment_expression ":" suite)*
               ["else" ":" suite]
```

Ele seleciona exatamente um dos conjuntos avaliando as expressões uma por uma até que uma seja considerada verdadeira (veja a seção *Operações booleanas* para a definição de verdadeiro e falso); então esse conjunto é executado (e nenhuma outra parte da instrução *if* é executada ou avaliada). Se todas as expressões forem falsas, o conjunto da cláusula *else*, se presente, é executado.

8.2 A instrução `while`

A instrução *while* é usada para execução repetida desde que uma expressão seja verdadeira:

```
while_stmt  ::=   "while" assignment_expression ":" suite
                  ["else" ":" suite]
```

Isto testa repetidamente a expressão e, se for verdadeira, executa o primeiro conjunto; se a expressão for falsa (o que pode ser a primeira vez que ela é testada) o conjunto da cláusula `else`, se presente, é executado e o laço termina.

Uma instrução *break* executada no primeiro conjunto termina o loop sem executar o conjunto da cláusula `else`. Uma instrução *continue* executada no primeiro conjunto ignora o resto do conjunto e volta a testar a expressão.

8.3 A instrução `for`

A instrução *for* é usada para iterar sobre os elementos de uma sequência (como uma string, tupla ou lista) ou outro objeto iterável:

```
for_stmt  ::=   "for" target_list "in" starred_list ":" suite
                ["else" ":" suite]
```

A expressão `starred_list` é avaliada uma vez; deve produzir um objeto *iterável*. Um *iterador* é criado para esse iterável. O primeiro item fornecido pelo iterador é então atribuído à lista de alvos usando as regras padrão para atribuições (veja *Instruções de atribuição*), e o conjunto é executado. Isso se repete para cada item fornecido pelo iterador. Quando o iterador se esgota, o conjunto na cláusula `else`, se presente, é executado e o loop termina.

Uma instrução *break* executada no primeiro conjunto termina o loop sem executar o conjunto da cláusula `else`. Uma instrução *continue* executada no primeiro conjunto pula o resto do conjunto e continua com o próximo item, ou com a cláusula `else` se não houver próximo item.

O laço for faz atribuições às variáveis na lista de destino. Isso substitui todas as atribuições anteriores a essas variáveis, incluindo aquelas feitas no conjunto do laço for:

```
for i in range(10):
    print(i)
    i = 5             # this will not affect the for-loop
                      # because i will be overwritten with the next
                      # index in the range
```

Os nomes na lista de destinos não são excluídos quando o laço termina, mas se a sequência estiver vazia, eles não serão atribuídos pelo laço. Dica: o tipo embutido `range()` representa sequências aritméticas imutáveis de inteiros. Por exemplo, iterar `range(3)` sucessivamente produz 0, 1 e depois 2.

Alterado na versão 3.11: Elementos marcados com estrela agora são permitidos na lista de expressões.

8.4 A instrução `try`

A instrução `try` especifica manipuladores de exceção e/ou código de limpeza para um grupo de instruções:

```
try_stmt   ::=  try1_stmt | try2_stmt | try3_stmt
try1_stmt  ::=  "try" ":" suite
                ("except" [expression ["as" identifier]] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]
try2_stmt  ::=  "try" ":" suite
                ("except" "*" expression ["as" identifier] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]
try3_stmt  ::=  "try" ":" suite
                "finally" ":" suite
```

Informações adicionais sobre exceções podem ser encontradas na seção *Exceções*, e informações sobre como usar a instrução *raise* para gerar exceções podem ser encontradas na seção *A instrução raise*.

8.4.1 Cláusula `except`

A(s) cláusula(s) `except` especifica(m) um ou mais manipuladores de exceção. Quando nenhuma exceção ocorre na cláusula *try*, nenhum manipulador de exceção é executado. Quando ocorre uma exceção no conjunto `try`, uma busca por um manipulador de exceção é iniciada. Esta pesquisa inspeciona as cláusulas `except` sucessivamente até que seja encontrada uma que corresponda à exceção. Uma cláusula `except` sem expressão, se presente, deve ser a última; corresponde a qualquer exceção. Para uma cláusula `except` com uma expressão, essa expressão é avaliada e a cláusula corresponde à exceção se o objeto resultante for "compatível" com a exceção. Um objeto é compatível com uma exceção se o objeto for a classe ou uma *classe base não virtual* do objeto de exceção, ou uma tupla contendo um item que seja a classe ou uma classe base não virtual do objeto de exceção.

Se nenhuma cláusula `except` corresponder à exceção, a busca por um manipulador de exceção continua no código circundante e na pilha de invocação.[1]

Se a avaliação de uma expressão no cabeçalho de uma cláusula `except` levantar uma exceção, a busca original por um manipulador será cancelada e uma busca pela nova exceção será iniciada no código circundante e na pilha de chamadas (ela é tratado como se toda a instrução *try* levantasse a exceção).

[1] The exception is propagated to the invocation stack unless there is a *finally* clause which happens to raise another exception. That new exception causes the old one to be lost.

Quando uma cláusula `except` correspondente é encontrada, a exceção é atribuída ao destino especificado após a palavra reservada `as` nessa cláusula `except`, se presente, e o conjunto da cláusula `except` é executado. Todas as cláusulas `except` devem ter um bloco executável. Quando o final deste bloco é atingido, a execução continua normalmente após toda a instrução *try*. (Isso significa que se existirem dois manipuladores aninhados para a mesma exceção, e a exceção ocorrer na cláusula `try` do manipulador interno, o manipulador externo não tratará a exceção.)

Quando uma exceção foi atribuída usando `as target`, ela é limpa no final da cláusula `except`. É como se

```
except E as N:
    foo
```

fosse traduzido para

```
except E as N:
    try:
        foo
    finally:
        del N
```

Isso significa que a exceção deve ser atribuída a um nome diferente para poder referenciá-la após a cláusula `except`. As exceções são limpas porque, com o traceback (situação da pilha de execução) anexado a elas, elas formam um ciclo de referência com o quadro de pilha, mantendo todos os locais nesse quadro vivos até que ocorra a próxima coleta de lixo.

Antes de um conjunto de cláusulas `except` ser executado, a exceção é armazenada no módulo `sys`, onde pode ser acessada de dentro do corpo da cláusula `except` chamando `sys.exception()`. Ao sair de um manipulador de exceções, a exceção armazenada no módulo `sys` é redefinida para seu valor anterior:

```
>>> print(sys.exception())
None
>>> try:
...     raise TypeError
... except:
...     print(repr(sys.exception()))
...     try:
...         raise ValueError
...     except:
...         print(repr(sys.exception()))
...     print(repr(sys.exception()))
...
TypeError()
ValueError()
TypeError()
>>> print(sys.exception())
None
```

8.4.2 `except*` clause

The `except*` clause(s) are used for handling `ExceptionGroups`. The exception type for matching is interpreted as in the case of *except*, but in the case of exception groups we can have partial matches when the type matches some of the exceptions in the group. This means that multiple `except*` clauses can execute, each handling part of the exception group. Each clause executes at most once and handles an exception group of all matching exceptions. Each exception in the group is handled by at most one `except*` clause, the first that matches it.

```
>>> try:
...     raise ExceptionGroup("eg",
...         [ValueError(1), TypeError(2), OSError(3), OSError(4)])
... except* TypeError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
... except* OSError as e:
```

(continua na próxima página)

(continuação da página anterior)

```
...     print(f'caught {type(e)} with nested {e.exceptions}')
...
caught <class 'ExceptionGroup'> with nested (TypeError(2),)
caught <class 'ExceptionGroup'> with nested (OSError(3), OSError(4))
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: eg
  +-+---------------- 1 ----------------
    | ValueError: 1
    +------------------------------------
```

Any remaining exceptions that were not handled by any `except*` clause are re-raised at the end, along with all exceptions that were raised from within the `except*` clauses. If this list contains more than one exception to reraise, they are combined into an exception group.

If the raised exception is not an exception group and its type matches one of the `except*` clauses, it is caught and wrapped by an exception group with an empty message string.

```
>>> try:
...     raise BlockingIOError
... except* BlockingIOError as e:
...     print(repr(e))
...
ExceptionGroup('', (BlockingIOError()))
```

An `except*` clause must have a matching type, and this type cannot be a subclass of `BaseExceptionGroup`. It is not possible to mix *except* and `except*` in the same *try*. *break*, *continue* and *return* cannot appear in an `except*` clause.

8.4.3 `else` clause

The optional `else` clause is executed if the control flow leaves the *try* suite, no exception was raised, and no *return*, *continue*, or *break* statement was executed. Exceptions in the `else` clause are not handled by the preceding *except* clauses.

8.4.4 `finally` clause

If `finally` is present, it specifies a 'cleanup' handler. The *try* clause is executed, including any *except* and *else* clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The `finally` clause is executed. If there is a saved exception it is re-raised at the end of the `finally` clause. If the `finally` clause raises another exception, the saved exception is set as the context of the new exception. If the `finally` clause executes a *return*, *break* or *continue* statement, the saved exception is discarded:

```
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
...
>>> f()
42
```

The exception information is not available to the program during execution of the `finally` clause.

When a *return*, *break* or *continue* statement is executed in the *try* suite of a `try`…`finally` statement, the `finally` clause is also executed 'on the way out.'

The return value of a function is determined by the last *return* statement executed. Since the `finally` clause always executes, a `return` statement executed in the `finally` clause will always be the last one executed:

```
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
>>> foo()
'finally'
```

Alterado na versão 3.8: Prior to Python 3.8, a *continue* statement was illegal in the `finally` clause due to a problem with the implementation.

8.5 The `with` statement

The *with* statement is used to wrap the execution of a block with methods defined by a context manager (see section *Gerenciadores de contexto da instrução with*). This allows common *try*…*except*…*finally* usage patterns to be encapsulated for convenient reuse.

