The Python Language Reference

Release 3.13.2

Guido van Rossum and the Python development team

março 25, 2025

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*, gerados pelo *analisador léxico* (também conhecido como *tokenizador*). 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: <nome-codificação> -*-
```

que é reconhecido também por GNU Emacs, e

```
# vim:fileencoding=<nome-codificação>
```

que é reconhecido pelo VIM de Bram Moolenaar.

Se nenhuma codificação é declarada, a codificação padrão é UTF-8. Se a codificação implícita ou explícita de um arquivo é UTF-8, uma marca inicial de ordem de byte UTF-8 (b'xefxbbxbf') será ignorada em vez de ser um erro de sintaxe.

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:   # Parece ser uma data válida
        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',      # Estes são os
               'April',   'Mei',      'Juni',       # nomes holandeses
               'Juli',    'Augustus', 'September',  # para os meses
               'Oktober', 'November', 'December']   # do ano
```

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):
        # Calcula a lista de todas as permutações de 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):                       # erro: primeira linha indentada
for i in range(len(l)):             # erro: não indentada
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # erro: indentação inesperada
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # erro: desindentação inconsistente
```

(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 incluem as letras maiúsculas e minúsculas `A` a `Z`, o sublinhado _ e, exceto pelo primeiro caractere, os dígitos `0` a `9`. O Python 3.0 introduziu caracteres adicionais de fora do intervalo ASCII (veja **PEP 3131**). Para esses caracteres, a classificação usa a versão do Unicode Character Database conforme incluído 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 underscore, a
id_continue  ::= <all characters in id_start, plus characters in the categories Mn, Mc, Nd, Pc a
xid_start    ::= <all characters in id_start whose NFKC normalization is in "id_start xid_contir
xid_continue ::= <all characters in id_continue whose NFKC normalization is in "id_continue*">
```

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 15.1.0 pode ser encontrado em https://www.unicode.org/Public/15.1.0/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

Adicionado 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'`; tais construções são chamadas de *literais de strings brutas* e *literais de bytes brutos* tratam as contrabarras como caracteres literais. Como resultado, em literais de string, os escapes `'\U'` e `'\u'` em strings brutas não são tratados de maneira especial.

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

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 *Literais de strings formatadas*. 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:

```
>>> 'Esta string não vai incluir \
... contrabarras e caracteres de nova linha.'
'Esta string não vai incluir contrabarras e caracteres de nova linha.'
```

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_]"       # letra ou sublinhado
           "[A-Za-z0-9_]*"   # letra, dígito ou sublinhado
          )
```

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.

2.4.3 Literais de strings formatadas

Adicionado 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 | replacement_field)*
literal_char      ::= <any code point except "{", "}" or NULL>
```

[1] https://www.unicode.org/Public/15.1.0/ucd/NameAliases.txt

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 # Este é um comentário }"
... + 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.

Adicionado 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:

```
>>> nome = "Fred"
>>> f"Ele falou que o nome dele é {nome!r}."
"Ele falou que o nome dele é 'Fred'."
>>> f"Ele falou que o nome dele é {repr(nome)}."  # repr() é um equivalente a !r
"Ele falou que o nome dele é 'Fred'."
>>> largura = 10
>>> precisão = 4
>>> valor = decimal.Decimal("12.34567")
>>> f"resultado: {valor:{largura}.{precisão}}"  # campos aninhados
'resultado: 12.35'
>>> hoje = datetime(year=2017, month=1, day=27)
>>> f"{hoje:%B %d, %Y}"  # usando especificador de formato de data
'January 27, 2017'
```

(continua na próxima página)

(continuação da página anterior)

```
>>> f"{hoje=:%B %d, %Y}"  # usando especificador de formato de data e depuração
'hoje=January 27, 2017'
>>> número = 1024
>>> f"{número:#0x}"  # usando especificador de formato de número inteiro
'0x400'
>>> foo = "bar"
>>> f"{ foo = }"  # preserva espaço em branco
" foo = 'bar'"
>>> linha = "Olha a caixa d'água"
>>> f"{linha = }"
'linha = "Olha a caixa d\'água"'
>>> f"{linha = :20}"
"linha = Olha a caixa d'água "
>>> f"{linha = !r:20}"
'linha = "Olha a caixa d\'água"'
```

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"A lista a contém:\n{"\n".join(a)}")
A lista a contém:
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"Não é uma 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.

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 comprimento?") 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.

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.

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

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

Os tipos afetam quase todos os aspectos do comportamento do objeto. Até mesmo a importância da identidade do objeto é afetada em algum sentido: para tipos imutáveis, as operações que calculam novos valores podem realmente retornar uma referência a qualquer objeto existente com o mesmo tipo e valor, enquanto para objetos mutáveis isso não é permitido. Por exemplo, após `a = 1; b = 1`, *a* e *b* podem ou não se referir ao mesmo objeto com o valor um, dependendo da implementação. Isto ocorre porque `int` é um tipo imutável, então a referência a `1` pode ser reutilizada. Este comportamento depende da implementação usada, então não deve ser considerada confiável, mas é algo para se estar ciente ao fazer uso de testes de identidade de objeto. No entanto, após `c = []; d = []`, *c* e *d* têm a garantia de referir-se a duas listas vazias diferentes e únicas. (Observe que `e = f = []` atribui o *mesmo* objeto para *e* e *f*.)

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 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 descontinuada. Embora atualmente seja avaliada como verdadeiro, é emitida uma exceção `DeprecationWarning`. 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 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 presumida 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]`. Algumas sequências, incluindo sequências embutidas, interpretam subscritos negativos adicionando o comprimento da sequência. Por exemplo, `a[-2]` é igual a `a[n-2]`, o penúltimo item da sequência a com comprimento `n`.

Sequências também provê 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. O comentário acima sobre índices negativos também se aplica a posições de fatias negativas.

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.

Atributos especiais de somente leitura

Atributo	Significado
`function.__globals__`	Uma referência ao `dicionário` que contém as *variáveis globais* da função – o espaço de nomes global do módulo no qual a função foi definida.
`function.__closure__`	`None` ou uma `tuple` de células que contêm ligação para os nomes especificados no atributo `co_freevars` do `objeto código` da função. Um objeto de célula tem o atributo `cell_contents`. Isso pode ser usado para obter o valor da célula, bem como definir o valor.

Atributos especiais graváveis

A maioria desses atributos verifica o tipo do valor atribuído:

Atributo	Significado
function.__doc__	A string de documentação da função, ou None se indisponível.
function.__name__	O nome da função. Veja também: atributos __name__.
function.__qualname__	O *nome qualificado* da função. Veja também: atributos __qualname__. Adicionado na versão 3.3.
function.__module__	O nome do módulo em que a função foi definida ou None se indisponível.
function.__defaults__	Uma tuple contendo valores de *parâmetro* padrão para aqueles parâmetros que possuem padrões, ou None se nenhum parâmetro tiver um valor padrão.
function.__code__	O *objeto código* que representa o corpo da função compilada.
function.__dict__	O espaço de nomes que provvê atributos de função arbitrários. Veja também: *atributos __dict__*.
function.__annotations__	Um dicionário contendo anotações de *parâmetros*. As chaves do dicionário são os nomes dos parâmetros e 'return' para a anotação de retorno, se fornecida. Veja também: annotations-howto.
function.__kwdefaults__	Um dicionário contendo padrões apenas para *parâmetros* somente-nomeados.
function.__type_params__	Uma tuple contendo os *parâmetros de tipo* de uma *função genérica*. Adicionado na versão 3.12.

Os objetos de função também dão suporte à obtenção e definição de atributos arbitrários, que podem ser usados, por exemplo, para anexar metadados a funções. A notação de ponto de atributo regular é usada para obter e definir tais atributos.

A implementação atual do CPython provê apenas atributos de função em funções definidas pelo usuário. Atributos de função em *funções embutido* podem ser suportados no futuro.

Informações adicionais sobre a definição de uma função podem ser obtidas de seu *objeto código* (acessível através do atributo *__code__*).

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).

Atributos especiais de somente leitura:

method.__self__	Refere-se ao objeto instância da classe ao qual o método é *vinculado*
method.__func__	Refere-se ao *objeto função* original
method.__doc__	A documentação do método (igual a *method.__func__.__doc__*). Um string se a função original tivesse uma docstring, caso contrário None.
method.__name__	O nome do método (mesmo que *method.__func__.__name__*)
method.__module__	O nome do módulo em que o método foi definido ou None se indisponível.

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

Objetos método definidos pelo usuário podem ser criados ao obter um atributo de uma classe (talvez através de uma instância dessa classe), se esse atributo for um *objeto função* definido pelo usuário ou um objeto classmethod .

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

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

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

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

É importante observar que funções definidas pelo usuário que são atributos de uma instância de classe não são convertidas em métodos vinculados; isso *somente* acontece quando a função é um atributo da classe.

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 *iterador assíncrono* que pode ser usado em uma instrução *async for* para executar o corpo da função.

Chamar o método *aiterator.__anext__* do iterador assíncrono retornará um *aguardável* que, quando aguardado, será executado até fornecer um valor usando a expressão *yield*. Quando a função executa uma instrução vazia *return* ou 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

Um objeto função embutida é um wrapper em torno de uma função C. Exemplos de funções embutidas são `len()` e `math.sin()` (`math` é um módulo embutido padrão). O número e o tipo dos argumentos são determinados pela função C. Atributos especiais de somente leitura:

- `__doc__` é a string de documentação da função, ou `None` se não estiver disponível. Veja *function.__doc__*.
- `__name__` é o nome da função. Veja *function.__name__*.
- `__self__` é definido para `None` (mas veja o próximo item).
- `__module__` é o nome do módulo no qual a função foi definida ou `None` se não estiver disponível. Veja *function.__module__*.

Métodos embutidos

Este é realmente um disfarce diferente de uma função embutida, desta vez contendo um objeto passado para a função C como um argumento extra implícito. Um exemplo de método embutido é `alist.append()`, presumindo que *alist* é um objeto de lista. Nesse caso, o atributo especial de somente leitura `__self__` é definido como o objeto denotado por *alist*. (O atributo tem a mesma semântica de *outros métodos de instância*.)

Classes

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

Instâncias de classe

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

3.2.9 Módulos

Módulos são uma unidade organizacional básica do código Python, e são criados pelo *sistema de importação* quando invocado pela instrução *import*, ou chamando funções como `importlib.import_module()` e a embutida `__import__()`. Um objeto módulo tem um espaço de nomes implementado por um objeto `dicionário` (este é o dicionário referenciado pelo atributo *__globals__* das funções definidas no módulo). As referências de atributos são traduzidas para pesquisas neste dicionário, por exemplo, `m.x` é equivalente a `m.__dict__["x"]`. Um objeto módulo não contém o objeto código usado para inicializar o módulo (uma vez que não é necessário depois que a inicialização é concluída).

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 relacionados à importação em objetos de módulo

Objetos de módulo têm os seguintes atributos que se relacionam ao *sistema de importação*. Quando um módulo é criado usando o maquinário associado ao sistema de importação, esses atributos são preenchidos com base no *spec* do módulo, antes que o *carregador* execute e carregue o módulo.

Para criar um módulo dinamicamente em vez de usar o sistema de importação, é recomendado usar `importlib.util.module_from_spec()`, que definirá os vários atributos controlados pela importação para valores apropriados. Também é possível usar o construtor `types.ModuleType` para criar módulos diretamente, mas essa técnica é mais propensa a erros, pois a maioria dos atributos deve ser definida manualmente no objeto do módulo após ele ter sido criado ao usar essa abordagem.

Cuidado

Com exceção de *__name__*, é **fortemente** recomendado que você confie no *__spec__* e seus atributos em vez de qualquer um dos outros atributos individuais listados nesta subseção. Observe que atualizar um atributo em `__spec__` não atualizará o atributo correspondente no próprio módulo:

```
>>> import typing
>>> typing.__name__, typing.__spec__.name
('typing', 'typing')
>>> typing.__spec__.name = 'spelling'
>>> typing.__name__, typing.__spec__.name
('typing', 'spelling')
>>> typing.__name__ = 'keyboard_smashing'
>>> typing.__name__, typing.__spec__.name
('keyboard_smashing', 'spelling')
```

`module.`**`__name__`**

O nome usado para identificar exclusivamente o módulo no sistema de importação. Para um módulo executado diretamente, isso será definido como `"__main__"`.

Este atributo deve ser definido como o nome totalmente qualificado do módulo. Espera-se que ele corresponda ao valor de `module.__spec__.name`.

`module.`**`__spec__`**

Um registro do estado relacionado ao sistema de importação do módulo.

Define com `spec de módulo` que foi usado ao importar o módulo. Veja *Especificações de módulo* para mais detalhes.

Adicionado na versão 3.4.

`module.`**`__package__`**

O *pacote* ao qual um módulo pertence.

Se o módulo for de nível superior (ou seja, não fizer parte de nenhum pacote específico), o atributo deve ser definido como `''` (a string vazia). Caso contrário, deve ser definido como o nome do pacote do módulo (que pode ser igual a *module.__name__* se o módulo em si for um pacote). Veja **PEP 366** para mais detalhes.

Este atributo é usado em vez de *__name__* para calcular importações relativas explícitas para módulos principais. O padrão é `None` para módulos criados dinamicamente usando o construtor `types.ModuleType`; use `importlib.util.module_from_spec()` em vez disso para garantir que o atributo seja definido como `str`.

É **fortemente** recomendado que você use `module.__spec__.parent` em vez de `module.__package__`. *__package__* agora só é usado como fallback se `__spec__.parent` não estiver definido, e esse caminho de fallback está descontinuado.

Alterado na versão 3.4: Este atributo agora presume o padrão `None` para módulos criados dinamicamente usando o construtor `types.ModuleType`. Anteriormente, o atributo era opcional.