```
with_stmt           ::=  "with" ( "(" with_stmt_contents ","? ")" | with_stmt_contents
with_stmt_contents  ::=  with_item (","  with_item)*
with_item           ::=  expression ["as" target]
```

The execution of the *with* statement with one "item" proceeds as follows:

1. The context expression (the expression given in the *with_item*) is evaluated to obtain a context manager.
2. The context manager's *__enter__()* is loaded for later use.
3. The context manager's *__exit__()* is loaded for later use.
4. The context manager's *__enter__()* method is invoked.
5. If a target was included in the *with* statement, the return value from *__enter__()* is assigned to it.

   > **Nota:** The *with* statement guarantees that if the *__enter__()* method returns without an error, then *__exit__()* will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 7 below.

6. The suite is executed.
7. The context manager's *__exit__()* method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to *__exit__()*. Otherwise, three `None` arguments are supplied.

   If the suite was exited due to an exception, and the return value from the *__exit__()* method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the *with* statement.

   If the suite was exited for any reason other than an exception, the return value from *__exit__()* is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

The following code:

```
with EXPRESSION as TARGET:
    SUITE
```

is semantically equivalent to:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

With more than one item, the context managers are processed as if multiple *with* statements were nested:

```
with A() as a, B() as b:
    SUITE
```

is semantically equivalent to:

```
with A() as a:
    with B() as b:
        SUITE
```

You can also write multi-item context managers in multiple lines if the items are surrounded by parentheses. For example:

```
with (
    A() as a,
    B() as b,
):
    SUITE
```

Alterado na versão 3.1: Support for multiple context expressions.

Alterado na versão 3.10: Support for using grouping parentheses to break the statement in multiple lines.

Ver também:

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

8.6 The `match` statement

Novo na versão 3.10.

The match statement is used for pattern matching. Syntax:

```
match_stmt    ::=   'match' subject_expr ":" NEWLINE INDENT case_block+ DEDENT
subject_expr  ::=   star_named_expression "," star_named_expressions?
                    | named_expression
case_block    ::=   'case' patterns [guard] ":" block
```

Nota: This section uses single quotes to denote *soft keywords*.

Pattern matching takes a pattern as input (following `case`) and a subject value (following `match`). The pattern (which may contain subpatterns) is matched against the subject value. The outcomes are:

- A match success or failure (also termed a pattern success or failure).
- Possible binding of matched values to a name. The prerequisites for this are further discussed below.

The `match` and `case` keywords are *soft keywords*.

Ver também:

- **PEP 634** – Structural Pattern Matching: Specification
- **PEP 636** – Structural Pattern Matching: Tutorial

8.6.1 Visão Geral

Here's an overview of the logical flow of a match statement:

1. The subject expression `subject_expr` is evaluated and a resulting subject value obtained. If the subject expression contains a comma, a tuple is constructed using the standard rules.
2. Each pattern in a `case_block` is attempted to match with the subject value. The specific rules for success or failure are described below. The match attempt can also bind some or all of the standalone names within the pattern. The precise pattern binding rules vary per pattern type and are specified below. **Name bindings made during a successful pattern match outlive the executed block and can be used after the match statement**.

   Nota: During failed pattern matches, some subpatterns may succeed. Do not rely on bindings being made for a failed match. Conversely, do not rely on variables remaining unchanged after a failed match. The exact behavior is dependent on implementation and may vary. This is an intentional decision made to allow different implementations to add optimizations.

3. If the pattern succeeds, the corresponding guard (if present) is evaluated. In this case all name bindings are guaranteed to have happened.
   - If the guard evaluates as true or is missing, the `block` inside `case_block` is executed.
   - Otherwise, the next `case_block` is attempted as described above.
   - If there are no further case blocks, the match statement is completed.

Nota: Users should generally never rely on a pattern being evaluated. Depending on implementation, the interpreter may cache values or use other optimizations which skip repeated evaluations.

A sample match statement:

```
>>> flag = False
>>> match (100, 200):
...     case (100, 300):  # Mismatch: 200 != 300
...         print('Case 1')
...     case (100, 200) if flag:  # Successful match, but guard fails
...         print('Case 2')
...     case (100, y):  # Matches and binds y to 200
...         print(f'Case 3, y: {y}')
...     case _:  # Pattern not attempted
...         print('Case 4, I match anything!')
```

(continua na próxima página)

(continuação da página anterior)

```
...
Case 3, y: 200
```

In this case, `if flag` is a guard. Read more about that in the next section.

8.6.2 Guards

```
guard  ::=  "if" named_expression
```

A `guard` (which is part of the `case`) must succeed for code inside the `case` block to execute. It takes the form: *`if`* followed by an expression.

The logical flow of a `case` block with a `guard` follows:

1. Check that the pattern in the `case` block succeeded. If the pattern failed, the `guard` is not evaluated and the next `case` block is checked.
2. If the pattern succeeded, evaluate the `guard`.
   - If the `guard` condition evaluates as true, the case block is selected.
   - If the `guard` condition evaluates as false, the case block is not selected.
   - If the `guard` raises an exception during evaluation, the exception bubbles up.

Guards are allowed to have side effects as they are expressions. Guard evaluation must proceed from the first to the last case block, one at a time, skipping case blocks whose pattern(s) don't all succeed. (I.e., guard evaluation must happen in order.) Guard evaluation must stop once a case block is selected.

8.6.3 Irrefutable Case Blocks

An irrefutable case block is a match-all case block. A match statement may have at most one irrefutable case block, and it must be last.

A case block is considered irrefutable if it has no guard and its pattern is irrefutable. A pattern is considered irrefutable if we can prove from its syntax alone that it will always succeed. Only the following patterns are irrefutable:

- *AS Patterns* whose left-hand side is irrefutable
- *OR Patterns* containing at least one irrefutable pattern
- *Capture Patterns*
- *Wildcard Patterns*
- parenthesized irrefutable patterns

8.6.4 Patterns

Nota: This section uses grammar notations beyond standard EBNF:

- the notation `SEP.RULE+` is shorthand for `RULE (SEP RULE)*`
- the notation `!RULE` is shorthand for a negative lookahead assertion

---

The top-level syntax for `patterns` is:

```
patterns        ::=  open_sequence_pattern | pattern
```

```
pattern         ::=   as_pattern | or_pattern
closed_pattern  ::=   | literal_pattern
                      | capture_pattern
                      | wildcard_pattern
                      | value_pattern
                      | group_pattern
                      | sequence_pattern
                      | mapping_pattern
                      | class_pattern
```

The descriptions below will include a description "in simple terms" of what a pattern does for illustration purposes (credits to Raymond Hettinger for a document that inspired most of the descriptions). Note that these descriptions are purely for illustration purposes and **may not** reflect the underlying implementation. Furthermore, they do not cover all valid forms.

OR Patterns

An OR pattern is two or more patterns separated by vertical bars `|`. Syntax:

```
or_pattern  ::=   "|".closed_pattern+
```

Only the final subpattern may be *irrefutable*, and each subpattern must bind the same set of names to avoid ambiguity.

An OR pattern matches each of its subpatterns in turn to the subject value, until one succeeds. The OR pattern is then considered successful. Otherwise, if none of the subpatterns succeed, the OR pattern fails.

In simple terms, `P1 | P2 | ...` will try to match `P1`, if it fails it will try to match `P2`, succeeding immediately if any succeeds, failing otherwise.

AS Patterns

An AS pattern matches an OR pattern on the left of the `as` keyword against a subject. Syntax:

```
as_pattern  ::=   or_pattern "as" capture_pattern
```

If the OR pattern fails, the AS pattern fails. Otherwise, the AS pattern binds the subject to the name on the right of the as keyword and succeeds. `capture_pattern` cannot be a `_`.

In simple terms `P as NAME` will match with `P`, and on success it will set `NAME = <subject>`.

Literal Patterns

A literal pattern corresponds to most *literals* in Python. Syntax:

```
literal_pattern  ::=   signed_number
                       | signed_number "+" NUMBER
                       | signed_number "-" NUMBER
                       | strings
                       | "None"
                       | "True"
                       | "False"
                       | signed_number: NUMBER | "-" NUMBER
```

The rule `strings` and the token `NUMBER` are defined in the *standard Python grammar*. Triple-quoted strings are

supported. Raw strings and byte strings are supported. *f-strings* are not supported.

The forms `signed_number '+' NUMBER` and `signed_number '-' NUMBER` are for expressing *complex numbers*; they require a real number on the left and an imaginary number on the right. E.g. `3 + 4j`.

In simple terms, `LITERAL` will succeed only if `<subject> == LITERAL`. For the singletons `None`, `True` and `False`, the `is` operator is used.

Capture Patterns

A capture pattern binds the subject value to a name. Syntax:

```
capture_pattern  ::=  !'_' NAME
```

A single underscore _ is not a capture pattern (this is what `!'_'` expresses). It is instead treated as a *wildcard_pattern*.

In a given pattern, a given name can only be bound once. E.g. `case x, x: ...` is invalid while `case [x] | x: ...` is allowed.

Capture patterns always succeed. The binding follows scoping rules established by the assignment expression operator in **PEP 572**; the name becomes a local variable in the closest containing function scope unless there's an applicable `global` or `nonlocal` statement.

In simple terms `NAME` will always succeed and it will set `NAME = <subject>`.

Wildcard Patterns

A wildcard pattern always succeeds (matches anything) and binds no name. Syntax:

```
wildcard_pattern  ::=  '_'
```

_ is a *soft keyword* within any pattern, but only within patterns. It is an identifier, as usual, even within `match` subject expressions, `guard`s, and `case` blocks.

In simple terms, _ will always succeed.

Value Patterns

A value pattern represents a named value in Python. Syntax:

```
value_pattern  ::=  attr
attr           ::=  name_or_attr "." NAME
name_or_attr   ::=  attr | NAME
```

The dotted name in the pattern is looked up using standard Python *name resolution rules*. The pattern succeeds if the value found compares equal to the subject value (using the == equality operator).

In simple terms `NAME1.NAME2` will succeed only if `<subject> == NAME1.NAME2`

Nota: If the same value occurs multiple times in the same match statement, the interpreter may cache the first value found and reuse it rather than repeat the same lookup. This cache is strictly tied to a given execution of a given match statement.

Group Patterns

A group pattern allows users to add parentheses around patterns to emphasize the intended grouping. Otherwise, it has no additional syntax. Syntax:

```
group_pattern  ::=  "(" pattern ")"
```

In simple terms `(P)` has the same effect as `P`.

Sequence Patterns

A sequence pattern contains several subpatterns to be matched against sequence elements. The syntax is similar to the unpacking of a list or tuple.

```
sequence_pattern         ::=  "[" [maybe_sequence_pattern] "]"
                              | "(" [open_sequence_pattern] ")"
open_sequence_pattern    ::=  maybe_star_pattern "," [maybe_sequence_pattern]
maybe_sequence_pattern   ::=  ",".maybe_star_pattern+ ","?
maybe_star_pattern       ::=  star_pattern | pattern
star_pattern             ::=  "*" (capture_pattern | wildcard_pattern)
```

There is no difference if parentheses or square brackets are used for sequence patterns (i.e. `(...)` vs `[...]`).

Nota: A single pattern enclosed in parentheses without a trailing comma (e.g. `(3 | 4)`) is a *group pattern*. While a single pattern enclosed in square brackets (e.g. `[3 | 4]`) is still a sequence pattern.

At most one star subpattern may be in a sequence pattern. The star subpattern may occur in any position. If no star subpattern is present, the sequence pattern is a fixed-length sequence pattern; otherwise it is a variable-length sequence pattern.

The following is the logical flow for matching a sequence pattern against a subject value:

1. If the subject value is not a sequence[2], the sequence pattern fails.
2. If the subject value is an instance of `str`, `bytes` or `bytearray` the sequence pattern fails.
3. The subsequent steps depend on whether the sequence pattern is fixed or variable-length.

   If the sequence pattern is fixed-length:

   1. If the length of the subject sequence is not equal to the number of subpatterns, the sequence pattern fails
   2. Subpatterns in the sequence pattern are matched to their corresponding items in the subject sequence from left to right. Matching stops as soon as a subpattern fails. If all subpatterns succeed in matching

[2] In pattern matching, a sequence is defined as one of the following:
- a class that inherits from `collections.abc.Sequence`
- a Python class that has been registered as `collections.abc.Sequence`
- a builtin class that has its (CPython) `Py_TPFLAGS_SEQUENCE` bit set
- a class that inherits from any of the above

The following standard library classes are sequences:
- `array.array`
- `collections.deque`
- `list`
- `memoryview`
- `range`
- `tuple`

Nota: Subject values of type `str`, `bytes`, and `bytearray` do not match sequence patterns.

---

their corresponding item, the sequence pattern succeeds.

Otherwise, if the sequence pattern is variable-length:

1. If the length of the subject sequence is less than the number of non-star subpatterns, the sequence pattern fails.
2. The leading non-star subpatterns are matched to their corresponding items as for fixed-length sequences.
3. If the previous step succeeds, the star subpattern matches a list formed of the remaining subject items, excluding the remaining items corresponding to non-star subpatterns following the star subpattern.
4. Remaining non-star subpatterns are matched to their corresponding subject items, as for a fixed-length sequence.

Nota: The length of the subject sequence is obtained via `len()` (i.e. via the `__len__()` protocol). This length may be cached by the interpreter in a similar manner as *value patterns*.

In simple terms `[P1, P2, P3, … , P<N>]` matches only if all the following happens:

- check `<subject>` is a sequence
- `len(subject) == <N>`
- `P1` matches `<subject>[0]` (note that this match can also bind names)
- `P2` matches `<subject>[1]` (note that this match can also bind names)
- … and so on for the corresponding pattern/element.

Mapping Patterns

A mapping pattern contains one or more key-value patterns. The syntax is similar to the construction of a dictionary. Syntax:

```
mapping_pattern      ::=   "{" [items_pattern] "}"
items_pattern        ::=   ",".key_value_pattern+ ","?
key_value_pattern    ::=   (literal_pattern | value_pattern) ":" pattern
                         | double_star_pattern
double_star_pattern  ::=   "**" capture_pattern
```

At most one double star pattern may be in a mapping pattern. The double star pattern must be the last subpattern in the mapping pattern.

Duplicate keys in mapping patterns are disallowed. Duplicate literal keys will raise a `SyntaxError`. Two keys that otherwise have the same value will raise a `ValueError` at runtime.

The following is the logical flow for matching a mapping pattern against a subject value:

1. If the subject value is not a mapping[3],the mapping pattern fails.
2. If every key given in the mapping pattern is present in the subject mapping, and the pattern for each key matches the corresponding item of the subject mapping, the mapping pattern succeeds.
3. If duplicate keys are detected in the mapping pattern, the pattern is considered invalid. A `SyntaxError` is raised for duplicate literal values; or a `ValueError` for named keys of the same value.

[3] In pattern matching, a mapping is defined as one of the following:
- a class that inherits from `collections.abc.Mapping`
- a Python class that has been registered as `collections.abc.Mapping`
- a builtin class that has its (CPython) `Py_TPFLAGS_MAPPING` bit set
- a class that inherits from any of the above

The standard library classes `dict` and `types.MappingProxyType` are mappings.

Nota: Key-value pairs are matched using the two-argument form of the mapping subject's `get()` method. Matched key-value pairs must already be present in the mapping, and not created on-the-fly via `__missing__()` or *`__getitem__()`*.

In simple terms `{KEY1: P1, KEY2: P2, ... }` matches only if all the following happens:

- check `<subject>` is a mapping
- `KEY1 in <subject>`
- `P1` matches `<subject>[KEY1]`
- … and so on for the corresponding KEY/pattern pair.

Class Patterns

A class pattern represents a class and its positional and keyword arguments (if any). Syntax:

```
class_pattern        ::=  name_or_attr "(" [pattern_arguments ","?] ")"
pattern_arguments    ::=  positional_patterns [","  keyword_patterns]
                        | keyword_patterns
positional_patterns  ::=  ",".pattern+
keyword_patterns     ::=  ",".keyword_pattern+
keyword_pattern      ::=  NAME "=" pattern
```

The same keyword should not be repeated in class patterns.

The following is the logical flow for matching a class pattern against a subject value:

1. If `name_or_attr` is not an instance of the builtin `type` , raise `TypeError`.
2. If the subject value is not an instance of `name_or_attr` (tested via `isinstance()`), the class pattern fails.
3. If no pattern arguments are present, the pattern succeeds. Otherwise, the subsequent steps depend on whether keyword or positional argument patterns are present.

   For a number of built-in types (specified below), a single positional subpattern is accepted which will match the entire subject; for these types keyword patterns also work as for other types.

   If only keyword patterns are present, they are processed as follows, one by one:

   I. The keyword is looked up as an attribute on the subject.

   - If this raises an exception other than `AttributeError`, the exception bubbles up.
   - If this raises `AttributeError`, the class pattern has failed.
   - Else, the subpattern associated with the keyword pattern is matched against the subject's attribute value. If this fails, the class pattern fails; if this succeeds, the match proceeds to the next keyword.

   II. If all keyword patterns succeed, the class pattern succeeds.

   If any positional patterns are present, they are converted to keyword patterns using the *`__match_args__`* attribute on the class `name_or_attr` before matching:

   I. The equivalent of `getattr(cls, "__match_args__", ())` is called.

   - If this raises an exception, the exception bubbles up.
   - If the returned value is not a tuple, the conversion fails and `TypeError` is raised.
   - If there are more positional patterns than `len(cls.__match_args__)`, `TypeError` is raised.

- Otherwise, positional pattern `i` is converted to a keyword pattern using `__match_args__[i]` as the keyword. `__match_args__[i]` must be a string; if not `TypeError` is raised.
- If there are duplicate keywords, `TypeError` is raised.

Ver também:

Customizando argumentos posicionais na classe correspondência de padrão

II. Once all positional patterns have been converted to keyword patterns,
the match proceeds as if there were only keyword patterns.

For the following built-in types the handling of positional subpatterns is different:

- `bool`
- `bytearray`
- `bytes`
- `dict`
- `float`
- `frozenset`
- `int`
- `list`
- `set`
- `str`
- `tuple`

These classes accept a single positional argument, and the pattern there is matched against the whole object rather than an attribute. For example `int(0|1)` matches the value `0`, but not the value `0.0`.

In simple terms `CLS(P1, attr=P2)` matches only if the following happens:

- `isinstance(<subject>, CLS)`
- convert `P1` to a keyword pattern using `CLS.__match_args__`
- For each keyword argument `attr=P2`:
  - `hasattr(<subject>, "attr")`
  - `P2` matches `<subject>.attr`
- … and so on for the corresponding keyword argument/pattern pair.

Ver também:

- **PEP 634** – Structural Pattern Matching: Specification
- **PEP 636** – Structural Pattern Matching: Tutorial

8.7 Definições de função

A function definition defines a user-defined function object (see section *A hierarquia de tipos padrão*):