Alterado na versão 3.6: Espera-se que o valor de `__package__` seja o mesmo que `__spec__.parent`. *__package__* agora é usado apenas como fallback durante a resolução de importação se `__spec__.parent` não estiver definido.

Alterado na versão 3.10: `ImportWarning` é levantada se uma resolução de importação retorna para `__package__` em vez de `__spec__.parent`.

Alterado na versão 3.12: Levanta `DeprecationWarning` em vez de `ImportWarning` ao retornar para `__package__` durante a resolução de importação.

Deprecated since version 3.13, will be removed in version 3.15: `__package__` deixará de ser definido ou levado em consideração pelo sistema de importação ou biblioteca padrão.

module.**__loader__**

O objeto *carregador* que o maquinário de importação usou para carregar o módulo.

Este atributo é útil principalmente para introspecção, mas pode ser usado para funcionalidades adicionais específicas do carregador, por exemplo, para obter dados associados a um carregador.

__loader__ assume como padrão None para módulos criados dinamicamente usando o construtor types.ModuleType; use importlib.util.module_from_spec() para garantir que o atributo seja definido como um objeto *carregador*.

É **fortemetne** recomendado que você use module.__spec__.loader em vez de module.__loader__.

Alterado na versão 3.4: Este atributo agora presume o padrão None para módulos criados dinamicamente usando o construtor types.ModuleType. Anteriormente, o atributo era opcional.

Deprecated since version 3.12, will be removed in version 3.16: A definição __loader__ em um módulo enquanto falha na definição de __spec__.loader está descontinuado. No Python 3.16, __loader__ deixará de ser definido ou levado em consideração pelo sistema de importação ou pela biblioteca padrão.

module.**__path__**

Uma *sequência* (possivelmente vazia) de strings enumerando os locais onde os submódulos do pacote serão encontrados. Módulos que não sejam de pacote não devem ter um atributo __path__. Veja *Atributo __path__ dos módulos* para mais detalhes.

É **fortemente** recomendado que você use module.__spec__.submodule_search_locations em vez de module.__path__.

module.**__file__**

module.**__cached__**

__file__ e __cached__ são atributos opcionais que podem ou não ser definidos. Ambos os atributos devem ser um str quando estiverem disponíveis.

__file__ indica o nome do caminho do arquivo do qual o módulo foi carregado (se carregado de um arquivo) ou o nome do caminho do arquivo da biblioteca compartilhada para módulos de extensão carregados dinamicamente de uma biblioteca compartilhada. Pode estar faltando para certos tipos de módulos, como módulos C que estão estaticamente vinculados ao interpretador, e o *sistema de importação* pode optar por deixá-lo sem definição se não tiver significado semântico (por exemplo, um módulo carregado de um banco de dados).

Se __file__ estiver definido então o atributo __cached__ também pode ser definido, que é o caminho para qualquer versão compilada do código (por exemplo, um arquivo compilado por byte). O arquivo não precisa existir para configurar esse atributo; o caminho pode simplesmente apontar para onde o arquivo compilado existiria (veja **PEP 3147**).

Observe que __cached__ pode ser definido mesmo se __file__ não estiver definido. No entanto, esse cenário é bastante atípico. Em última análise, o *carregador* é o que faz uso do spec de módulo fornecido pelo *localizador* (do qual __file__ e __cached__ são derivados). Portanto, se um carregador puder carregar a partir de um módulo em cache, mas não carregar a partir de um arquivo, esse cenário atípico poderá ser apropriado.

É **fortemente** recomendado que você use module.__spec__.cached em vez de module.__cached__.

Deprecated since version 3.13, will be removed in version 3.15: A definição __cached__ em um módulo enquanto falha na definição de __spec__.cached está descontinuado. No Python 3.15, __cached__ deixará de ser definido ou levado em consideração pelo sistema de importação ou pela biblioteca padrão.

Outros atributos graváveis em objetos de módulo

Além dos atributos relacionados à importação listados acima, os objetos de módulo também têm os seguintes atributos graváveis:

module.**__doc__**

A string de documentação do módulo, ou None se indisponível. Veja também: atributos __doc__.

module.**__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.

Dicionários do módulo

Os objetos de módulo também têm o seguinte atributo especial somente leitura:

module.**__dict__**

O espaço de nomes do módulo como um objeto dicionário. Exclusivamente entre os atributos listados aqui, `__dict__` não pode ser acessado como uma variável global de dentro de um módulo; ele só pode ser acessado como um atributo em objetos de módulo.

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 em losango, onde há vários caminhos de herança que levam de volta a um ancestral comum. Detalhes adicionais sobre a ordem de resolução de métodos (MRO) C3 usado pelo Python podem ser encontrados em python_2.3_mro.

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

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

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

Atributos especiais

Atributo	Significado
`type.__name__`	O nome da classe. Veja também: `atributos __name__`.
`type.__qualname__`	O *nome qualificado* da classe. Veja também: `atributos __qualname__`.
`type.__module__`	O nome do módulo no qual a classe foi definida.
`type.__dict__`	Um `proxy de mapeamento` fornecendo uma visão somente leitura do espaço de nomes da classe. Veja também: *`atributos __dict__`*.
`type.__bases__`	Uma `tuple` contendo as bases da classe. Na maioria dos casos, para uma classe definida como `class X(A, B, C)`, `X.__bases__` será exatamente igual a `(A, B, C)`.
`type.__doc__`	A string de documentação da classe, ou `None` se não estiver definida. Não herdado por subclasses.
`type.__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. **Cuidado** Acessar o atributo `__annotations__` de um objeto classe diretamente pode produzir resultados incorretos na presença de metaclasses. Além disso, o atributo pode não existir para algumas classes. Use `inspect.get_annotations()` para recuperar anotações de classe com segurança.
`type.__type_params__`	Uma `tuple` contendo os *parâmetros de tipo* de uma *classe genérica*. Adicionado na versão 3.12.
`type.__static_attributes__`	Uma `tuple` contendo nomes de atributos dessa classe que são atribuídos por meio de `self.X` de qualquer função em seu corpo. Adicionado na versão 3.13.
`type.__firstlineno__`	O número da linha da primeira linha da definição de classe, incluindo decoradores. Definir o atributo `__module__` remove o item `__firstlineno__` do dicionário do tipo. Adicionado na versão 3.13.
`type.__mro__`	A `tuple` de classes que são consideradas ao procurar por classes bases durante resolução de métodos.

Métodos especiais

Além dos atributos especiais descritos acima, todas as classes Python também têm os dois métodos a seguir disponíveis:

`type.`**`mro`**`()`

Este método pode ser substituído por uma metaclasse para personalizar a ordem de resolução de métodos para suas instâncias. Ele é chamado na instanciação da classe, e o seu resultado é armazenado em *`__mro__`*.

`type.`**`__subclasses__`**`()`

Cada classe mantém uma lista de referências fracas para suas subclasses imediatas. Este método retorna uma lista de todas essas referências ainda vivas. A lista está na ordem que são definidas. Exemplo:

```
>>> class A: pass
>>> class B(A): pass
>>> A.__subclasses__()
[<class 'B'>]
```

3.2.11 Instâncias de classe

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

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

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

Atributos especiais

`object.`**`__class__`**

A classe à qual pertence uma instância de classe.

`object.`**`__dict__`**

Um dicionário ou outro objeto de mapeamento usado para armazenar atributos (graváveis) de um objeto. Nem todas as instâncias têm um atributo `__dict__`; veja a seção sobre *`__slots__`* para mais detalhes.

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.

Atributos especiais de somente leitura

codeobject.**co_name**	O nome da função
codeobject.**co_qualname**	O nome completo da função Adicionado na versão 3.11.
codeobject.**co_argcount**	O número total de *parâmetros* posicionais (incluindo parâmetros somente-posicionais e parâmetros com valores padrão) que a função possui
codeobject.**co_posonlyargcount**	O número de *parâmetros* somente-posicionais (incluindo argumentos com valores padrão) que a função possui
codeobject.**co_kwonlyargcount**	O número de *parâmetros* somente-nomeados (incluindo argumentos com valores padrão) que a função possui
codeobject.**co_nlocals**	O número de *variáveis locais* usadas pela função (incluindo parâmetros)
codeobject.**co_varnames**	Uma `tuple` contendo os nomes das variáveis locais na função (começando com os nomes dos parâmetros)
codeobject.**co_cellvars**	Uma `tuple` contendo os nomes de *variáveis locais* que são referenciadas a partir de pelo menos um *escopo aninhado* de dentro da função
codeobject.**co_freevars**	Uma `tuple` contendo os nomes de *variáveis livres (de clausura)* que um *escopo aninhado* referencia em um escopo externo. Veja também *function.__closure__*. Observação: referências a nomes globais e embutidos *não* estão incluídas.
codeobject.**co_code**	Uma string representando a sequência de instruções *bytecode* na função
codeobject.**co_consts**	Um `tuple` contendo os literais usados pelo *bytecode* na função
codeobject.**co_names**	Um `tuple` contendo os nomes usados pelo *bytecode* na função
codeobject.**co_filename**	O nome do arquivo do qual o código foi compilado
codeobject.**co_firstlineno**	O número da linha da primeira linha da função
codeobject.**co_lnotab**	Uma string que codifica o mapeamento de *bytecode* compensa para números de linha. Para obter detalhes, consulte o código-fonte do interpretador. Descontinuado desde a versão 3.12: Este atributo de objetos código está descontinuado e pode ser removido no Python 3.15.
codeobject.**co_stacksize**	O tamanho de pilha necessário do objeto código
codeobject.**co_flags**	Um número `inteiro` codificando uma série de sinalizadores para o interpretador.

Os seguintes bits sinalizadores são definidos para *co_flags*: o bit `0x04` é definido se a função usa a sintaxe `*arguments` para aceitar um número arbitrário de argumentos posicionais; o bit `0x08` é definido se a função usa a sintaxe `**keywords` para aceitar argumentos nomeados arbitrários; o bit `0x20` é definido se a função for um gerador. Veja inspect-module-co-flags para detalhes na semântica de cada sinalizadores que podem estar presentes.

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

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

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

Métodos de objetos código

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

Retorna um iterável das posições no código-fonte de cada instrução *bytecode* no objeto código.

O iterador retorna `tuple`s contendo `(start_line, end_line, start_column, end_column)`. A *i-nésima* tupla corresponde à posição do código-fonte que compilou para a *i-nésima* unidade de código. As informações da coluna são deslocamentos de bytes utf-8 indexados em 0 na linha de código fornecida.

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`.

Adicionado 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`**`()`

Retorna um iterador que produz informações sobre intervalos sucessivos de *bytecode*s. Cada item gerado é uma `tuple` de `(start, end, lineno)`:

- `start` (um `int`) representa o deslocamento (inclusivo) do início do intervalo *bytecode*
- `end` (um `int`) representa o deslocamento (exclusivo) do fim do intervalo *bytecode*
- `lineno` é um `int` representando o número da linha do intervalo do *bytecode*, ou `None` se os bytecodes no intervalo fornecido não tiverem número de linha

Os itens gerados terão as seguintes propriedades:

- O primeiro intervalo gerado terá um `start` de 0.
- Os intervalos `(start, end)` serão não decrescentes e consecutivos. Ou seja, para qualquer par de `tuple`s, o `start` do segundo será igual ao `end` do primeiro.
- Nenhum intervalo será inverso: `end >= start` para todos os trios.

- A última `tuple` gerada terá `end` igual ao tamanho do *bytecode*.

Intervalos de largura zero, onde `start == end`, são permitidos. Intervalos de largura zero são usados para linhas que estão presentes no código-fonte, mas foram eliminadas pelo compilador de *bytecode*.

Adicionado na versão 3.10.

Ver também

PEP 626 - Números de linha precisos para depuração e outras ferramentas.
A PEP que introduziu o método `co_lines()`.

`codeobject.`**`replace`**`(`*`**kwargs`*`)`

Retorna uma cópia do objeto de código com novos valores para os campos especificados.

Objetos de código também são suportados pela função genérica `copy.replace()`.

Adicionado na versão 3.8.

Objetos quadro

Objetos quadro representam quadros de execução. Eles podem ocorrer em *objetos traceback* e também são passados para funções de rastreamento registradas.

Atributos especiais de somente leitura

`frame.`**`f_back`**	Aponta para o quadro de pilha anterior (em direção ao chamador), ou `None` se este for o quadro de pilha mais abaixo.
`frame.`**`f_code`**	O *objeto código* sendo executado neste quadro. Acessar este atributo levanta um evento de auditoria `object.__getattr__` com os argumentos `obj` e `"f_code"`.
`frame.`**`f_locals`**	O mapeamento usado pelo quadro para procurar *variáveis locais*. Se o quadro se referir a um *escopo otimizado*, isso pode retornar um objeto proxy write-through. Alterado na versão 3.13: Retorna um proxy para escopos otimizados.
`frame.`**`f_globals`**	O dicionário usado pelo quadro para procurar *variáveis globais*
`frame.`**`f_builtins`**	O dicionário usado pelo quadro para procurar *nomes embutidos (intrínsecos)*
`frame.`**`f_lasti`**	A "instrução precisa" do objeto quadro (este é um índice na string *bytecode* do *objeto código*)

Atributos especiais graváveis

`frame.`**`f_trace`**	Se não for `None`, esta é uma função chamada para vários eventos durante a execução do código (isso é usado por depuradores). Normalmente, um evento é disparado para cada nova linha de origem (veja *`f_trace_lines`*).
`frame.`**`f_trace_lines`**	Defina este atributo como `False` para desabilitar o acionamento de um evento de rastreamento para cada linha de origem.
`frame.`**`f_trace_opcodes`**	Defina este atributo para `True` para permitir que eventos por opcode sejam solicitados. Observe que isso pode levar a um comportamento indefinido do interpretador se as exceções levantadas pela função de rastreamento escaparem para a função que está sendo rastreada.
`frame.`**`f_lineno`**	O número da linha atual do quadro – escrever para isso de dentro de uma função de rastreamento faz saltar para a linha dada (apenas para o quadro mais abaixo). Um depurador pode implementar um comando Jump (também conhecido como Set Next Statement) escrevendo para esse atributo.

Métodos de objetos quadro

Objetos quadro têm suporte a um método:

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

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

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

Adicionado na versão 3.4.

Alterado na versão 3.13: Tentar limpar um quadro suspenso levanta `RuntimeError` (como sempre foi o caso para quadros em execução).

Objetos traceback

Objetos traceback representam o stack trace (situação da pilha de execução) de uma exceção. Um objeto traceback é criado implicitamente quando ocorre uma exceção e também pode ser criado explicitamente chamando `types.TracebackType`.

Alterado na versão 3.7: Objetos traceback agora podem ser instanciados explicitamente a partir de código Python.

Para tracebacks criados implicitamente, quando a busca por um manipulador de exceção desenrola a pilha de execução, em cada nível desenrolado um objeto traceback é inserido na frente do traceback atual. Quando um manipulador de exceção é inserido, o stack trace é disponibilizado para o programa. (Veja a seção *A instrução try*.) É acessível como o terceiro item da tupla retornada por `sys.exc_info()`, e como o atributo `__traceback__` da exceção capturada.

Quando o programa não contém um manipulador adequado, o stack trace é escrito (formatado de maneira adequada) no fluxo de erro padrão; se o interpretador for interativo, ele também é disponibilizado ao usuário como `sys.last_traceback`.

Para tracebacks criados explicitamente, cabe ao criador do traceback determinar como os atributos *`tb_next`* devem ser vinculados para formar um stack trace completo.

Atributos especiais de somente leitura:

traceback.**tb_frame**	Aponta para o *quadro* de execução do nível atual. Acessar este atributo levanta um evento de auditoria `object.__getattr__` com os argumentos `obj` e `"tb_frame"`.
traceback.**tb_lineno**	Fornece o número da linha onde ocorreu a exceção
traceback.**tb_lasti**	Indica a "instrução precisa".

O número da linha e a última instrução no traceback podem diferir do número da linha do seu *objeto quadro* se a exceção ocorreu em uma instrução *try* sem cláusula except correspondente ou com uma cláusula *finally*.

traceback.**tb_next**

O atributo especial de escrita `tb_next` é o próximo nível no stack trace (em direção ao quadro onde a exceção ocorreu), ou `None` se não houver próximo nível.

Alterado na versão 3.7: Este atributo agora é gravável

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

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

3.3 Nomes de métodos especiais

Uma classe pode implementar certas operações que são chamadas por sintaxe especial (como operações aritméticas ou 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 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.

[2] Os métodos *__hash__()*, *__iter__()*, *__reversed__()* e *__contains__()*, *__class_getitem__()* e `__fspath__()` têm um tratamento especial para isso. Outros ainda irão levantar um `TypeError`, mas podem fazer isso contando com o comportamento de que `None` não é chamável.

Não há garantia de que os métodos *__del__()* sejam chamados para objetos que ainda existem quando o interpretador sai. `weakref.finalize` fornece uma maneira direta de registrar uma função de limpeza a ser chamada quando um objeto é coletado como lixo.

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.

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. Uma implementação padrão é fornecida pela própria classe `object`.

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

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

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`. A classe `object` em si não fornece este método.

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.

A implementação padrão pela classe `object` deve receber uma string *format_spec* vazia. Ela delega para *__str__()*.

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()`.

Por padrão, a classe `object` fornece implementações consistentes com *Comparações de valor*: igualdade compara de acordo com a identidade do objeto, e comparações de ordem levantam `TypeError`. Cada método padrão pode gerar esses resultados diretamente, mas também pode retornar `NotImplemented`.

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.

Quando nenhum método apropriado retorna qualquer valor diferente de `NotImplemented`, os operadores `==` e `!=` retornarão para `is` e `is not`, respectivamente.

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 (herdados da classe `object`); 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 substitui *`__eq__()`* precisa manter a implementação de *`__hash__()`* de uma classe base, o interpretador deve ser informado disso explicitamente pela configuração `__hash__ = <ClasseBase>.__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.

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

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 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__()` (o que é verdadeiro da classe `object` em si), 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`. A classe `object` em si não fornece este método.

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 obter 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*.

Para acessos a certos atributos sensíveis, levanta um evento de auditoria `object.__getattr__` com os argumentos `obj` e `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)`.

Para atribuições de certos atributos sensíveis, levanta um evento de auditoria `object.__setattr__` com os argumentos `obj`, `name` e `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.

Para exclusões a certos atributos sensíveis, levanta um evento de auditoria `object.__delattr__` com os argumentos `obj` e `name`.

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

Chamado quando `dir()` é chamado com o objeto como argumento. Um iterável deve ser retornada. `dir()` converte o iterável retornado em uma lista e o 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 um iterável 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.

Adicionado 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. A classe `object` em si não implementa quaisquer desses protocolos.

`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, instâncias de uma subclasse filha vai receber um *__dict__* e *__weakref__* a menos que a subclasse também defina *__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.

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

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

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)`.

Adicionado 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()  # Automaticamente chama: 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()
```

(continua na próxima página)

(continuação da página anterior)

```
A.x = c                  # O gancho não é chamado
c.__set_name__(A, 'x')   # Invoca manualmente o gancho
```

Consulte *Criando o objeto classe* para mais detalhes.

Adicionado 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.

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 travamento/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.

`type.`**`__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)`.

`type.`**`__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)

    # Se a classe de  obj define __getitem__,
    # chama 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)

    # Do contrário, se obj for uma classe e define __class_getitem__,
    # chama obj.__class_getitem__(x)
    elif isclass(obj) and hasattr(obj, '__class_getitem__'):
        return obj.__class_getitem__(x)

    # Do contrário, levanta uma exceção
    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__()`*:

```
>>> # lista tem a classe "type" como sua metaclasse, como a maioria das classes:
>>> type(list)
<class 'type'>
>>> type(dict) == type(list) == type(tuple) == type(str) == type(bytes)
True
>>> # "list[int]" chama "list.__class_getitem__(int)"
>>> list[int]
list[int]
>>> # list.__class_getitem__ retorna um objeto GenericAlias:
>>> 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'
...
>>> # Classes Enum têm uma metaclasses personalizada:
>>> type(Menu)
<class 'enum.EnumMeta'>
>>> # EnumMeta define __getitem__,
>>> # de forma que __class_getitem__ não é chamado,
>>> # e o resultado não é um objeto GenericAlias:
>>> 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, ...)`. A classe `object` em si não fornece este método.

3.3.7 Emulando tipos contêineres

Os métodos a seguir podem ser definidos para implementar objetos contêineres. Nenhum deles são fornecidos pela classe `object` em si. Os contêineres geralmente são *sequências* (tal como `listas` ou `tuplas`) ou *mapeamentos* (com *dicionários*), mas também podem representar outros contêineres. O primeiro conjunto de métodos é usado para emular uma sequência ou para emular um mapeamento; a diferença é que, para uma sequência, as chaves permitidas devem ser os inteiros *k* para os quais `0 <= k < N` onde *N* é o comprimento da sequência, ou objetos `slice`, que definem um intervalo de itens. Também é recomendado que os mapeamentos forneçam os métodos `keys()`, `values()`, `items()`, `get()`, `clear()`, `setdefault()`, `pop()`, `popitem()`, `copy()` e `update()` se comportando de forma semelhante aos objetos `dicionário` padrão do Python. O módulo `collections.abc` fornece uma *classe base abstrata* `MutableMapping` para ajudar a criar esses métodos a partir de um conjunto base de *`__getitem__()`*, *`__setitem__()`*, *`__delitem__()`* e `keys()`. Sequências mutáveis devem fornecer métodos `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()` e `sort()`, como objetos `list` padrão do Python. Finalmente, os tipos sequência devem implementar adição (significando concatenação) e multiplicação (significando repetição) definindo os métodos *`__add__()`*, *`__radd__()`*, *`__iadd__()`*, *`__mul__()`*, *`__rmul__()`* e *`__imul__()`* descritos abaixo; eles não devem definir outros operadores numéricos. É recomendado que ambos os mapeamentos e sequências implementem o método *`__contains__()`* para permitir o uso eficiente do operador `in`; para mapeamentos, `in` deve pesquisar as chaves do mapeamento; para sequências, ele deve pesquisar os valores. É ainda recomendado que ambos os mapeamentos e sequências implementem o método *`__iter__()`* para permitir a iteração eficiente através do contêiner; para mapeamentos, `__iter__()` deve iterar através das chaves do objeto; para sequências, ele deve iterar por meio dos valores.

`object.__len__`(*self*)

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

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

`object.__length_hint__`(*self*)

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.

Adicionado na versão 3.4.

Nota

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

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

é traduzida com

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

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

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

Chamado para implementar a avaliação de `self[key]`. Para tipos de *sequência*, as chaves aceitas devem ser inteiros. Opcionalmente, eles também podem oferecer suporte a objetos `slice`. Suporte a índice negativo também é opcional. Se *key* for de um tipo impróprio, `TypeError` pode ser levantada; se *key* for de um valor fora do conjunto de índices para a sequência (após qualquer interpretação especial de valores negativos), `IndexError` deve ser levantada. Para tipos *mapeamento*, se *key* estiver faltando (não no contêiner), `KeyError` deve ser levantada.

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

> **Nota**
>
> Ao fazer *subscrição* de uma *classe*, o método de classe especial *__class_getitem__()* pode ser chamado em vez de `__getitem__()`. Veja *__class_getitem__ versus __getitem__* para mais detalhes.

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

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

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

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

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

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

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

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

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*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Esses métodos são chamados para implementar as operações aritméticas binárias (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) com operandos refletidos (trocados). Essas funções são chamadas apenas se o operando esquerdo não suportar a operação correspondente[3] e os operandos forem de tipos

[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.

diferentes.[4] Por exemplo, para avaliar a expressão `x - y`, onde *y* é uma instância de uma classe que tem um método `__rsub__()`, `type(y).__rsub__(y, x)` é chamado se `type(x).__sub__(x, y)` retorna `NotImplemented`.

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, ou se esse método retorna `NotImplemented`, a atribuição aumentada volta aos métodos normais. Por exemplo, se *x* é uma instância de uma classe com um método `__iadd__()`, `x += y` equivale a `x = x.__iadd__(y)`. Se `__iadd__()` não existe, ou se `x.__iadd__(y)` retorna `NotImplemented`, `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 (veja 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.

[4] Para operandos do mesmo tipo, presume-se que se o método não refletido – como `__add__()` – falhar, a operação geral não será suportada, razão pela qual o método refletido não é chamado.

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`).

A função embutida `int()` retorna para *__trunc__()* se nem *__int__()* nem *__index__()* estiverem definidos.

Alterado na versão 3.11: A delegação de `int()` para *__trunc__()* foi descontinuada.

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 *A instrução with*), 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, travar e destravar recursos, fechar arquivos abertos, etc.

Para mais informações sobre gerenciadores de contexto, veja typecontextmanager. A classe `object` em si não fornece os métodos do gerenciador de contexto.

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.

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

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

Ao usar um nome de classe em um padrão, argumentos posicionais não são permitidos por padrão, ou seja, `case MyClass(x, y)` é tipicamente inválida sem suporte especial em `MyClass`. Para permitir a utilização desse tipo de padrão, a classe precisa definir um atributo *__match_args__*

`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`.

Adicionado na versão 3.10.

Ver também

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

3.3.11 Emulando tipos buffer

O protocolo buffer fornece uma maneira para objetos Python exporem acesso eficiente a um vetor de memória de baixo nível. Este protocolo é implementado por tipos embutido como `bytes` e `memoryview`, e bibliotecas de terceiros podem definir tipos de buffer adicionais.

Embora os tipos buffer sejam geralmente implementados em C, também é possível implementar o protocolo em Python.

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

Chamado quando um buffer é solicitado de *self* (por exemplo, pelo construtor de `memoryview`). O argumento *flags* é um inteiro que representa o tipo de buffer solicitado, afetando, por exemplo, se o buffer retornado é somente leitura ou gravável. `inspect.BufferFlags` fornece uma maneira conveniente de interpretar os sinalizadores. O método deve retornar um objeto `memoryview`.

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

Chamado quando um buffer não é mais necessário. O argumento *buffer* é um objeto `memoryview` que foi retornado anteriormente por *__buffer__()*. O método deve liberar quaisquer recursos associados ao buffer. Este método deve retornar `None`. Objetos buffer que não precisam executar nenhuma limpeza não são necessários para implementar este método.

Adicionado na versão 3.12.

Ver também

PEP 688 - Tornando o protocolo de buffer acessível no Python
Introduz os métodos `__buffer__` e `__release_buffer__` no Python

`collections.abc.Buffer`
ABC para tipos buffer.