```
funcdef                 ::=  [decorators] "def" funcname [type_params] "(" [paramete
                             ["->" expression] ":" suite
decorators              ::=  decorator+
decorator               ::=  "@" assignment_expression NEWLINE
parameter_list          ::=  defparameter ("," defparameter)* "," "/" ["," [paramete
                           | parameter_list_no_posonly
parameter_list_no_posonly ::=  defparameter ("," defparameter)* ["," [parameter_list_s
                           | parameter_list_starargs
parameter_list_starargs ::=  "*" [parameter] ("," defparameter)* ["," ["**" paramete
                           | "**" parameter ["," ]
parameter               ::=  identifier [":" expression]
defparameter            ::=  parameter ["=" expression]
funcname                ::=  identifier
```

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called.[4]

A function definition may be wrapped by one or more *decorator* expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code

```
@f1(arg)
@f2
def func(): pass
```

is roughly equivalent to

```
def func(): pass
func = f1(arg)(f2(func))
```

except that the original function is not temporarily bound to the name `func`.

Alterado na versão 3.9: Functions may be decorated with any valid `assignment_expression`. Previously, the grammar was much more restrictive; see **PEP 614** for details.

A list of *type parameters* may be given in square brackets between the function's name and the opening parenthesis for its parameter list. This indicates to static type checkers that the function is generic. At runtime, the type parameters can be retrieved from the function's `__type_params__` attribute. See *Generic functions* for more.

Alterado na versão 3.12: Type parameter lists are new in Python 3.12.

When one or more *parameters* have the form *parameter* = *expression*, the function is said to have "default parameter values." For a parameter with a default value, the corresponding *argument* may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters up until the "`*`" must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

Default parameter values are evaluated from left to right when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same "pre-computed" value is used for each call. This is especially important to understand when a default parameter value is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default parameter

[4] A string literal appearing as the first statement in the function body is transformed into the function's `__doc__` attribute and therefore the function's *docstring*.

value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section *Chamadas*. A function call always assigns values to all parameters mentioned in the parameter list, either from positional arguments, from keyword arguments, or from default values. If the form "`*identifier`" is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form "`**identifier`" is present, it is initialized to a new ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type. Parameters after "`*`" or "`*identifier`" are keyword-only parameters and may only be passed by keyword arguments. Parameters before "`/`" are positional-only parameters and may only be passed by positional arguments.

Alterado na versão 3.8: The `/` function parameter syntax may be used to indicate positional-only parameters. See **PEP 570** for details.

Parameters may have an *annotation* of the form "`: expression`" following the parameter name. Any parameter may have an annotation, even those of the form `*identifier` or `**identifier`. Functions may have "return" annotation of the form "`-> expression`" after the parameter list. These annotations can be any valid Python expression. The presence of annotations does not change the semantics of a function. The annotation values are available as values of a dictionary keyed by the parameters' names in the `__annotations__` attribute of the function object. If the `annotations` import from `__future__` is used, annotations are preserved as strings at runtime which enables postponed evaluation. Otherwise, they are evaluated when the function definition is executed. In this case annotations may be evaluated in a different order than they appear in the source code.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda expressions, described in section *Lambdas*. Note that the lambda expression is merely a shorthand for a simplified function definition; a function defined in a "*`def`*" statement can be passed around or assigned to another name just like a function defined by a lambda expression. The "`def`" form is actually more powerful since it allows the execution of multiple statements and annotations.

Programmer's note: Functions are first-class objects. A "`def`" statement executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section *Nomeação e ligação* for details.

Ver também:

PEP 3107 - Function Annotations
The original specification for function annotations.

PEP 484 - Dicas de tipos
Definition of a standard meaning for annotations: type hints.

PEP 526 - Sintaxe para Anotações de Variáveis
Ability to type hint variable declarations, including class variables and instance variables.

PEP 563 - Postponed Evaluation of Annotations
Support for forward references within annotations by preserving annotations in a string form at runtime instead of eager evaluation.

PEP 318 - Decorators for Functions and Methods
Function and method decorators were introduced. Class decorators were introduced in **PEP 3129**.

8.8 Definições de classe

A class definition defines a class object (see section *A hierarquia de tipos padrão*):

```
classdef    ::=  [decorators] "class" classname [type_params] [inheritance] ":" suite
inheritance ::=  "(" [argument_list] ")"
classname   ::=  identifier
```

A class definition is an executable statement. The inheritance list usually gives a list of base classes (see *Metaclasses* for more advanced uses), so each item in the list should evaluate to a class object which allows subclassing. Classes without an inheritance list inherit, by default, from the base class `object`; hence,

```
class Foo:
    pass
```

é equivalente a:

```
class Foo(object):
    pass
```

The class's suite is then executed in a new execution frame (see *Nomeação e ligação*), using a newly created local namespace and the original global namespace. (Usually, the suite contains mostly function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved.[5] A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

The order in which attributes are defined in the class body is preserved in the new class's `__dict__`. Note that this is reliable only right after the class is created and only for classes that were defined using the definition syntax.

Class creation can be customized heavily using *metaclasses*.

Classes can also be decorated: just like when decorating functions,

```
@f1(arg)
@f2
class Foo: pass
```

is roughly equivalent to

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

The evaluation rules for the decorator expressions are the same as for function decorators. The result is then bound to the class name.

Alterado na versão 3.9: Classes may be decorated with any valid `assignment_expression`. Previously, the grammar was much more restrictive; see **PEP 614** for details.

A list of *type parameters* may be given in square brackets immediately after the class's name. This indicates to static type checkers that the class is generic. At runtime, the type parameters can be retrieved from the class's `__type_params__` attribute. See *Generic classes* for more.

Alterado na versão 3.12: Type parameter lists are new in Python 3.12.

Programmer's note: Variables defined in the class definition are class attributes; they are shared by instances. Instance attributes can be set in a method with `self.name = value`. Both class and instance attributes are accessible through the notation "`self.name`", and an instance attribute hides a class attribute with the same name when accessed in this way. Class attributes can be used as defaults for instance attributes, but using mutable values

[5] A string literal appearing as the first statement in the class body is transformed into the namespace's `__doc__` item and therefore the class's *docstring*.

there can lead to unexpected results. *Descriptors* can be used to create instance variables with different implementation details.

Ver também:

PEP 3115 - Metaclasses no Python 3000
The proposal that changed the declaration of metaclasses to the current syntax, and the semantics for how classes with metaclasses are constructed.

PEP 3129 - Class Decorators
The proposal that added class decorators. Function and method decorators were introduced in **PEP 318**.

8.9 Corrotinas

Novo na versão 3.5.

8.9.1 Coroutine function definition

```
async_funcdef   ::=   [decorators] "async" "def" funcname "(" [parameter_list] ")"
                      ["->" expression] ":" suite
```

Execution of Python coroutines can be suspended and resumed at many points (see *coroutine*). `await` expressions, `async for` and `async with` can only be used in the body of a coroutine function.

Functions defined with `async def` syntax are always coroutine functions, even if they do not contain `await` or `async` keywords.

It is a `SyntaxError` to use a `yield from` expression inside the body of a coroutine function.

An example of a coroutine function:

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

Alterado na versão 3.7: `await` and `async` are now keywords; previously they were only treated as such inside the body of a coroutine function.

8.9.2 The `async for` statement

```
async_for_stmt   ::=   "async" for_stmt
```

An *asynchronous iterable* provides an `__aiter__` method that directly returns an *asynchronous iterator*, which can call asynchronous code in its `__anext__` method.

The `async for` statement allows convenient iteration over asynchronous iterables.

The following code:

```
async for TARGET in ITER:
    SUITE
else:
    SUITE2
```

Is semantically equivalent to:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
```

(continua na próxima página)

(continuação da página anterior)

```
running = True

while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        SUITE
else:
    SUITE2
```

See also *__aiter__()* and *__anext__()* for details.