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()
```

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

```
>>> 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
```

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

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

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

3.4 Corrotinas

3.4.1 Objetos aguardáveis

Um objeto *aguardável* geralmente implementa um método `__await__()`. Os *objetos corrotina* retornados das funções `async def` são aguardáveis.

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

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

Deve retornar um *iterador*. Deve ser usado para implementar objetos *aguardáveis*. Por exemplo, `asyncio.Future` implementa este método para ser compatível com a expressão `await`. A classe `object` em si não é aguardável e não fornece este método.

> **Nota**
>
> A linguagem não impõe nenhuma restrição ao tipo ou valor dos objetos produzidos pelo iterador retornado por `__await__`, pois isso é específico para a implementação da estrutura de execução assíncrona (por exemplo, `asyncio`) que gerenciará o objeto *awaitable*.

Adicionado na versão 3.5.

> **Ver também**
>
> **PEP 492** para informações adicionais sobre objetos aguardáveis.

3.4.2 Objetos corrotina

Objetos corrotina são objetos *aguardáveis*. A execução de uma corrotina pode ser controlada chamando `__await__()` e iterando sobre o resultado. Quando a corrotina termina a execução e retorna, o iterador levanta `StopIteration`, e o atributo `value` da exceção contém o valor de retorno. Se a corrotina levantar uma exceção, ela será propagada pelo iterador. As corrotinas não devem levantar exceções `StopIteration` diretamente não tratadas.

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*`)`

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

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

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

Levanta a exceção especificada na corrotina. Este método delega ao método `throw()` do iterador que causou

a suspensão da corrotina, se ela tiver tal método. Caso contrário, a exceção é levantada no ponto de suspensão. O resultado (valor de retorno, `StopIteration` ou outra exceção) é o mesmo de iterar sobre o valor de retorno *`__await__()`*, descrito acima. Se a exceção não for capturada na corrotina, ela se propagará de volta para o chamador.

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`*.

A classe `object` em si não fornece estes métodos.

`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):
        ...

    def __aiter__(self):
        return self

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

Adicionado na versão 3.5.

Alterado na versão 3.7: Antes do Python 3.7, *`__aiter__()`* poderia retornar um *aguardável* que resolveria para um *iterador assíncrono*.

A partir do Python 3.7, *`__aiter__()`* deve retornar um objeto iterador assíncrono. Retornar qualquer outra coisa resultará em um erro `TypeError`.

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`*.

A classe `object` em si não fornece estes métodos.

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

Semanticamente semelhante a *__enter__()*, a única diferença é que ele deve retornar um *aguardável*.

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

Semanticamente semelhante a *__exit__()*, a única diferença é que ele deve retornar um *aguardável*.

Um exemplo de uma classe gerenciadora de contexto assíncrona:

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entrando no contexto')

    async def __aexit__(self, exc_type, exc, tb):
        await log('saindo do contexto')
```

Adicionado 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 o FAQ sobre 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 embutido, 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 embutidos será pesquisado em seguida. Se os nomes também não forem encontrados no espaço de nomes embutido, novas variáveis são criadas no espaço de nomes global. 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, restrições e valores padrão para parâmetros de tipo (*avaliados preguiçosamente*).
- O valor dos apelidos de tipo (*avaliado preguiçosamente*).

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 *nome qualificado* dos objetos definidos dentro do escopo. Em vez disso, o `__qualname__` de tais objetos é como se o objeto fosse definido no escopo delimitador.

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

Alterado na versão 3.13: Os escopos de anotação também são usados para padrões de parâmetros de tipo, conforme introduzido pela **PEP 696**.

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, restrições e valores padrão de variáveis de tipo criadas através da *sintaxe do parâmetro 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 preguiçosa 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 preguiçosamente são pesquisados como se fossem usados no escopo imediatamente envolvente.

Adicionado na versão 3.12.

4.2.5 Builtins e execução restrita

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

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

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*.

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 invocar o mecanismo de importação, mas não é a única maneira. Funções como `importlib.import_module()` e a função 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**).

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 função embutida `__import__()` para chamar o mecanismo de importação. Consulte a documentação da biblioteca `importlib` para obter detalhes adicionais.

[1] Veja `types.ModuleType`.

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, o 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 a instrução *`import`*, ou dos parâmetros das funções `importlib.import_module()` ou `__import__()`.

Esse nome será usado em várias fases da busca da importação, e pode ser um nome com pontos para um submódulo como, por exemplo, `foo.bar.baz`. Nesse caso, Python primeiro tenta importar `foo`, depois `foo.bar` e, finalmente, `foo.bar.baz`. Se alguma das importações intermediárias falharem, uma exceção `ModuleNotFoundError` é levantada.

5.3.1 O cache de módulos

A primeira verificação durante a busca da importação é feita no `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` e `foo.bar.baz`. Cada chave terá como valor um objeto módulo correspondente.

Durante a importação, o nome do módulo é procurado em `sys.modules` e, se estiver presente, o valor associado é o módulo que satisfaz a importação, 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 pelo módulo.

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 na próxima importação. Pode ser atribuído `None` para a chave, forçando que a próxima importação 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 Localizadores e carregadores

Se o módulo nomeado não for encontrado em `sys.modules`, então o protocolo de importação do Python é invocado para localizar e carregar o módulo. Este protocolo consiste em dois objetos conceituais, *localizadores* e *carregadores*. O trabalho de um localizador é determinar se ele pode localizar o módulo nomeado usando qualquer estratégia que ele conheça. Objetos que implementam ambas essas interfaces são referenciadas como *importadores* – eles retornam a si mesmos, quando eles descobrem que eles podem carregar o módulo requisitado.

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

O mecanismo de importação é extensível, então novos localizadores podem ser adicionados para estender o alcance e o escopo de buscar módulos.

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

As seguintes seções descrevem o protocolo para localizadores e carregadores em mais detalhes, incluindo como você pode criar e registrar novos para estender o mecanismo de importação.

Alterado na versão 3.4: Em versões anteriores do Python, localizadores retornavam *carregadores* diretamente, enquanto agora eles retornam especificações de módulo, as qual *contêm* carregadores. Carregadores ainda são usados durante a importação, mas possuem menos responsabilidades.

5.3.3 Ganchos de importação

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

Metaganchos 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 aos metaganchos substituir processamento de `sys.path`, módulos congelados ou mesmo módulos embutidos. Metaganchos são registrados adicionando novos objetos localizadores a `sys.meta_path`, conforme descrito abaixo.

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

5.3.4 O metacaminho

Quando o módulo nomeado não é encontrado em `sys.modules`, o Python em seguida o busca em `sys.meta_path`, o qual contém uma lista de objetos localizadores de metacaminho. Esses localizadores 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 três 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 localizador de metacaminho souber como tratar o módulo nomeado, ele retorna um objeto spec. 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 um spec, então `ModuleNotFoundError` é levantada. Quaisquer outras exceções levantadas são simplesmente propagadas para cima, abortando o processo de importação.

O método `find_spec()` dos localizadores de metacaminhos é chamado com dois ou três argumentos. O primeiro é o nome totalmente qualificado do módulo sendo importado, por exemplo `foo.bar.baz`. O segundo argumento são as entradas do caminho a ser usado na busca do módulo. Para módulos de nível superior, o segundo argumento é `None`, mas para submódulos ou subpacotes, o segundo argumento é o valor do atributo `__path__` do pacote pai. Se o atributo `__path__` apropriado não puder ser acessado, uma exceção `ModuleNotFoundError` é levantada. 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 metacaminho pode ser percorrido múltiplas vezes para uma requisição de importação individual. Por exemplo, presumindo 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 localizador de metacaminho (`mpf`). Depois que `foo` foi importado, `foo.bar` será importado percorrendo o metacaminho 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 localizadores de metacaminho apenas dão suporte a importações de alto nível. Estes importadores vão sempre retornar `None` quando qualquer coisa diferente de `None` for passada como o segundo argumento.

O `sys.meta_path` padrão do Python possui três localizador de metacaminho, um que sabe como importar módulos embutidos, um que sabe como importar módulos congelados, e outro que sabe como importar módulos de um *caminho de importação* (isto é, o *localizador baseado no caminho*).

Alterado na versão 3.4: O método `find_spec()` dos localizador de metacaminho substituiu `find_module()`, o qual agora foi descontinuado. Embora continue a funcionar sem alterações, a mecanismo de importação só tentará fazê-lo se o localizador não implementar `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()` foi removido. Use `find_spec()`.

5.4 Carregando

Se e quando uma spec de módulo é encontrada, o mecanismo de importação vai usá-la (e o carregador que ela contém) durante o carregamento do módulo. Esta é 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'):
    # É assumido que o carregador também define 'exec_module'.
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)
# Define os atributos do módulo relacionados à importação:
_init_module_attrs(spec, module)

if spec.loader is None:
    # não suportado
    raise ImportError
if spec.origin is None and spec.submodule_search_locations is not None:
    # pacote espaço de nomes
    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 mecanismo de importação define os atributos de módulo relacionados a importação ("_init_module_attrs" no exemplo de pseudocódigo 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 inerentes dos carregadores. Essas responsabilidades eram anteriormente executadas pelo método `importlib.abc.Loader.load_module()`.

[2] A implementação de importlib evita usar o valor de retorno diretamente. Em vez disso, ela obtém o objeto do módulo procurando o nome do módulo em `sys.modules`. O efeito indireto disso é que um módulo importado pode substituir a si mesmo em `sys.modules`. Esse é um comportamento específico da implementação que não tem garantia de funcionar em outras implementações do Python.

5.4.1 Carregadores

Os carregadores de módulo fornecem a função crítica de carregamento: execução do módulo. O mecanismo de importação chama o método `importlib.abc.Loader.exec_module()` com um único argumento, o objeto do módulo a ser executado. Qualquer valor retornado de `exec_module()` é ignorado.

Os carregadores devem atender aos seguintes requisitos:

- Se o módulo for um módulo Python (em oposição a um módulo embutido ou uma extensão carregada dinamicamente), o carregador deve executar o código do módulo no espaço de nomes global do módulo (`module.__dict__`).
- Se o carregador não puder executar o módulo, ele deve levantar uma execção `ImportError`, embora qualquer outra exceção levantada durante `exec_module()` será propagada.

Em muitos casos, o localizador e o carregador podem ser o mesmo objeto; nesses casos o método `find_spec()` apenas retornaria um spec com o carregador definido como `self`.

Os carregadores de módulo podem optar por criar o objeto do módulo durante o carregamento, implementando um método `create_module()`. Leva um argumento, o spec de módulo e retorna o novo objeto do módulo para usar durante o carregamento. `create_module()` não precisa definir nenhum atributo no objeto do módulo. Se o método retornar `None`, o mecanismo de importação criará ele mesmo o novo módulo.

Adicionado na versão 3.4: O método `create_module()` de carregadores.

Alterado na versão 3.4: O método `load_module()` foi substituído por `exec_module()` e o mecanismo de importação assumiu todas as responsabilidades inerentes de carregamento.

Para compatibilidade com carregadores existentes, o mecanismo de importação usará o método `load_module()` de carregadores se ele existir e o carregador também não implementar `exec_module()`. No entanto, `load_module()` foi descontinuado e os carregadores devem implementar `exec_module()` em seu lugar.

O método `load_module()` deve implementar toda a funcionalidade inerente de carregamento descrita acima, além de executar o módulo. Todas as mesmas restrições se aplicam, com alguns esclarecimentos adicionais:

- Se houver um objeto de módulo existente com o nome fornecido em `sys.modules`, o carregador deverá usar esse módulo existente. (Caso contrário, `importlib.reload()` não funcionará corretamente.) Se o módulo nomeado não existir em `sys.modules`, o carregador deverá criar um novo objeto de módulo e adicioná-lo a `sys.modules`.
- O módulo *deve* existir em `sys.modules` antes que o carregador execute o código do módulo, para evitar recursão ilimitada ou carregamento múltiplo.
- Se o carregamento falhar, o carregador deverá remover quaisquer módulos inseridos em `sys.modules`, mas deverá remover **apenas** o(s) módulo(s) com falha, e somente se o próprio carregador tiver carregado o(s) módulo(s) explicitamente.

Alterado na versão 3.5: Uma exceção `DeprecationWarning` é levantada quando `exec_module()` está definido, mas `create_module()` não.

Alterado na versão 3.6: Uma exceção `ImportError` é levantada quando `exec_module()` está definido, mas `create_module()` não.

Alterado na versão 3.10: O uso de `load_module()` vai levantar `ImportWarning`.

5.4.2 Submódulos

Quando um submódulo é carregado usando qualquer mecanismo (por exemplo, APIs `importlib`, as instruções `import` ou `import-from`, ou a função `__import__()` embutida) uma ligação é colocada no espaço de nomes do módulo pai para o objeto submódulo. Por exemplo, se o pacote `spam` tiver um submódulo `foo`, após importar `spam.foo`, `spam` terá um atributo `foo` que está vinculado ao submódulo. Digamos que você tenha a seguinte estrutura de diretórios:

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

e `spam/__init__.py` tem a seguinte linha:

```
from .foo import Foo
```

então executar o seguinte coloca ligações de nome para `foo` e `Foo` no módulo `spam`:

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

Dadas as conhecidas regras de ligação de nomes do Python, isso pode parecer surpreendente, mas na verdade é um recurso fundamental do sistema de importação. A propriedade invariante é que se você tiver `sys.modules['spam']` e `sys.modules['spam.foo']` (como faria após a importação acima), o último deve aparecer como o atributo `foo` do primeiro.

5.4.3 Especificações de módulo

O mecanismo de importação utiliza diversas informações sobre cada módulo durante a importação, principalmente antes do carregamento. A maior parte das informações é comum a todos os módulos. O propósito da spec do módulo é encapsular essas informações relacionadas à importação por módulo.

Usar um spec durante a importação permite que o estado seja transferido entre componentes do sistema de importação, por exemplo entre o localizador que cria o spec de módulo e o carregador que o executa. Mais importante ainda, permite que o mecanismo de importação execute as operações inerentes de carregamento, enquanto que sem um spec de módulo o carregador tinha essa responsabilidade.

O spec do módulo é exposto no atributo *`module.__spec__`*. A configuração oportuna do `__spec__` se aplica igualmente aos *módulos inicializados durante a inicialização do interpretador*. A única exceção é o `__main__`, cujo `__spec__` é *definido como None em alguns casos*.

Consulte `ModuleSpec` para detalhes sobre o conteúdo do spec de módulo.

Adicionado na versão 3.4.

5.4.4 Atributo __path__ dos módulos

O atributo *`__path__`* deve ser uma *sequência* (possivelmente vazia) de strings enumerando os locais onde os submódulos do pacote serão encontrados. Por definição, um módulo que tenha um atributo `__path__` é um *pacote*.

O atributo *`__path__`* de um pacote é usado durante as importações de seus subpacotes. Dentro do mecanismo de importação, funciona da mesma forma que `sys.path`, ou seja, fornecendo uma lista de locais para procurar módulos durante a importação. Entretanto, o `__path__` normalmente é muito mais restrito que `sys.path`.

As mesmas regras usadas para `sys.path` também se aplicam ao `__path__` de um pacote. Os `sys.path_hooks` (descritos abaixo) são consultados ao percorrer o `__path__` de um pacote.

O arquivo `__init__.py` de um pacote pode definir ou alterar o atributo *`__path__`* do pacote, e esta era tipicamente a forma como os pacotes de espaço de nomes eram implementados antes da **PEP 420**. Com a adoção da **PEP 420**, os pacotes de espaço de nomes não precisam mais fornecer arquivos `__init__.py` contendo apenas código de manipulação do `__path__`; o mecanismo de importação define automaticamente o `__path__` corretamente para o pacote de espaço de nomes.

5.4.5 Representações do módulo

Por padrão, todos os módulos têm uma representação (repr) utilizável, no entanto, dependendo dos atributos definidos acima e do spec do módulo, você pode controlar mais explicitamente a representação dos objetos módulo.

Se o módulo tiver um spec (`__spec__`), o mecanismo de importação tentará gerar uma representação a partir dele. Se isso falhar ou não houver nenhuma especificação, o sistema de importação criará uma representação padrão usando qualquer informação disponível no módulo. Ele tentará usar `module.__name__`, `module.__file__` e `module.__loader__` como entrada para a representação, com padrões para qualquer informação que esteja faltando.

Arquivo estão as exatas regras usadas:

- Se o módulo tiver um atributo `__spec__`, a informação no spec é usada para gerar a representação. Os atributos "name", "loader", "origin" e "has_location" são consultados.
- Se o módulo tiver um atributo `__file__`, ele será usado como parte da representação do módulo.
- Se o módulo não tem `__file__` mas tem um `__loader__` que não seja `None`, então a representação do carregador é usado como parte da representação do módulo.
- Caso contrário, basta usar o `__name__` do módulo na representação.

Alterado na versão 3.12: O uso de `module_repr()`, descontinuado desde o Python 3.4, foi removido no Python 3.12 e não é mais chamado durante a resolução da representação de um módulo.

5.4.6 Invalidação de bytecode em cache

Antes do Python carregar o bytecode armazenado em cache de um arquivo `.pyc`, ele verifica se o cache está atualizado com o arquivo fonte `.py`. Por padrão, o Python faz isso armazenando o registro de data e hora da última modificação da fonte e o tamanho no arquivo de cache ao escrevê-lo. No tempo de execução, o sistema de importação valida o arquivo de cache verificando os metadados armazenados no arquivo de cache em relação aos metadados do código-fonte.

Python também oferece suporte a arquivos de cache "baseados em hash", que armazenam um hash do conteúdo do arquivo fonte em vez de seus metadados. Existem duas variantes de arquivos `.pyc` baseados em hash: verificados e não verificados. Para arquivos `.pyc` baseados em hash verificados, o Python valida o arquivo de cache fazendo hash do arquivo fonte e comparando o hash resultante com o hash no arquivo de cache. Se um arquivo de cache baseado em hash verificado for inválido, o Python o regenerará e gravará um novo arquivo de cache baseado em hash verificado. Para arquivos `.pyc` baseados em hash não verificados, o Python simplesmente presume que o arquivo de cache é válido, se existir. O comportamento de validação de arquivos `.pyc` baseados em hash pode ser substituído pelo sinalizador `--check-hash-based-pycs`.

Alterado na versão 3.7: Adicionados arquivos `.pyc` baseados em hash. Anteriormente, o Python oferecia suporte apenas à invalidação de caches de bytecode baseada em registro de data e hora.

5.5 O localizador baseado no caminho

Conforme mencionado anteriormente, Python vem com vários localizadores de metacaminho padrão. Um deles, chamado *localizador baseado no caminho* (`PathFinder`), pesquisa um *caminho de importação*, que contém uma lista de *entradas de caminho*. Cada entrada de caminho nomeia um local para procurar módulos.

O próprio localizador baseado no caminho não sabe como importar nada. Em vez disso, ele percorre as entradas de caminho individuais, associando cada uma delas a um localizador de entrada de caminho que sabe como lidar com esse tipo específico de caminho.

O conjunto padrão de localizadores de entrada de caminho implementa toda a semântica para localizar módulos no sistema de arquivos, manipulando tipos de arquivos especiais, como código-fonte Python (arquivos `.py`), código de bytes Python (arquivos `.pyc`) e bibliotecas compartilhadas (por exemplo, arquivos `.so`). Quando suportado pelo módulo `zipimport` na biblioteca padrão, os localizadores de entrada de caminho padrão também lidam com o carregamento de todos esses tipos de arquivos (exceto bibliotecas compartilhadas) de arquivos zip.

As entradas de caminho não precisam ser limitadas aos locais do sistema de arquivos. Eles podem referir-se a URLs, consultas de banco de dados ou qualquer outro local que possa ser especificado como uma string.

O localizador baseado no caminho fornece ganchos e protocolos adicionais para que você possa estender e personalizar os tipos de entradas de caminho pesquisáveis. Por exemplo, se você quiser oferecer suporte a entradas de caminho como URLs de rede, poderá escrever um gancho que implemente a semântica HTTP para localizar módulos na web. Este gancho (um chamável) retornaria um *localizador de entrada de caminho* suportando o protocolo descrito abaixo, que foi então usado para obter um carregador para o módulo da web.

Uma palavra de advertência: esta seção e a anterior usam o termo *localizador*, distinguindo-os usando os termos *localizador de metacaminho* e *localizador de entrada de caminho*. Esses dois tipos de localizadores são muito semelhantes, oferecem suporte a protocolos semelhantes e funcionam de maneira semelhante durante o processo de

importação, mas é importante ter em mente que eles são sutilmente diferentes. Em particular, os localizadores de metacaminho operam no início do processo de importação, conforme a travessia de `sys.meta_path`.

Por outro lado, os localizadores de entrada de caminho são, em certo sentido, um detalhe de implementação do localizador baseado no caminho e, de fato, se o localizador baseado no caminho fosse removido de `sys.meta_path`, nenhuma semântica do localizador de entrada de caminho seria ser invocado.

5.5.1 Localizadores de entrada de caminho

O *localizador baseado no caminho* é responsável por encontrar e carregar módulos e pacotes Python cuja localização é especificada com uma string *entrada de caminho*. A maioria das entradas de caminho nomeiam locais no sistema de arquivos, mas não precisam ser limitadas a isso.

Como um localizador de metacaminho, o *localizador baseado no caminho* implementa o protocolo `find_spec()` descrito anteriormente, no entanto, ele expõe ganchos adicionais que podem ser usados para personalizar como os módulos são encontrados e carregado do *caminho de importação*.

Três variáveis são usadas pelo *localizador baseado no caminho*, `sys.path`, `sys.path_hooks` e `sys.path_importer_cache`. Os atributos `__path__` em objetos de pacote também são usados. Eles fornecem maneiras adicionais de personalizar o mecanismo de importação.

`sys.path` contém uma lista de strings fornecendo locais de pesquisa para módulos e pacotes. Ele é inicializado a partir da variável de ambiente `PYTHONPATH` e vários outros padrões específicos de instalação e implementação. Entradas em `sys.path` podem nomear diretórios no sistema de arquivos, arquivos zip e potencialmente outros "locais" (veja o módulo `site`) que devem ser pesquisados por módulos, como URLs, ou consultas ao banco de dados. Apenas strings devem estar presentes em `sys.path`; todos os outros tipos de dados são ignorados.

O *localizador baseado no caminho* é um *localizador de metacaminho*, então o mecanismo de importação inicia a pesquisa no *caminho de importação* chamando o método `find_spec()` do localizador baseado no caminho conforme descrito anteriormente. Quando o argumento `path` para `find_spec()` for fornecido, será uma lista de caminhos de string a serem percorridos – normalmente o atributo `__path__` de um pacote para uma importação dentro desse pacote. Se o argumento `path` for `None`, isso indica uma importação de nível superior e `sys.path` é usado.

O localizador baseado no caminho itera sobre cada entrada no caminho de pesquisa e, para cada uma delas, procura um *localizador de entrada de caminho* (`PathEntryFinder`) apropriado para a entrada do caminho. Como esta pode ser uma operação custosa (por exemplo, pode haver sobrecargas de chamada `stat()` para esta pesquisa), o localizador baseado no caminho mantém um cache mapeando entradas de caminho para localizadores de entrada de caminho. Este cache é mantido em `sys.path_importer_cache` (apesar do nome, este cache na verdade armazena objetos localizadores em vez de ser limitado a objetos *importador*). Desta forma, a dispendiosa busca pelo *localizador de entrada de caminho* de local específico de uma *entrada de caminho* só precisa ser feita uma vez. O código do usuário é livre para remover entradas de cache de `sys.path_importer_cache`, forçando o localizador baseado no caminho a realizar a pesquisa de entrada de caminho novamente.

Se a entrada de caminho não estiver presente no cache, o localizador baseado no caminho itera sobre cada chamável em `sys.path_hooks`. Cada um dos *ganchos de entrada de caminho* nesta lista é chamado com um único argumento, a entrada de caminho a ser pesquisada. Este chamável pode retornar um *localizador de entrada de caminho* que pode manipular a entrada de caminho ou pode levantar `ImportError`. Um `ImportError` é usado pelo localizador baseado no caminho para sinalizar que o gancho não consegue encontrar um *localizador de entrada de caminho* para aquela *entrada de caminho*. A exceção é ignorada e a iteração com o *caminho de importação* continua. O gancho deve esperar um objeto string ou bytes; a codificação de objetos bytes depende do gancho (por exemplo, pode ser uma codificação de sistema de arquivos, UTF-8 ou outra coisa) e, se o gancho não puder decodificar o argumento, ele deve levantar `ImportError`.

Se a iteração `sys.path_hooks` terminar sem que nenhum *localizador de entrada de caminho* seja retornado, o método `find_spec()` do localizador baseado no caminho armazenará `None` em `sys.path_importer_cache` (para indicar que não há um localizador para esta entrada de caminho) e retornará `None`, indicando que este *localizador de metacaminho* não conseguiu encontrar o módulo.

Se um *localizador de entrada de caminho for* retornado por um dos chamáveis de *gancho de entrada de caminho* em `sys.path_hooks`, então o seguinte protocolo é usado para solicitar ao localizador um spec de módulo, que é então usada ao carregar o módulo.

O diretório de trabalho atual – denotado por uma string vazia – é tratado de forma ligeiramente diferente de outras entradas em `sys.path`. Primeiro, se o diretório de trabalho atual for considerado inexistente, nenhum valor será armazenado em `sys.path_importer_cache`. Segundo, o valor para o diretório de trabalho atual é pesquisado novamente para cada pesquisa de módulo. Terceiro, o caminho usado para `sys.path_importer_cache` e retornado por `importlib.machinery.PathFinder.find_spec()` será o diretório de trabalho atual real e não a string vazia.

5.5.2 Protocolo do localizador de entrada de caminho

Para dar suporte a importações de módulos e pacotes inicializados e também contribuir com partes para pacotes de espaço de nomes, os localizadores de entrada de caminho devem implementar o método `find_spec()`.

`find_spec()` recebe dois argumentos: o nome totalmente qualificado do módulo que está sendo importado e o módulo de destino (opcional). `find_spec()` retorna um spec totalmente preenchido para o módulo. Este spec sempre terá "loader" definido (com uma exceção).

Para indicar ao maquinário de importação que o spec representa uma *porção* de espaço de nomes, o localizador de entrada de caminho define `submodule_search_locations` como uma lista contendo a porção.

Alterado na versão 3.4: `find_spec()` substituiu `find_loader()` e `find_module()`, ambos descontinuados, mas serão usados se `find_spec()` não estiver definido.

Os localizadores de entrada de caminho mais antigos podem implementar um desses dois métodos descontinuados em vez de `find_spec()`. Os métodos ainda são respeitados para fins de compatibilidade com versões anteriores. No entanto, se `find_spec()` for implementado no localizador de entrada de caminho, os métodos legados serão ignorados.

`find_loader()` recebe um argumento, o nome totalmente qualificado do módulo que está sendo importado. `find_loader()` retorna uma tupla 2 onde o primeiro item é o carregador e o segundo item é uma *porção* de espaço de nomes.

Para compatibilidade com versões anteriores de outras implementações do protocolo de importação, muitos localizadores de entrada de caminho também dão suporte ao mesmo método tradicional `find_module()` que os localizadores de metacaminho. No entanto, os métodos `find_module()` do localizador de entrada de caminho nunca são chamados com um argumento `path` (espera-se que eles registrem as informações de caminho apropriadas da chamada inicial para o gancho de caminho).

O método `find_module()` em localizadores de entrada de caminho foi descontinuado, pois não permite que o localizador de entrada de caminho contribua com porções para pacotes de espaço de nomes. Se `find_loader()` e `find_module()` existirem em um localizador de entrada de caminho, o sistema de importação sempre chamará `find_loader()` em preferência a `find_module()`.

Alterado na versão 3.10: Chamadas para `find_module()` e `find_loader()` pelo sistema de importação vão levantar `ImportWarning`.

Alterado na versão 3.12: `find_module()` e `find_loader()` foram removidos.

5.6 Substituindo o sistema de importação padrão

O mecanismo mais confiável para substituir todo o sistema de importação é excluir o conteúdo padrão de `sys.meta_path`, substituindo-o inteiramente por um gancho de metacaminho personalizado.