It is a `SyntaxError` to use an `async for` statement outside the body of a coroutine function.

8.9.3 The `async with` statement

```
async_with_stmt ::= "async" with_stmt
```

An *asynchronous context manager* is a *context manager* that is able to suspend execution in its *enter* and *exit* methods.

The following code:

```
async with EXPRESSION as TARGET:
    SUITE
```

is semantically equivalent to:

```
manager = (EXPRESSION)
aenter = type(manager).__aenter__
aexit = type(manager).__aexit__
value = await aenter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not await aexit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        await aexit(manager, None, None, None)
```

See also *__aenter__()* and *__aexit__()* for details.

It is a `SyntaxError` to use an `async with` statement outside the body of a coroutine function.

Ver também:

PEP 492 - Coroutines with async and await syntax
The proposal that made coroutines a proper standalone concept in Python, and added supporting syntax.

8.10 Type parameter lists

Novo na versão 3.12.

```
type_params   ::=  "[" type_param ("," type_param)* "]"
type_param    ::=  typevar | typevartuple | paramspec
typevar       ::=  identifier (":" expression)?
typevartuple  ::=  "*" identifier
paramspec     ::=  "**" identifier
```

Functions (including *coroutines*), *classes* and *type aliases* may contain a type parameter list:

```
def max[T](args: list[T]) -> T:
    ...

async def amax[T](args: list[T]) -> T:
    ...

class Bag[T]:
    def __iter__(self) -> Iterator[T]:
        ...

    def add(self, arg: T) -> None:
        ...

type ListOrSet[T] = list[T] | set[T]
```

Semantically, this indicates that the function, class, or type alias is generic over a type variable. This information is primarily used by static type checkers, and at runtime, generic objects behave much like their non-generic counterparts.

Type parameters are declared in square brackets (`[]`) immediately after the name of the function, class, or type alias. The type parameters are accessible within the scope of the generic object, but not elsewhere. Thus, after a declaration `def func[T](): pass`, the name `T` is not available in the module scope. Below, the semantics of generic objects are described with more precision. The scope of type parameters is modeled with a special function (technically, an *annotation scope*) that wraps the creation of the generic object.

Generic functions, classes, and type aliases have a `__type_params__` attribute listing their type parameters.

Type parameters come in three kinds:

- `typing.TypeVar`, introduced by a plain name (e.g., `T`). Semantically, this represents a single type to a type checker.
- `typing.TypeVarTuple`, introduced by a name prefixed with a single asterisk (e.g., `*Ts`). Semantically, this stands for a tuple of any number of types.
- `typing.ParamSpec`, introduced by a name prefixed with two asterisks (e.g., `**P`). Semantically, this stands for the parameters of a callable.

`typing.TypeVar` declarations can define *bounds* and *constraints* with a colon (`:`) followed by an expression. A single expression after the colon indicates a bound (e.g. `T: int`). Semantically, this means that the `typing.TypeVar` can only represent types that are a subtype of this bound. A parenthesized tuple of expressions after the colon indicates a set of constraints (e.g. `T: (str, bytes)`). Each member of the tuple should be a type (again, this is not enforced at runtime). Constrained type variables can only take on one of the types in the list of constraints.

For `typing.TypeVar`s declared using the type parameter list syntax, the bound and constraints are not evaluated when the generic object is created, but only when the value is explicitly accessed through the attributes `__bound__` and `__constraints__`. To accomplish this, the bounds or constraints are evaluated in a separate *annotation scope*.

`typing.TypeVarTuple`s and `typing.ParamSpec`s cannot have bounds or constraints.

The following example indicates the full set of allowed type parameter declarations:

```
def overly_generic[
   SimpleTypeVar,
   TypeVarWithBound: int,
   TypeVarWithConstraints: (str, bytes),
   *SimpleTypeVarTuple,
   **SimpleParamSpec,
](
   a: SimpleTypeVar,
   b: TypeVarWithBound,
   c: Callable[SimpleParamSpec, TypeVarWithConstraints],
   *d: SimpleTypeVarTuple,
): ...
```

8.10.1 Generic functions

Generic functions are declared as follows:

```
def func[T](arg: T): ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_func():
    T = typing.TypeVar("T")
    def func(arg: T): ...
    func.__type_params__ = (T,)
    return func
func = TYPE_PARAMS_OF_func()
```

Here `annotation-def` indicates an *annotation scope*, which is not actually bound to any name at runtime. (One other liberty is taken in the translation: the syntax does not go through attribute access on the `typing` module, but creates an instance of `typing.TypeVar` directly.)

The annotations of generic functions are evaluated within the annotation scope used for declaring the type parameters, but the function's defaults and decorators are not.

The following example illustrates the scoping rules for these cases, as well as for additional flavors of type parameters:

```
@decorator
def func[T: int, *Ts, **P](*args: *Ts, arg: Callable[P, T] = some_default):
    ...
```

Except for the *lazy evaluation* of the `TypeVar` bound, this is equivalent to:

```
DEFAULT_OF_arg = some_default

annotation-def TYPE_PARAMS_OF_func():

    annotation-def BOUND_OF_T():
        return int
    # In reality, BOUND_OF_T() is evaluated only on demand.
    T = typing.TypeVar("T", bound=BOUND_OF_T())

    Ts = typing.TypeVarTuple("Ts")
    P = typing.ParamSpec("P")

    def func(*args: *Ts, arg: Callable[P, T] = DEFAULT_OF_arg):
        ...

    func.__type_params__ = (T, Ts, P)
```

(continua na próxima página)

(continuação da página anterior)

```
    return func
func = decorator(TYPE_PARAMS_OF_func())
```

The capitalized names like `DEFAULT_OF_arg` are not actually bound at runtime.

8.10.2 Generic classes

Generic classes are declared as follows:

```
class Bag[T]: ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(typing.Generic[T]):
        __type_params__ = (T,)
        ...
    return Bag
Bag = TYPE_PARAMS_OF_Bag()
```

Here again `annotation-def` (not a real keyword) indicates an *annotation scope*, and the name `TYPE_PARAMS_OF_Bag` is not actually bound at runtime.

Generic classes implicitly inherit from `typing.Generic`. The base classes and keyword arguments of generic classes are evaluated within the type scope for the type parameters, and decorators are evaluated outside that scope. This is illustrated by this example:

```
@decorator
class Bag(Base[T], arg=T): ...
```

Isso equivale a:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(Base[T], typing.Generic[T], arg=T):
        __type_params__ = (T,)
        ...
    return Bag
Bag = decorator(TYPE_PARAMS_OF_Bag())
```

8.10.3 Generic type aliases

The *type* statement can also be used to create a generic type alias:

```
type ListOrSet[T] = list[T] | set[T]
```

Except for the *lazy evaluation* of the value, this is equivalent to:

```
annotation-def TYPE_PARAMS_OF_ListOrSet():
    T = typing.TypeVar("T")

    annotation-def VALUE_OF_ListOrSet():
        return list[T] | set[T]
    # In reality, the value is lazily evaluated
    return typing.TypeAliasType("ListOrSet", VALUE_OF_ListOrSet(), type_params=(T,
↪))
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

Here, `annotation-def` (not a real keyword) indicates an *annotation scope*. The capitalized names like `TYPE_PARAMS_OF_ListOrSet` are not actually bound at runtime.

CAPÍTULO 9

Componentes de Alto Nível

O interpretador Python pode receber suas entradas de uma quantidade de fontes: de um script passado a ele como entrada padrão ou como um argumento do programa, digitado interativamente, de um arquivo fonte de um módulo, etc. Este capítulo mostra a sintaxe usada nesses casos.

9.1 Programas Python completos

Ainda que uma especificação de linguagem não precise prescrever como o interpretador da linguagem é invocado, é útil ter uma noção de um programa Python completo. Um programa Python completo é executado em um ambiente minimamente inicializado: todos os módulos embutidos e padrões estão disponíveis, mas nenhum foi inicializado, exceto por `sys` (serviços de sistema diversos), `builtins` (funções embutidas, exceções e `None`) e `__main__`. O último é usado para fornecer o espaço de nomes global e local para execução de um programa completo.

A sintaxe para um programa Python completo é esta para uma entrada de arquivo, descrita na próxima seção.

O interpretador também pode ser invocado no modo interativo; neste caso, ele não lê e executa um programa completo, mas lê e executa uma instrução (possivelmente composta) por vez. O ambiente inicial é idêntico àquele de um programa completo; cada instrução é executada no espaço de nomes de `__main__`.

Um programa completo pode ser passado ao interpretador de três formas: com a opção de linha de comando `-c` *string*, como um arquivo passado como o primeiro argumento da linha de comando, ou como uma entrada padrão. Se o arquivo ou a entrada padrão é um dispositivo tty, o interpretador entra em modo interativo; caso contrário, ele executa o arquivo como um programa completo.

9.2 Entrada de arquivo

Toda entrada lida de arquivos não-interativos têm a mesma forma:

```
file_input  ::=  (NEWLINE | statement)*
```

Essa sintaxe é usada nas seguintes situações:

- quando analisando um programa Python completo (a partir de um arquivo ou de uma string);
- quando analisando um módulo;

- quando analisando uma string passada à função `exec()`;

9.3 Entrada interativa

A entrada em modo interativo é analisada usando a seguinte gramática:

```
interactive_input   ::=   [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Note que uma instrução composta (de alto-nível) deve ser seguida por uma linha em branco no modo interativo; isso é necessário para ajudar o analisador sintático a detectar o fim da entrada.

9.4 Entrada de expressão

A função `eval()` é usada para uma entrada de expressão. Ela ignora espaços à esquerda. O argumento em string para `eval()` deve ter a seguinte forma:

```
eval_input   ::=   expression_list NEWLINE*
```

CAPÍTULO 10

Especificação Completa da Gramática

Esta é a gramática completa do Python, derivada diretamente da gramática usada para gerar o analisador sintático de CPython (consulte Grammar/python.gram). A versão aqui omite detalhes relacionados à geração de código e recuperação de erros.

A notação é uma mistura de EBNF e GASE (em inglês, PEG). Em particular, `&` seguido por um símbolo, token ou grupo entre parênteses indica um "olhar a frente" positivo (ou seja, é necessário para corresponder, mas não consumido), enquanto `!` indica um "olhar a frente" negativo (ou seja, é necessário *não* combinar). Usamos o separador `|` para significar a "escolha ordenada" do GASE (escrito como `/` nas gramáticas GASE tradicionais). Veja **PEP 617** para mais detalhes sobre a sintaxe da gramática.

```
# PEG grammar for Python



# ========================= START OF THE GRAMMAR ==========================

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
#     - These rules are NOT used in the first pass of the parser.
#     - Only if the first pass fails to parse, a second pass including the invalid
#       rules will be executed.
#     - If the parser fails in the second phase with a generic syntax error, the
#       location of the generic failure of the first pass will be used (this avoids
#       reporting incorrect locations due to the invalid rules).
#     - The order of the alternatives involving invalid rules matter
#       (like any rule in PEG).
#
# Grammar Syntax (see PEP 617 for more information):
#
# rule_name: expression
#   Optionally, a type can be included right after the rule name, which
#   specifies the return type of the C or Python function corresponding to the
#   rule:
# rule_name[return_type]: expression
#   If the return type is omitted, then a void * is returned in C and an Any in
```

(continua na próxima página)

(continuação da página anterior)

```
#   Python.
# e1 e2
#   Match e1, then match e2.
# e1 | e2
#   Match e1 or e2.
#   The first alternative can also appear on the line after the rule name for
#   formatting purposes. In that case, a | must be used before the first
#   alternative, like so:
#       rule_name[return_type]:
#            | first_alt
#            | second_alt
# ( e )
#   Match e (allows also to use other operators in the group like '(e)*')
# [ e ] or e?
#   Optionally match e.
# e*
#   Match zero or more occurrences of e.
# e+
#   Match one or more occurrences of e.
# s.e+
#   Match one or more occurrences of e, separated by s. The generated parse tree
#   does not include the separator. This is otherwise identical to (e (s e)*).
# &e
#   Succeed if e can be parsed, without consuming any input.
# !e
#   Fail if e can be parsed, without consuming any input.
# ~
#   Commit to the current alternative, even if it fails to parse.
#

# STARTING RULES
# ==============

file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE* ENDMARKER

# GENERAL STATEMENTS
# ==================

statements: statement+

statement: compound_stmt  | simple_stmts

statement_newline:
    | compound_stmt NEWLINE
    | simple_stmts
    | NEWLINE
    | ENDMARKER

simple_stmts:
    | simple_stmt !';' NEWLINE  # Not needed, there for speedup
    | ';'.simple_stmt+ [';'] NEWLINE

# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt:
    | assignment
    | type_alias
    | star_expressions
```

(continua na próxima página)

(continuação da página anterior)

```
    | return_stmt
    | import_stmt
    | raise_stmt
    | 'pass'
    | del_stmt
    | yield_stmt
    | assert_stmt
    | 'break'
    | 'continue'
    | global_stmt
    | nonlocal_stmt

compound_stmt:
    | function_def
    | if_stmt
    | class_def
    | with_stmt
    | for_stmt
    | try_stmt
    | while_stmt
    | match_stmt

# SIMPLE STATEMENTS
# =================

# NOTE: annotated_rhs may start with 'yield'; yield_expr must start with 'yield'
assignment:
    | NAME ':' expression ['=' annotated_rhs ]
    | ('(' single_target ')'
         | single_subscript_attribute_target) ':' expression ['=' annotated_rhs ]
    | (star_targets '=' )+ (yield_expr | star_expressions) !'=' [TYPE_COMMENT]
    | single_target augassign ~ (yield_expr | star_expressions)

annotated_rhs: yield_expr | star_expressions

augassign:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | '>>='
    | '**='
    | '//='

return_stmt:
    | 'return' [star_expressions]

raise_stmt:
    | 'raise' expression ['from' expression ]
    | 'raise'

global_stmt: 'global' ','.NAME+

nonlocal_stmt: 'nonlocal' ','.NAME+
```

(continua na próxima página)

(continuação da página anterior)

```
del_stmt:
    | 'del' del_targets &(';' | NEWLINE)

yield_stmt: yield_expr

assert_stmt: 'assert' expression [',' expression ]

import_stmt:
    | import_name
    | import_from

# Import statements
# -----------------

import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from:
    | 'from' ('.' | '...')* dotted_name 'import' import_from_targets
    | 'from' ('.' | '...')+ 'import' import_from_targets
import_from_targets:
    | '(' import_from_as_names [','] ')'
    | import_from_as_names !','
    | '*'
import_from_as_names:
    | ','.import_from_as_name+
import_from_as_name:
    | NAME ['as' NAME ]
dotted_as_names:
    | ','.dotted_as_name+
dotted_as_name:
    | dotted_name ['as' NAME ]
dotted_name:
    | dotted_name '.' NAME
    | NAME

# COMPOUND STATEMENTS
# ===================

# Common elements
# ---------------

block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmts

decorators: ('@' named_expression NEWLINE )+

# Class definitions
# -----------------

class_def:
    | decorators class_def_raw
    | class_def_raw

class_def_raw:
    | 'class' NAME [type_params] ['(' [arguments] ')' ] ':' block

# Function definitions
# --------------------

function_def:
```

(continua na próxima página)

(continuação da página anterior)

```
    | decorators function_def_raw
    | function_def_raw

function_def_raw:
    | 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_type_
↪comment] block
    | ASYNC 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_
↪type_comment] block

# Function parameters
# --------------------

params:
    | parameters

parameters:
    | slash_no_default param_no_default* param_with_default* [star_etc]
    | slash_with_default param_with_default* [star_etc]
    | param_no_default+ param_with_default* [star_etc]
    | param_with_default+ [star_etc]
    | star_etc

# Some duplication here because we can't write (',' | &')'),
# which is because we don't support empty alternatives (yet).

slash_no_default:
    | param_no_default+ '/' ','
    | param_no_default+ '/' &')'
slash_with_default:
    | param_no_default* param_with_default+ '/' ','
    | param_no_default* param_with_default+ '/' &')'

star_etc:
    | '*' param_no_default param_maybe_default* [kwds]
    | '*' param_no_default_star_annotation param_maybe_default* [kwds]
    | '*' ',' param_maybe_default+ [kwds]
    | kwds

kwds:
    | '**' param_no_default

# One parameter.  This *includes* a following comma and type comment.
#
# There are three styles:
# - No default
# - With default
# - Maybe with default
#
# There are two alternative forms of each, to deal with type comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close paren
# The latter form is for a final parameter without trailing comma.
#

param_no_default:
    | param ',' TYPE_COMMENT?
    | param TYPE_COMMENT? &')'
param_no_default_star_annotation:
    | param_star_annotation ',' TYPE_COMMENT?
    | param_star_annotation TYPE_COMMENT? &')'
param_with_default:
```

(continua na próxima página)

(continuação da página anterior)

```
    | param default ',' TYPE_COMMENT?
    | param default TYPE_COMMENT? &')'
param_maybe_default:
    | param default? ',' TYPE_COMMENT?
    | param default? TYPE_COMMENT? &')'
param: NAME annotation?
param_star_annotation: NAME star_annotation
annotation: ':' expression
star_annotation: ':' star_expression
default: '=' expression  | invalid_default

# If statement
# ------------

if_stmt:
    | 'if' named_expression ':' block elif_stmt
    | 'if' named_expression ':' block [else_block]
elif_stmt:
    | 'elif' named_expression ':' block elif_stmt
    | 'elif' named_expression ':' block [else_block]
else_block:
    | 'else' ':' block

# While statement
# ---------------

while_stmt:
    | 'while' named_expression ':' block [else_block]

# For statement
# -------------

for_stmt:
    | 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_
↪block]
    | ASYNC 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block␣
↪[else_block]

# With statement
# --------------

with_stmt:
    | 'with' '(' ','.with_item+ ','? ')' ':' block
    | 'with' ','.with_item+ ':' [TYPE_COMMENT] block
    | ASYNC 'with' '(' ','.with_item+ ','? ')' ':' block
    | ASYNC 'with' ','.with_item+ ':' [TYPE_COMMENT] block

with_item:
    | expression 'as' star_target &(',' | ')' | ':')
    | expression

# Try statement
# -------------

try_stmt:
    | 'try' ':' block finally_block
    | 'try' ':' block except_block+ [else_block] [finally_block]
    | 'try' ':' block except_star_block+ [else_block] [finally_block]