Se for aceitável alterar apenas o comportamento de instruções de importação sem afetar outras APIs que acessam o sistema de importação, então substituir a função embutida `__import__()` pode ser suficiente. Essa técnica também pode ser empregada no nível do módulo para alterar apenas o comportamento de instruções de importação dentro desse módulo.

Para impedir seletivamente a importação de alguns módulos de um gancho no início do metacaminho (em vez de desabilitar o sistema de importação padrão completamente), é suficiente levantar `ModuleNotFoundError` diretamente de `find_spec()` em vez de retornar `None`. O último indica que a busca do metacaminho deve continuar, enquanto levantar uma exceção a encerra imediatamente.

5.7 Importações relativas ao pacote

Importações relativas usam caracteres de ponto no início. Um único ponto no início indica uma importação relativa, começando com o pacote atual. Dois ou mais pontos no início indicam uma importação relativa para o(s) pai(s) do pacote atual, um nível por ponto após o primeiro. Por exemplo, dado o seguinte layout de pacote:

```
package/
    __init__.py
    subpackage1/
        __init__.py
        moduleX.py
        moduleY.py
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

Em `subpackage1/moduleX.py` ou `subpackage1/__init__.py`, as seguintes são importações relativas válidas:

```
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
```

Importações absolutas podem usar a sintaxe `import <>` ou `from <> import <>`, mas importações relativas podem usar apenas a segunda forma; o motivo para isso é que:

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

deve expor `XXX.YYY.ZZZ` como uma expressão utilizável, mas .moduleY não é uma expressão válida.

5.8 Considerações especiais para __main__

O módulo `__main__` é um caso especial em relação ao sistema de importação do Python. Conforme observado em *em outro lugar*, o módulo `__main__` é inicializado diretamente na inicialização do interpretador, muito parecido com `sys` e `builtins`. No entanto, diferentemente desses dois, ele não se qualifica estritamente como um módulo integrado. Isso ocorre porque a maneira como `__main__` é inicializado depende dos sinalizadores e outras opções com as quais o interpretador é invocado.

5.8.1 __main__.__spec__

Dependendo de como `__main__` é inicializado, `__main__.__spec__` é definido apropriadamente ou como `None`.

Quando o Python é iniciado com a opção `-m`, `__spec__` é definido como o spec de módulo ou pacote correspondente. `__spec__` também é preenchido quando o módulo `__main__` é carregado como parte da execução de um diretório, arquivo zip ou outra entrada `sys.path`.

Nos demais casos, `__main__.__spec__` é definido como `None`, pois o código usado para preencher o `__main__` não corresponde diretamente a um módulo importável:

- prompt interativo
- opção `-c`
- executar a partir de stdin
- executar diretamente de um arquivo de código-fonte ou bytecode

Note que `__main__.__spec__` é sempre `None` no último caso, *mesmo se* o arquivo pudesse ser importado diretamente como um módulo. Use a opção `-m` se metadados de módulo válidos forem desejados em `__main__`.

Note também que mesmo quando `__main__` corresponde a um módulo importável e `__main__.__spec__` é definido adequadamente, eles ainda são considerados módulos *distintos*. Isso se deve ao fato de que os blocos protegidos por verificações `if __name__ == "__main__":` são executados somente quando o módulo é usado para preencher o espaço de nomes `__main__`, e não durante a importação normal.

5.9 Referências

O maquinário de importação evoluiu consideravelmente desde os primeiros dias do Python. A especificação original para pacotes ainda está disponível para leitura, embora alguns detalhes tenham mudado desde a escrita desse documento.

A especificação original para `sys.meta_path` era **PEP 302**, com extensão subsequente em **PEP 420**.

PEP 420 introduziu *pacotes de espaço de nomes* para Python 3.3. **PEP 420** também introduziu o protocolo `find_loader()` como uma alternativa ao `find_module()`.

PEP 366 descreve a adição do atributo `__package__` para importações relativas explícitas em módulos principais.

PEP 328 introduziu importações relativas absolutas e explícitas e inicialmente propôs `__name__` para semântica. **PEP 366** eventualmente especificaria `__package__`.

PEP 338 define módulos de execução como scripts.

PEP 451 adiciona o encapsulamento do estado de importação por módulo em objetos spec. Ele também descarrega a maioria das responsabilidades inerentes dos carregadores de volta para o maquinário de importação. Essas mudanças permitem a descontinuação de várias APIs no sistema de importação e também a adição de novos métodos para localizadores e carregadores.

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`.

Desfiguração 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 com dois ou mais sublinhados, ele é considerado um *nome privado* daquela classe.

Ver também

As *especificações de classe*.

Mais precisamente, os nomes privados são transformados em um formato mais longo antes que o código seja gerado para eles. Se o nome transformado tiver mais de 255 caracteres, poderá ocorrer truncamento definido pela implementação.

A transformação é independente do contexto sintático no qual o identificador é usado, mas apenas os seguintes identificadores privados são desfigurados:

- Qualquer nome usado como nome de uma variável que é atribuída ou lida ou qualquer nome de um atributo que está sendo acessado.

  O atributo `__name__` de funções aninhadas, classes e apelidos de tipo, entretanto, não é desfigurado.

- O nome dos módulos importados, por exemplo, `__spam` em `import __spam`. Se o módulo faz parte de um pacote (ou seja, seu nome contém um ponto), o nome *não* é desfigurado, por exemplo, o `__foo` em `import __foo.bar` não é desfigurado.

- O nome de um membro importado, por exemplo, `__f` em `from spam import __f`.

A regra de transformação está definida da seguinte forma:

- O nome da classe, com os sublinhados iniciais removidos e um único sublinhado inicial inserido, é inserido na frente do identificador, por exemplo, o identificador `__spam` ocorrendo em uma classe chamada `Foo`, `_Foo` ou `__Foo` é transformado em `_Foo__spam`.

- Se o nome da classe consiste apenas em sublinhados, a transformação é a identidade, por exemplo, o identificador `__spam` que ocorre em uma classe chamada `_` ou `__` é deixado como está.

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 forma 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 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 inicial, pode conter `for` adicional ou `async for` e também pode usar expressões *`await`*.

Se uma compreensão contém cláusulas `async for`, ou se contém expressões `await` ou outras compreensões assíncronas em qualquer lugar, exceto a expressão iterável na cláusula `for` mais à esquerda, ela é chamada uma *compreensão assíncrona*. Uma compreensão assíncrona pode suspender a execução da função de corrotina em que aparece. Veja também a **PEP 530**.

Adicionado 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 ::= "[" [flexible_expression_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 ::= "{" (flexible_expression_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.

Adicionado na versão 3.5: Desempacotando 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*).

Adicionado 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" yield_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():  # define uma função geradora
    yield 123

async def agen(): # define uma função geradora assíncrona
    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 *yield_list* ao chamador do gerador, ou `None` se *yield_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 *yield_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 captura a exceção, e retorna um valor, este valor é retornado de *close()*. Se a função geradora já estiver fechada ou levantar `GeneratorExit` (por não capturar a exceção), *close()* retornará `None`. Se o gerador produzir um valor, uma exceção `RuntimeError` é levantada. Se o gerador levantar qualquer outra exceção, ela será propagada para o chamador. Se o gerador já saiu devido a uma exceção ou saída normal, *close()* retorna `None` e tem nenhum outro efeito.

Alterado na versão 3.13: Se um gerador retornar um valor ao ser fechado, o valor será retornado por *close()*.

Exemplos

Aqui está um exemplo simples que demonstra o comportamento de geradores e funções geradoras:

```
>>> def echo(value=None):
...     print("A execução inicia quando 'next()' é chamada pela primeira vez.")
...     try:
...         while True:
...             try:
...                 value = (yield value)
...             except Exception as e:
...                 value = e
...     finally:
...         print("Não se esqueça de fazer uma limpeza quando 'close()' for␣
↪chamada.")
...
>>> generator = echo(1)
>>> print(next(generator))
A execução inicia quando 'next()' é chamada pela primeira vez.
1
>>> print(next(generator))
None
>>> print(generator.send(2))
2
```

(continua na próxima página)

(continuação da página anterior)

```
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Não se esqueça de fazer uma limpeza quando 'close()' for chamada.
```

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 *yield_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.

async 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 *yield_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*.

async 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.

async agen.**athrow**(*value*)

async 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.

async 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 "[" flexible_expression_list "]"
```

Quando um objeto é subscrito, o interpretador avaliará o primário e a lista de expressões.

O primário deve ser avaliado como um objeto que dê suporte à subscrição. Um objeto pode prover suporte a subscrição através da definição de um ou ambos *__getitem__()* e *__class_getitem__()*. Quando o primário é subscrito, o resultado avaliado da lista de expressões será passado para um desses métodos. Para mais detalhes sobre quando `__class_getitem__` é chamado em vez de `__getitem__`, veja *__class_getitem__ versus __getitem__*.

Se a lista de expressões contiver pelo menos uma vírgula ou se alguma das expressões for estrelada, ela será avaliada como uma `tuple` contendo os itens da lista de expressões. Caso contrário, a lista de expressões será avaliada como o valor do único membro da lista.

Alterado na versão 3.11: Expressões em uma lista de expressões podem ser estreladas. Veja a **PEP 646**.

Para objetos embutido, existem dois tipos de objetos que oferecem suporte a subscrição via *__getitem__()*:

1. Mapeamentos. Se o primário for um *mapeamento*, a lista de expressões deve ser avaliada como um objeto cujo valor é uma das chaves do mapeamento, e a subscrição seleciona o valor no mapeamento que corresponde a essa chave. Um exemplo de classe de mapeamento integrada é a classe `dict`.
2. Sequências. Se o primário for uma *sequência*, a lista de expressões deve ser avaliada como `int` ou `slice` (conforme discutido na seção seguinte). Exemplos de classes de sequência embutidas incluem as classes `str`, `list` e `tuple`.

A sintaxe formal não faz nenhuma provisão especial para índices negativos em *sequências*. No entanto, todas as sequências embutidas fornecem um método *__getitem__()* que interpreta índices negativos adicionando o comprimento da sequência ao índice para que, por exemplo, `x[-1]` selecione o último item de `x`. O valor resultante deve ser um número inteiro não negativo menor que o número de itens na sequência, e a subscrição seleciona o item cujo índice é esse valor (contando a partir de zero). Como o suporte para índices negativos e fatiamento ocorre no método *__getitem__()* do objeto, as subclasses que substituem esse método precisarão adicionar explicitamente esse suporte.

Uma `string` é um tipo especial de sequência cujos itens são *caracteres*. Um caractere não é um tipo de dados separado, mas uma string de exatamente um caractere.

6.3.3 Fatiamentos

Um fatiamento seleciona um intervalo de itens em um objeto sequência (por exemplo, uma string, tupla ou lista). As fatias podem ser usadas como expressões ou como alvos em instruções de atribuição ou *del*. A sintaxe para um fatiamento:

```
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
```

Há ambiguidade na sintaxe formal aqui: qualquer coisa que se pareça com uma lista de expressões também se parece com uma lista de fatias, portanto qualquer subscrição pode ser interpretada como um fatiamento. Em vez de complicar ainda mais a sintaxe, isso é eliminado pela definição de que, neste caso, a interpretação como uma subscrição tem prioridade sobre a interpretação como um fatiamento (este é o caso se a lista de fatias não contiver uma fatia adequada).

A semântica para um fatiamento é a seguinte. O primário é indexado (usando o mesmo método *__getitem__()* da subscrição normal) com uma chave que é construída a partir da lista de fatias, como segue. Se a lista de fatias contiver pelo menos uma vírgula, a chave será uma tupla contendo a conversão dos itens da fatia; caso contrário, a

conversão do item de fatia isolada é a chave. A conversão de um item de fatia que é uma expressão é essa expressão. A conversão de uma fatia adequada é um objeto fatia (veja a seção *A hierarquia de tipos padrão*) cujos `start`, `stop` e `step` atributos são os valores das expressões fornecidas como limite inferior, limite superior e passo, respectivamente, substituindo `None` pelas expressões ausentes.

6.3.4 Chamadas

Uma chamada chama um objeto que é um chamável (por exemplo, uma *função*) com uma série possivelmente vazia de *argumentos*:

```
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
```

Uma vírgula final opcional pode estar presente após os argumentos posicionais e nomeados, mas não afeta a semântica.

O primário deve ser avaliado como um objeto que pode ser chamado (funções definidas pelo usuário, funções embutidas, métodos de objetos embutidos, objetos de classe, métodos de instâncias de classe e todos os objetos que possuem um método *`__call__()`* são chamáveis). Todas as expressões de argumento são avaliadas antes da tentativa de chamada. Consulte a seção *Definições de função* para a sintaxe das listas formais de *parâmetros*.

Se houver argumentos nomeados, eles serão primeiro convertidos em argumentos posicionais, como segue. Primeiro, é criada uma lista de slots não preenchidos para os parâmetros formais. Se houver N argumentos posicionais, eles serão colocados nos primeiros N slots. A seguir, para cada argumento nomeado, o identificador é usado para determinar o slot correspondente (se o identificador for igual ao primeiro nome formal do parâmetro, o primeiro slot será usado e assim por diante). Se o slot já estiver preenchido, uma exceção `TypeError` será levantada. Caso contrário, o argumento é colocado no slot, preenchendo-o (mesmo que a expressão seja `None`, ela preenche o slot). Quando todos os argumentos forem processados, os slots ainda não preenchidos serão preenchidos com o valor padrão correspondente da definição da função. (Os valores padrão são calculados, uma vez, quando a função é definida; assim, um objeto mutável, como uma lista ou dicionário usado como valor padrão, será compartilhado por todas as chamadas que não especificam um valor de argumento para o slot correspondente; isso deve geralmente ser evitado.) Se houver algum slot não preenchido para o qual nenhum valor padrão for especificado, uma exceção `TypeError` será levantada. Caso contrário, a lista de slots preenchidos será usada como lista de argumentos para a chamada.