# Except statement
```

(continua na próxima página)

(continuação da página anterior)

```
# ----------------

except_block:
    | 'except' expression ['as' NAME ] ':' block
    | 'except' ':' block
except_star_block:
    | 'except' '*' expression ['as' NAME ] ':' block
finally_block:
    | 'finally' ':' block

# Match statement
# ---------------

match_stmt:
    | "match" subject_expr ':' NEWLINE INDENT case_block+ DEDENT

subject_expr:
    | star_named_expression ',' star_named_expressions?
    | named_expression

case_block:
    | "case" patterns guard? ':' block

guard: 'if' named_expression

patterns:
    | open_sequence_pattern
    | pattern

pattern:
    | as_pattern
    | or_pattern

as_pattern:
    | or_pattern 'as' pattern_capture_target

or_pattern:
    | '|'.closed_pattern+

closed_pattern:
    | literal_pattern
    | capture_pattern
    | wildcard_pattern
    | value_pattern
    | group_pattern
    | sequence_pattern
    | mapping_pattern
    | class_pattern

# Literal patterns are used for equality and identity constraints
literal_pattern:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

# Literal expressions are used to restrict permitted mapping pattern keys
literal_expr:
    | signed_number !('+' | '-')
```

(continua na próxima página)

(continuação da página anterior)

```
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

complex_number:
    | signed_real_number '+' imaginary_number
    | signed_real_number '-' imaginary_number

signed_number:
    | NUMBER
    | '-' NUMBER

signed_real_number:
    | real_number
    | '-' real_number

real_number:
    | NUMBER

imaginary_number:
    | NUMBER

capture_pattern:
    | pattern_capture_target

pattern_capture_target:
    | !"_" NAME !('.' | '(' | '=')

wildcard_pattern:
    | "_"

value_pattern:
    | attr !('.' | '(' | '=')

attr:
    | name_or_attr '.' NAME

name_or_attr:
    | attr
    | NAME

group_pattern:
    | '(' pattern ')'

sequence_pattern:
    | '[' maybe_sequence_pattern? ']'
    | '(' open_sequence_pattern? ')'

open_sequence_pattern:
    | maybe_star_pattern ',' maybe_sequence_pattern?

maybe_sequence_pattern:
    | ','.maybe_star_pattern+ ','?

maybe_star_pattern:
    | star_pattern
    | pattern

star_pattern:
```

(continua na próxima página)

(continuação da página anterior)

```
    | '*' pattern_capture_target
    | '*' wildcard_pattern

mapping_pattern:
    | '{' '}'
    | '{' double_star_pattern ','? '}'
    | '{' items_pattern ',' double_star_pattern ','? '}'
    | '{' items_pattern ','? '}'

items_pattern:
    | ','.key_value_pattern+

key_value_pattern:
    | (literal_expr | attr) ':' pattern

double_star_pattern:
    | '**' pattern_capture_target

class_pattern:
    | name_or_attr '(' ')'
    | name_or_attr '(' positional_patterns ','? ')'
    | name_or_attr '(' keyword_patterns ','? ')'
    | name_or_attr '(' positional_patterns ',' keyword_patterns ','? ')'

positional_patterns:
    | ','.pattern+

keyword_patterns:
    | ','.keyword_pattern+

keyword_pattern:
    | NAME '=' pattern

# Type statement
# ---------------

type_alias:
    | "type" NAME [type_params] '=' expression

# Type parameter declaration
# --------------------------

type_params: '[' type_param_seq  ']'

type_param_seq: ','.type_param+ [',']

type_param:
    | NAME [type_param_bound]
    | '*' NAME ':' expression
    | '*' NAME
    | '**' NAME ':' expression
    | '**' NAME

type_param_bound: ':' expression

# EXPRESSIONS
# -----------

expressions:
    | expression (',' expression )+ [',']
    | expression ','
```

(continua na próxima página)

(continuação da página anterior)

```
    | expression

expression:
    | disjunction 'if' disjunction 'else' expression
    | disjunction
    | lambdef

yield_expr:
    | 'yield' 'from' expression
    | 'yield' [star_expressions]

star_expressions:
    | star_expression (',' star_expression )+ [',']
    | star_expression ','
    | star_expression

star_expression:
    | '*' bitwise_or
    | expression

star_named_expressions: ','.star_named_expression+ [',']

star_named_expression:
    | '*' bitwise_or
    | named_expression

assignment_expression:
    | NAME ':=' ~ expression

named_expression:
    | assignment_expression
    | expression !':='

disjunction:
    | conjunction ('or' conjunction )+
    | conjunction

conjunction:
    | inversion ('and' inversion )+
    | inversion

inversion:
    | 'not' inversion
    | comparison

# Comparison operators
# --------------------

comparison:
    | bitwise_or compare_op_bitwise_or_pair+
    | bitwise_or

compare_op_bitwise_or_pair:
    | eq_bitwise_or
    | noteq_bitwise_or
    | lte_bitwise_or
    | lt_bitwise_or
    | gte_bitwise_or
    | gt_bitwise_or
    | notin_bitwise_or
    | in_bitwise_or
```

(continua na próxima página)

(continuação da página anterior)

```
    | isnot_bitwise_or
    | is_bitwise_or

eq_bitwise_or: '==' bitwise_or
noteq_bitwise_or:
    | ('!=' ) bitwise_or
lte_bitwise_or: '<=' bitwise_or
lt_bitwise_or: '<' bitwise_or
gte_bitwise_or: '>=' bitwise_or
gt_bitwise_or: '>' bitwise_or
notin_bitwise_or: 'not' 'in' bitwise_or
in_bitwise_or: 'in' bitwise_or
isnot_bitwise_or: 'is' 'not' bitwise_or
is_bitwise_or: 'is' bitwise_or

# Bitwise operators
# -----------------

bitwise_or:
    | bitwise_or '|' bitwise_xor
    | bitwise_xor

bitwise_xor:
    | bitwise_xor '^' bitwise_and
    | bitwise_and

bitwise_and:
    | bitwise_and '&' shift_expr
    | shift_expr

shift_expr:
    | shift_expr '<<' sum
    | shift_expr '>>' sum
    | sum

# Arithmetic operators
# --------------------

sum:
    | sum '+' term
    | sum '-' term
    | term

term:
    | term '*' factor
    | term '/' factor
    | term '//' factor
    | term '%' factor
    | term '@' factor
    | factor

factor:
    | '+' factor
    | '-' factor
    | '~' factor
    | power

power:
    | await_primary '**' factor
    | await_primary
```

(continua na próxima página)

(continuação da página anterior)

```
# Primary elements
# ----------------

# Primary elements are things like "obj.something.something", "obj[something]",
↪"obj(something)", "obj" ...

await_primary:
    | AWAIT primary
    | primary

primary:
    | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom

slices:
    | slice !','
    | ','.(slice | starred_expression)+ [',']

slice:
    | [expression] ':' [expression] [':' [expression] ]
    | named_expression

atom:
    | NAME
    | 'True'
    | 'False'
    | 'None'
    | strings
    | NUMBER
    | (tuple | group | genexp)
    | (list | listcomp)
    | (dict | set | dictcomp | setcomp)
    | '...'

group:
    | '(' (yield_expr | named_expression) ')'

# Lambda functions
# ----------------

lambdef:
    | 'lambda' [lambda_params] ':' expression

lambda_params:
    | lambda_parameters

# lambda_parameters etc. duplicates parameters but without annotations
# or type comments, and if there's no comma after a parameter, we expect
# a colon, not a close parenthesis.  (For more, see parameters above.)
#
lambda_parameters:
    | lambda_slash_no_default lambda_param_no_default* lambda_param_with_default*␣
↪[lambda_star_etc]
    | lambda_slash_with_default lambda_param_with_default* [lambda_star_etc]
    | lambda_param_no_default+ lambda_param_with_default* [lambda_star_etc]
    | lambda_param_with_default+ [lambda_star_etc]
    | lambda_star_etc
```

(continua na próxima página)

(continuação da página anterior)

```
lambda_slash_no_default:
    | lambda_param_no_default+ '/' ','
    | lambda_param_no_default+ '/' &':'

lambda_slash_with_default:
    | lambda_param_no_default* lambda_param_with_default+ '/' ','
    | lambda_param_no_default* lambda_param_with_default+ '/' &':'

lambda_star_etc:
    | '*' lambda_param_no_default lambda_param_maybe_default* [lambda_kwds]
    | '*' ',' lambda_param_maybe_default+ [lambda_kwds]
    | lambda_kwds

lambda_kwds:
    | '**' lambda_param_no_default

lambda_param_no_default:
    | lambda_param ','
    | lambda_param &':'
lambda_param_with_default:
    | lambda_param default ','
    | lambda_param default &':'
lambda_param_maybe_default:
    | lambda_param default? ','
    | lambda_param default? &':'
lambda_param: NAME

# LITERALS
# ========

fstring_middle:
    | fstring_replacement_field
    | FSTRING_MIDDLE
fstring_replacement_field:
    | '{' (yield_expr | star_expressions) '='? [fstring_conversion] [fstring_full_
↪format_spec] '}'
fstring_conversion:
    | "!" NAME
fstring_full_format_spec:
    | ':' fstring_format_spec*
fstring_format_spec:
    | FSTRING_MIDDLE
    | fstring_replacement_field
fstring:
    | FSTRING_START fstring_middle* FSTRING_END

string: STRING
strings: (fstring|string)+

list:
    | '[' [star_named_expressions] ']'

tuple:
    | '(' [star_named_expression ',' [star_named_expressions]  ] ')'

set: '{' star_named_expressions '}'

# Dicts
# -----

dict:
```

(continua na próxima página)

(continuação da página anterior)

```
    | '{' [double_starred_kvpairs] '}'

double_starred_kvpairs: ','.double_starred_kvpair+ [',']

double_starred_kvpair:
    | '**' bitwise_or
    | kvpair

kvpair: expression ':' expression

# Comprehensions & Generators
# ---------------------------

for_if_clauses:
    | for_if_clause+

for_if_clause:
    | ASYNC 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
    | 'for' star_targets 'in' ~ disjunction ('if' disjunction )*

listcomp:
    | '[' named_expression for_if_clauses ']'

setcomp:
    | '{' named_expression for_if_clauses '}'

genexp:
    | '(' ( assignment_expression | expression !':=') for_if_clauses ')'

dictcomp:
    | '{' kvpair for_if_clauses '}'

# FUNCTION CALL ARGUMENTS
# =======================

arguments:
    | args [','] &')'

args:
    | ','.(starred_expression | ( assignment_expression | expression !':=') !'=')+␣
↪[',' kwargs ]
    | kwargs

kwargs:
    | ','.kwarg_or_starred+ ',' ','.kwarg_or_double_starred+
    | ','.kwarg_or_starred+
    | ','.kwarg_or_double_starred+

starred_expression:
    | '*' expression

kwarg_or_starred:
    | NAME '=' expression
    | starred_expression

kwarg_or_double_starred:
    | NAME '=' expression
    | '**' expression

# ASSIGNMENT TARGETS
# ==================
```

(continua na próxima página)

(continuação da página anterior)

```
# Generic targets
# ---------------

# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
    | star_target !','
    | star_target (',' star_target )* [',']

star_targets_list_seq: ','.star_target+ [',']

star_targets_tuple_seq:
    | star_target (',' star_target )+ [',']
    | star_target ','

star_target:
    | '*' (!'*' star_target)
    | target_with_star_atom

target_with_star_atom:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | star_atom

star_atom:
    | NAME
    | '(' target_with_star_atom ')'
    | '(' [star_targets_tuple_seq] ')'
    | '[' [star_targets_list_seq] ']'

single_target:
    | single_subscript_attribute_target
    | NAME
    | '(' single_target ')'

single_subscript_attribute_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead

t_primary:
    | t_primary '.' NAME &t_lookahead
    | t_primary '[' slices ']' &t_lookahead
    | t_primary genexp &t_lookahead
    | t_primary '(' [arguments] ')' &t_lookahead
    | atom &t_lookahead

t_lookahead: '(' | '[' | '.'

# Targets for del statements
# --------------------------

del_targets: ','.del_target+ [',']

del_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | del_t_atom

del_t_atom:
    | NAME
    | '(' del_target ')'
```

(continua na próxima página)

(continuação da página anterior)

```
    | '(' [del_targets] ')'
    | '[' [del_targets] ']'

# TYPING ELEMENTS
# ---------------

# type_expressions allow */** but ignore them
type_expressions:
    | ','.expression+ ',' '*' expression ',' '**' expression
    | ','.expression+ ',' '*' expression
    | ','.expression+ ',' '**' expression
    | '*' expression ',' '**' expression
    | '*' expression
    | '**' expression
    | ','.expression+

func_type_comment:
    | NEWLINE TYPE_COMMENT &(NEWLINE INDENT)   # Must be followed by indented block
    | TYPE_COMMENT

# ========================= END OF THE GRAMMAR ===========================



# ========================= START OF INVALID RULES ========================
```

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 *Chamadas* 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 *Identificadores e palavras-chave*, 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:

```
callable(argument1, argument2, argumentN)
```

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: *Implementando descritores* 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 *Sintaxe de criação de listas, conjuntos e dicionários*.

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
A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

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 em Python assume 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 *codificação da localidade*.

localizador
Um objeto que tenta encontrar o *carregador* para um módulo que está sendo importado.

Desde o Python 3.3, 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`.

Veja **PEP 302**, **PEP 420** e **PEP 451** para mais informações.

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 *Definições de função*.

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 sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

A sintaxe de anotação de função é explicada na seção *Definições de função*.

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))         # sum of squares 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 *bloqueio global do interpretador*.

bloqueio 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 o GIL ao realizar tarefas computacionalmente muito intensas, como compactação ou cálculos de hash. Além disso, o 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 *Cached bytecode 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.

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 em 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 HowTo 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 bloqueios ou usando a abordagem EAFP.

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 da codificação da localidade.

Veja também *tratador de erros e codificação do sistema de arquivos*.

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
Um objeto que carrega um módulo. Deve definir um método chamado `load_module()`. Um carregador é normalmente devolvido por um *localizador*. Veja a **PEP 302** para detalhes e `importlib.abc.Loader` para um *classe base abstrata*.

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 A ordem de resolução de métodos do Python 2.3 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`.

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]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
1024
>>> isinstance(sys.float_info, tuple)    # kind of tuple
True
```

Algumas tuplas nomeadas são tipos embutidos (tal como os exemplos acima). Alternativamente, uma tupla nomeada pode ser criada a partir 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 com uma função fábrica `collections.namedtuple()`. A segunda técnica também adiciona 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 *posonly1* e *posonly2* a seguir:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *somente-nomeado*: especifica um argumento que pode ser passado para a função somente por nome. Parâmetros somente-nomeados podem ser definidos com um simples parâmetro var-posicional ou um `*` antes deles na lista de parâmetros na definição da função, por exemplo *kw_only1* and *kw_only2* a seguir:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-posicional*: especifica 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 *Definições de função* 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* 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 os idiomas mais comuns da linguagem Python, ao invés de implementar códigos usando conceitos comuns a outros idiomas. Por exemplo, um idioma comum em Python é fazer um loop sobre todos os elementos de uma iterável usando a instrução `for`. 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(food)):
    print(food[i])
```

Ao contrário do método limpo, ou então, Pythônico:

```
for piece in food:
    print(piece)
```

nome qualificado
Um nome pontilhado (quando 2 termos são ligados por um ponto) que mostra o "path" do escopo global de um módulo para uma classe, função ou método definido num determinado módulo, conforme definido pela **PEP 3155**. Para funções e classes de nível superior, o nome qualificado é o mesmo que o nome do objeto:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Quando usado para se referir a módulos, o *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 *imutável* arbitrária em vez de inteiros.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just *`__getitem__()`* and *`__len__()`*, adding `count()`, `index()`, *`__contains__()`*, and *`__reversed__()`*. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

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 *Sintaxe de criação de listas, conjuntos e dicionários*.

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.

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 *Nomes de métodos especiais*.

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*.

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 tipo 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)`.

tipo alias
Um sinônimo para um tipo, criado através da atribuição do tipo para um identificador.

Tipos alias são úteis para simplificar *dicas de tipo*. Por exemplo:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

pode tornar-se mais legível desta forma:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    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:
    field: 'annotation'
```

  Anotações de variáveis são normalmente usadas para *dicas de tipo*: por exemplo, espera-se que esta variável receba valores do tipo `int`:

```
count: int = 0
```

  A sintaxe de anotação de variável é explicada na seção *instruções de atribuição anotado*.

  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 3.12.2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation␣
↪("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise␣
↪using Python
   3.12.2 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 3.12.2 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 3.12.2 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 3.12.2 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
   3.12.2.

4. PSF is making Python 3.12.2 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 3.12.2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.12.2
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A␣
↪RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.12.2, 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 3.12.2, 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
```

(continua na próxima página)

(continuação da página anterior)

```
   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.

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
```

(continua na próxima página)

(continuação da página anterior)

```
   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 3.12.2

```
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
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
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
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

O módulo `uu` contém o seguinte aviso:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  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
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/

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

Os módulos `hashlib`, `posix`, `ssl`, `crypt` usam a biblioteca OpenSSL para desempenho adicional se forem disponibilizados pelo sistema operacional. Além disso, os instaladores do Windows e do Mac OS X para Python podem incluir uma cópia das bibliotecas do OpenSSL, portanto incluímos uma cópia da licença do OpenSSL aqui: Para o lançamento do OpenSSL 3.0, e lançamentos posteriores derivados deste, se aplica a Apache License v2:

```
                                 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
   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
```

(continua na próxima página)

(continuação da página anterior)

```
   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,
   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
```

(continua na próxima página)

(continuação da página anterior)

```
      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,
   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
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.

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
```

(continua na próxima página)

(continuação da página anterior)

```
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.
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.19 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

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
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
```

(continua na próxima página)

(continuação da página anterior)

```
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

Veja: *História e Licença* para informações completas de licença e permissões.

Índice

Não alfabético

A

B

C

D

E

F

G

H

N

O

Q

R

S

T

U

V

W

X

Y

Z