Uma implementação pode fornecer funções integradas cujos parâmetros posicionais não possuem nomes, mesmo que sejam 'nomeados' para fins de documentação e que, portanto, não possam ser fornecidos por nomes. No CPython, este é o caso de funções implementadas em C que usam `PyArg_ParseTuple()` para analisar seus argumentos.

Se houver mais argumentos posicionais do que slots de parâmetros formais, uma exceção `TypeError` será levantada, a menos que um parâmetro formal usando a sintaxe `*identificador` esteja presente; neste caso, esse parâmetro formal recebe uma tupla contendo os argumentos posicionais em excesso (ou uma tupla vazia se não houver argumentos posicionais em excesso).

Se algum argumento nomeado não corresponder a um nome de parâmetro formal, uma exceção `TypeError` é levantada, a menos que um parâmetro formal usando a sintaxe `**identificador` esteja presente; neste caso, esse parâmetro formal recebe um dicionário contendo os argumentos nomeados em excesso (usando os nomes como chaves e os valores dos argumentos como valores correspondentes), ou um (novo) dicionário vazio se não houver argumentos nomeados em excesso.

Se a sintaxe `*expressão` aparecer na chamada da função, `expressão` deverá ser avaliada como *iterável*. Os elementos desses iteráveis são tratados como se fossem argumentos posicionais adicionais. Para a chamada `f(x1, x2, *y, x3, x4)`, se *y* for avaliado como uma sequência *y1*, …, *yM*, isso é equivalente a uma chamada com M+4 argumentos posicionais *x1*, *x2*, *y1*, …, *yM*, *x3*, *x4*.

Uma consequência disso é que embora a sintaxe `*expressão` possa aparecer *depois* de argumentos nomeados explícitos, ela é processada *antes* dos argumentos nomeados (e de quaisquer argumentos de `**expressão` – veja abaixo). Então:

```
>>> 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
```

É incomum que ambos os argumentos nomeados e a sintaxe `*expressão` sejam usados na mesma chamada, portanto, na prática, essa confusão não surge com frequência.

Se a sintaxe `**expressão` aparecer na chamada de função, `expressão` deve ser avaliada como um *mapeamento*, cujo conteúdo é tratado como argumentos nomeados adicionais. Se um parâmetro que corresponde a uma chave já recebeu um valor (por um argumento nomeado explícito ou de outro desempacotamento), uma exceção `TypeError` é levantada.

Quando `**expressão` é usada, cada chave neste mapeamento deve ser uma string. Cada valor do mapeamento é atribuído ao primeiro parâmetro formal elegível para atribuição de nomeas cujo nome é igual à chave. Uma chave não precisa ser um identificador Python (por exemplo, `"max-temp °F"` é aceitável, embora não corresponda a nenhum parâmetro formal que possa ser declarado). Se não houver correspondência com um parâmetro formal, o par chave-valor é coletado pelo parâmetro `**`, se houver, ou se não houver, uma exceção `TypeError` é levantada.

Parâmetros formais usando a sintaxe `*identificador` ou `**identificador` não podem ser usados como slots de argumentos posicionais ou como nomes de argumentos nomeados.

Alterado na versão 3.5: Chamadas de função aceitam qualquer número de desempacotamentos `*` e `**`, argumentos posicionais podem seguir desempacotamentos iteráveis (`*`) e argumentos nomeados podem seguir desempacotamentos de dicionário (`**`). Originalmente proposto pela **PEP 448**.

Uma chamada sempre retorna algum valor, possivelmente `None`, a menos que levanta uma exceção. A forma como esse valor é calculado depende do tipo do objeto chamável.

Se for…

uma função definida por usuário:
: O bloco de código da função é executado, passando-lhe a lista de argumentos. A primeira coisa que o bloco de código fará é vincular os parâmetros formais aos argumentos; isso é descrito na seção *Definições de função*. Quando o bloco de código executa uma instrução *`return`*, isso especifica o valor de retorno da chamada de função. Se a execução atingir o final do bloco de código sem executar uma instrução *`return`*, o valor de retorno será `None`.

um método embutido ou uma função embutida:
: O resultado fica por conta do interpretador; veja built-in-funcs para descrições de funções embutidas e métodos embutidos.

um objeto classe:
: Uma nova instância dessa classe é retornada.

um método de instância de classe:
: A função correspondente definida pelo usuário é chamada, com uma lista de argumentos que é maior que a lista de argumentos da chamada: a instância se torna o primeiro argumento.

uma instância de classe:
: A classe deve definir um método *`__call__()`*; o efeito é então o mesmo como se esse método fosse chamado.

6.4 Expressão await

Suspende a execução de *corrotina* em um objeto *aguardável*. Só pode ser usado dentro de uma *função de corrotina*.

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

Adicionado 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.

Para operandos int, o resultado tem o mesmo tipo que os operandos, a menos que o segundo argumento seja negativo; nesse caso, todos os argumentos são convertidos em ponto flutuante e um resultado ponto flutuante é entregue. Por exemplo, `10**2` retorna `100`, mas `10**-2` retorna `0.01`.

Elevar `0.0` a uma potência negativa resulta em uma exceção `ZeroDivisionError`. Elevar um número negativo a uma potência fracionária resulta em um número `complex`. (Em versões anteriores, levantava `ValueError`.)

Esta operação pode ser personalizada usando os métodos especial *__pow__()* e *__rpow__()*.

6.6 Operações aritméticas unárias e bit a bit

Todas as operações aritméticas unárias e bit a bit têm a mesma prioridade:

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

O operador unário - (menos) produz a negação de seu argumento numérico; a operação pode ser substituída pelo método especial *__neg__()*.

O operador unário + (mais) produz seu argumento numérico inalterado; a operação pode ser substituída pelo método especial *__pos__()*.

O operador unário ~ (inverter) produz a inversão bit a bit de seu argumento inteiro. A inversão bit a bit de `x` é definida como `-(x+1)`. Aplica-se apenas a números inteiros ou a objetos personalizados que substituem o método especial *__invert__()*.

Em todos os três casos, se o argumento não tiver o tipo adequado, uma exceção `TypeError` é levantada.

6.7 Operações binárias aritméticas

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.

Esta operação pode ser personalizada usando os métodos especial *__mul__()* e *__rmul__()*.

O operador `@` (arroba) deve ser usado para multiplicação de matrizes. Nenhum tipo embutido do Python implementa este operador.

Esta operação pode ser personalizada usando os métodos especial *__matmul__()* e *__rmatmul__()*.

Adicionado 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`.

A operação de divisão pode ser personalizada usando os métodos especiais *__truediv__()* e *__rtruediv__()*. A operação de divisão pelo piso pode ser personalizada usando os métodos especiais *__floordiv__()* e *__rfloordiv__()*.

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.

A operação *módulo* pode ser personalizada usando os métodos especial *__mod__()* e *__rmod__()*.

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.

Esta operação pode ser personalizada usando os métodos especial *__add__()* e *__radd__()*.

O operador `-` (subtração) produz a diferença de seus argumentos. Os argumentos numéricos são primeiro convertidos em um tipo comum.

Esta operação pode ser personalizada usando os métodos especial *__sub__()* e *__rsub__()*.

6.8 Operações de deslocamento

As operações de deslocamento têm menor prioridade que as operações aritméticas:

```
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.

[1] Embora `abs(x%y) < abs(y)` seja verdadeiro matematicamente, para números flutuantes pode não ser verdadeiro numericamente devido ao arredondamento. Por exemplo, e presumindo uma plataforma na qual um float Python seja um número de precisão dupla IEEE 754, para que `-1e-100 % 1e100` tenha o mesmo sinal que `1e100`, o resultado calculado é `-1e-100 + 1e100`, que é numericamente exatamente igual a `1e100`. A função `math.fmod()` retorna um resultado cujo sinal corresponde ao sinal do primeiro argumento e, portanto, retorna `-1e-100` neste caso. Qual abordagem é mais apropriada depende da aplicação.

[2] Se x estiver muito próximo de um múltiplo inteiro exato de y, é possível que `x//y` seja maior que `(x-x%y)//y` devido ao arredondamento. Nesses casos, Python retorna o último resultado, para preservar que `divmod(x,y)[0] * y + x % y` esteja muito próximo de `x`.

A operação de deslocamento à esquerda pode ser personalizada usando os métodos especiais *__lshift__()* e *__rlshift__()*. A operação de deslocamento à direita pode ser personalizada usando os métodos especiais *__rshift__()* e *__rrshift__()*.

Um deslocamento para a direita por *n* bits é definido como divisão pelo piso por `pow(2,n)`. Um deslocamento à esquerda por *n* bits é definido como multiplicação com `pow(2,n)`.

6.9 Operações binárias bit a bit

Cada uma das três operações bit a bit tem um nível de prioridade diferente:

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

O operador `&` produz o E (AND) bit a bit de seus argumentos, que devem ser inteiros ou um deles deve ser um objeto personalizado substituindo os métodos especiais *__and__()* ou *__rand__()*.

O operador `^` produz o XOR bit a bit (OU exclusivo) de seus argumentos, que devem ser inteiros ou um deles deve ser um objeto personalizado sobrescrevendo os métodos especiais *__xor__()* ou *__rxor__()*.

O operador `|` produz o OU (OR) bit a bit de seus argumentos, que devem ser inteiros ou um deles deve ser um objeto personalizado sobrescrevendo os métodos especiais *__or__()* ou *__ror__()*.

6.10 Comparações

Ao contrário de C, todas as operações de comparação em Python têm a mesma prioridade, que é menor do que qualquer operação aritmética, de deslocamento ou bit a bit. Também diferentemente de C, expressões como `a < b < c` têm a interpretação que é convencional em matemática:

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

Comparações produzem valores booleanos: `True` ou `False`. *métodos de comparação rica* personalizados podem retornar valores não booleanos. Neste caso, o Python chamará `bool()` nesse valor em contextos booleanos.

As comparações podem ser encadeadas arbitrariamente, por exemplo, `x < y <= z` é equivalente a `x < y and y <= z`, exceto que `y` é avaliado apenas uma vez (mas em ambos os casos `z` não é avaliado quando `x < y` é considerado falso).

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.

Observe que `a op1 b op2 c` não implica qualquer tipo de comparação entre *a* e *c*, de modo que, por exemplo, `x < y > z` é perfeitamente válido (embora talvez não seja bonito).

6.10.1 Comparações de valor

Os operadores <, >, ==, >=, <= e != comparam os valores de dois objetos. Os objetos não precisam ser do mesmo tipo.

O capítulo *Objetos, valores e tipos* afirma que os objetos possuem um valor (além do tipo e da identidade). O valor de um objeto é uma noção bastante abstrata em Python: por exemplo, não existe um método de acesso canônico para o valor de um objeto. Além disso, não há exigência de que o valor de um objeto seja construído de uma maneira específica, por exemplo. composto por todos os seus atributos de dados. Os operadores de comparação implementam uma noção específica de qual é o valor de um objeto. Pode-se pensar neles como definindo o valor de um objeto indiretamente, por meio de sua implementação de comparação.

Como todos os tipos são subtipos (diretos ou indiretos) de `object`, eles herdam o comportamento de comparação padrão de `object`. Os tipos podem personalizar seu comportamento de comparação implementando *métodos de comparação rica* como `__lt__()`, descrito em *Personalização básica*.

O comportamento padrão para comparação de igualdade (== e !=) é baseado na identidade dos objetos. Consequentemente, a comparação da igualdade de instâncias com a mesma identidade resulta em igualdade, e a comparação da igualdade de instâncias com identidades diferentes resulta em desigualdade. Uma motivação para este comportamento padrão é o desejo de que todos os objetos sejam reflexivos (ou seja, `x is y` implica `x == y`).

Uma comparação de ordem padrão (<, >, <= e >=) não é fornecida; uma tentativa levanta `TypeError`. Uma motivação para este comportamento padrão é a falta de um invariante semelhante ao da igualdade.

O comportamento da comparação de igualdade padrão, de que instâncias com identidades diferentes são sempre desiguais, pode contrastar com o que os tipos precisarão ter uma definição sensata de valor de objeto e igualdade baseada em valor. Esses tipos precisarão personalizar seu comportamento de comparação e, de fato, vários tipos embutidos fizeram isso.

A lista a seguir descreve o comportamento de comparação dos tipos embutidos mais importantes.

- Números de tipos numéricos embutidos (typesnumeric) e dos tipos de biblioteca padrão `fractions.Fraction` e `decimal.Decimal` podem ser comparados dentro e entre seus tipos, com a restrição que os números complexos não oferecem suporte a comparação de ordens. Dentro dos limites dos tipos envolvidos, eles comparam matematicamente (algoritmicamente) corretos sem perda de precisão.

  Os valores não numéricos `float('NaN')` e `decimal.Decimal('NaN')` são especiais. Qualquer comparação ordenada de um número com um valor que não é um número é falsa. Uma implicação contraintuitiva é que os valores que não são numéricos não são iguais a si mesmos. Por exemplo, se `x = float('NaN')`, `3 < x`, `x < 3` e `x == x` são todos falsos, enquanto `x != x` é verdadeiro. Esse comportamento é compatível com IEEE 754.

- `None` e `NotImplemented` são singletons. **PEP 8** aconselha que comparações para singletons devem sempre ser feitas com `is` ou `is not`, nunca com os operadores de igualdade.

- Sequências binárias (instâncias de `bytes` ou `bytearray`) podem ser comparadas dentro e entre seus tipos. Eles comparam lexicograficamente usando os valores numéricos de seus elementos.

- Strings (instâncias de `str`) são comparadas lexicograficamente usando os pontos de código Unicode numéricos (o resultado da função embutida `ord()`) de seus caracteres.[3]

  Strings e sequências binárias não podem ser comparadas diretamente.

- Sequências (instâncias de `tuple`, `list` ou `range`) podem ser comparadas apenas dentro de cada um de seus tipos, com a restrição de que intervalos não oferecem suporte a comparação de ordem. A comparação de igualdade entre esses tipos resulta em desigualdade, e a comparação ordenada entre esses tipos levanta `TypeError`.

  As sequências são comparadas lexicograficamente usando a comparação de elementos correspondentes. Os contêineres embutidos normalmente presumem que objetos idênticos são iguais a si mesmos. Isso permite ignorar testes de igualdade para objetos idênticos para melhorar o desempenho e manter seus invariantes internos.

  A comparação lexicográfica entre coleções embutidas funciona da seguinte forma:

  - Para que duas coleções sejam comparadas iguais, elas devem ser do mesmo tipo, ter o mesmo comprimento e cada par de elementos correspondentes deve ser comparado igual (por exemplo, `[1,2] == (1,2)` é false porque o tipo não é o mesmo).

[3] O padrão Unicode distingue entre *pontos de código* (por exemplo, U+0041) e *caracteres abstratos* (por exemplo, "LATIN CAPITAL LETTER A"). Embora a maioria dos caracteres abstratos em Unicode sejam representados apenas por meio de um ponto de código, há vários caracteres abstratos que também podem ser representados por meio de uma sequência de mais de um ponto de código. Por exemplo, o caractere abstrato "LATIN CAPITAL LETTER C WITH CEDILLA" pode ser representado como um único *caractere pré-composto* na posição de código U+00C7, ou como uma sequência de um *caractere base* na posição de código U+0043 (LATIN CAPITAL LETTER C), seguido por um *caractere de combinação* na posição de código U+0327 (COMBINING CEDILLA).

Os operadores de comparação em strings são comparados no nível dos pontos de código Unicode. Isso pode ser contraintuitivo para os humanos. Por exemplo, `"\u00C7" == "\u0043\u0327"` é `False`, mesmo que ambas as strings representem o mesmo caractere abstrato "LATIN CAPITAL LETTER C WITH CEDILLA".

Para comparar strings no nível de caracteres abstratos (ou seja, de uma forma intuitiva para humanos), use `unicodedata.normalize()`.

  - Coleções que oferecem suporte a comparação de ordem são ordenadas da mesma forma que seus primeiros elementos desiguais (por exemplo, `[1,2,x] <= [1,2,y]` tem o mesmo valor que `x <= y`). Se um elemento correspondente não existir, a coleção mais curta é ordenada primeiro (por exemplo, `[1,2] < [1,2,3]` é verdadeiro).

- Mapeamentos (instâncias de `dict`) comparam iguais se e somente se eles tiverem pares `(chave, valor)` iguais. A comparação de igualdade das chaves e valores reforça a reflexividade.

  Comparações de ordem (`<`, `>`, `<=` e `>=`) levantam `TypeError`.

- Conjuntos (instâncias de `set` ou `frozenset`) podem ser comparados dentro e entre seus tipos.

  Eles definem operadores de comparação de ordem para significar testes de subconjunto e superconjunto. Essas relações não definem ordenações totais (por exemplo, os dois conjuntos `{1,2}` e `{2,3}` não são iguais, nem subconjuntos um do outro, nem superconjuntos um do outro). Consequentemente, conjuntos não são argumentos apropriados para funções que dependem de ordenação total (por exemplo, `min()`, `max()` e `sorted()` produzem resultados indefinidos dada uma lista de conjuntos como entradas) .

  A comparação de conjuntos reforça a reflexividade de seus elementos.

- A maioria dos outros tipos embutidos não possui métodos de comparação implementados, portanto, eles herdam o comportamento de comparação padrão.

As classes definidas pelo usuário que personalizam seu comportamento de comparação devem seguir algumas regras de consistência, se possível:

- A comparação da igualdade deve ser reflexiva. Em outras palavras, objetos idênticos devem ser comparados iguais:

  `x is y` implica em `x == y`

- A comparação deve ser simétrica. Em outras palavras, as seguintes expressões devem ter o mesmo resultado:

  `x == y` e `y == x`

  `x != y` e `y != x`

  `x < y` e `y > x`

  `x <= y` e `y >= x`

- A comparação deve ser transitiva. Os seguintes exemplos (não exaustivos) ilustram isso:

  `x > y and y > z` implica em `x > z`

  `x < y and y <= z` implica em `x < z`

- A comparação inversa deve resultar na negação booleana. Em outras palavras, as seguintes expressões devem ter o mesmo resultado:

  `x == y` e `not x != y`

  `x < y` e `not x >= y` (pra classificação total)

  `x > y` e `not x <= y` (pra classificação total)

  As duas últimas expressões aplicam-se a coleções totalmente ordenadas (por exemplo, a sequências, mas não a conjuntos ou mapeamentos). Veja também o decorador `total_ordering()`.

- O resultado `hash()` deve ser consistente com a igualdade. Objetos iguais devem ter o mesmo valor de hash ou ser marcados como não-hasheáveis.

Python não impõe essas regras de consistência. Na verdade, os valores não numéricos são um exemplo de não cumprimento dessas regras.

6.10.2 Operações de teste de pertinência

Os operadores *in* e *not in* testam se um operando é membro ou não de outro. `x in s` é avaliado como `True` se *x* for membro de *s*, e `False` caso contrário. `x not in s` retorna a negação de `x in s`. Todas as sequências e tipos de conjuntos embutidos oferecem suporte a isso, assim como o dicionário, para o qual `in` testa se o dicionário tem uma determinada chave. Para tipos de contêiner como list, tuple, set, frozenset, dict ou Collections.deque, a expressão `x in y` é equivalente a `any(x is e or x == e for e in y)`.

Para os tipos string e bytes, `x in y` é `True` se e somente se *x* for uma substring de *y*. Um teste equivalente é `y.find(x) != -1`. Strings vazias são sempre consideradas uma substring de qualquer outra string, então `"" in "abc"` retornará `True`.

Para classes definidas pelo usuário que definem o método *__contains__()*, `x in y` retorna `True` se `y.__contains__(x)` retorna um valor verdadeiro, e `False` caso contrário.

Para classes definidas pelo usuário que não definem *__contains__()*, mas definem *__iter__()*, `x in y` é `True` se algum valor `z`, para a qual a expressão `x is z or x == z` é verdadeira, é produzida durante a iteração sobre `y`. Se uma exceção for levantada durante a iteração, é como se *in* tivesse levantado essa exceção.

Por último, o protocolo de iteração de estilo antigo é tentado: se uma classe define *__getitem__()*, `x in y` é `True` se, e somente se, houver um índice inteiro não negativo *i* tal que `x is y[i] or x == y[i]`, e nenhum índice inteiro inferior levanta a exceção `IndexError`. (Se qualquer outra exceção for levantada, é como se *in* levantasse essa exceção).

O operador *not in* é definido para ter o valor verdade inverso de *in*.

6.10.3 Comparações de identidade

Os operadores *is* e *is not* testam a identidade de um objeto: `x is y` é verdadeiro se, e somente se, *x* e *y* são o mesmo objeto. A identidade de um objeto é determinada usando a função `id()`. `x is not y` produz o valor verdade inverso.[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
```

No contexto de operações booleanas, e também quando expressões são usadas por instruções de fluxo de controle, os seguintes valores são interpretados como falsos: `False`, `None`, zero numérico de todos os tipos e strings e contêineres vazios (incluindo strings, tuplas, listas, dicionários, conjuntos e frozensets). Todos os outros valores são interpretados como verdadeiros. Objetos definidos pelo usuário podem personalizar seu valor verdade fornecendo um método *__bool__()*.

O operador *not* produz `True` se seu argumento for falso, `False` caso contrário.

A expressão `x and y` primeiro avalia *x*; se *x* for falso, seu valor será retornado; caso contrário, *y* será avaliado e o valor resultante será retornado.

A expressão `x or y` primeiro avalia *x*; se *x* for verdadeiro, seu valor será retornado; caso contrário, *y* será avaliado e o valor resultante será retornado.

Observe que nem *and* nem *or* restringem o valor e o tipo que retornam para `False` e `True`, mas sim retornam o último argumento avaliado. Isso às vezes é útil, por exemplo, se `s` é uma string que deve ser substituída por um valor padrão se estiver vazia, a expressão `s or 'foo'` produz o valor desejado. Como *not* precisa criar um novo valor, ele retorna um valor booleano independente do tipo de seu argumento (por exemplo, `not 'foo'` produz `False` em vez de `''`.)

[4] Devido à coleta de lixo automática, às listas livres e à natureza dinâmica dos descritores, você pode notar um comportamento aparentemente incomum em certos usos do operador *is*, como aqueles que envolvem comparações entre métodos de instância ou constantes. Confira a documentação para obter mais informações.

6.12 Expressões de atribuição

```
assignment_expression ::= [identifier ":="] expression
```

Uma expressão de atribuição (às vezes também chamada de "expressão nomeada" ou "morsa") atribui um *expression* a um *identifier*, ao mesmo tempo que retorna o valor de *expression*.

Um caso de uso comum é ao lidar com expressões regulares correspondentes:

```
if matching := pattern.search(data):
    do_something(matching)
```

Ou, ao processar um fluxo de arquivos em partes:

```
while chunk := file.read(9000):
    process(chunk)
```

As expressões de atribuição devem ser colocadas entre parênteses quando usadas como instruções de expressão e quando usadas como subexpressões em expressões de fatiamento, condicionais, de lambda, de argumento nomeado e de if de compreensão e em instruções `assert`, `with` e `assignment`. Em todos os outros lugares onde eles podem ser usados, os parênteses não são necessários, inclusive nas instruções `if` e `while`.

Adicionado na versão 3.8: Veja **PEP 572** para mais detalhes sobre expressões de atribuição.

6.13 Expressões condicionais

```
conditional_expression ::= or_test ["if" or_test "else" expression]
expression             ::= conditional_expression | lambda_expr
```

Expressões condicionais (às vezes chamadas de "operador ternário") têm a prioridade mais baixa de todas as operações Python.

A expressão `x if C else y` primeiro avalia a condição, *C* em vez de *x*. Se *C* for verdadeiro, *x* é avaliado e seu valor é retornado; caso contrário, *y* será avaliado e seu valor será retornado.

Veja **PEP 308** para mais detalhes sobre expressões condicionais.

6.14 Lambdas

```
lambda_expr ::= "lambda" [parameter_list] ":" expression
```

Expressões lambda (às vezes chamadas de funções lambda) são usadas para criar funções anônimas. A expressão `lambda parameters: expression` produz um objeto função. O objeto sem nome se comporta como um objeto de função definido com:

```
def <lambda>(parâmetros):
    return expressão
```

Veja a seção *Definições de função* para a sintaxe das listas de parâmetros. Observe que as funções criadas com expressões lambda não podem conter instruções ou anotações.

6.15 Listas de expressões

```
starred_expression       ::= ["*"] or_expr
flexible_expression      ::= assignment_expression | starred_expression
flexible_expression_list ::= flexible_expression ("," flexible_expression)* [","]
starred_expression_list  ::= starred_expression ("," starred_expression)* [","]
expression_list          ::= expression ("," expression)* [","]
```

```
yield_list              ::= expression_list | starred_expression "," [starred_expression_list]
```

Exceto quando parte de uma sintaxe de criação de lista ou conjunto, uma lista de expressões contendo pelo menos uma vírgula produz uma tupla. O comprimento da tupla é o número de expressões na lista. As expressões são avaliadas da esquerda para a direita.

Um asterisco `*` denota *desempacotamento de iterável*. Seu operando deve ser um *iterável*. O iterável é expandido em uma sequência de itens, que são incluídos na nova tupla, lista ou conjunto, no local do desempacotamento.

Adicionado na versão 3.5: Desempacotamento de iterável em listas de expressões, originalmente proposta pela **PEP 448**.

Adicionado na versão 3.11: Qualquer item em uma lista de expressões pode ser estrelado. Veja a **PEP 646**.

Uma vírgula final é necessária apenas para criar uma tupla de um item, como `1,`; é opcional em todos os outros casos. Uma única expressão sem vírgula final não cria uma tupla, mas produz o valor dessa expressão. (Para criar uma tupla vazia, use um par vazio de parênteses: `()`.)

6.16 Ordem de avaliação

Python avalia expressões da esquerda para a direita. Observe que ao avaliar uma tarefa, o lado direito é avaliado antes do lado esquerdo.

Nas linhas a seguir, as expressões serão avaliadas na ordem aritmética de seus sufixos:

```
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

A tabela a seguir resume a precedência de operadores no Python, da precedência mais alta (mais vinculativa) à precedência mais baixa (menos vinculativa). Os operadores na mesma caixa têm a mesma precedência. A menos que a sintaxe seja fornecida explicitamente, os operadores são binários. Os operadores na mesma caixa agrupam-se da esquerda para a direita (exceto exponenciação e expressões condicionais, que agrupam da direita para a esquerda).

Observe que comparações, testes de pertinência e testes de identidade têm todos a mesma precedência e possuem um recurso de encadeamento da esquerda para a direita, conforme descrito na seção *Comparações*.

Operador	Descrição
`(expressions...)`, `[expressões...]`, `{chave: valor...}`, `{expressões...}`	Expressão entre parênteses ou de ligação, sintaxe de criação de lista, sintaxe de criação de dicionário, sintaxe de criação de conjunto
`x[índice]`, `x[índice:índice]`, `x(argumentos...)`, `x.atributo`	subscrição, fatiamento, chamada, referência a atributo
`await x`	Expressão await
`**`	Exponenciação[5]
`+x`, `-x`, `~x`	positivo, negativo, NEGAÇÃO (NOT) bit a bit
`*`, `@`, `/`, `//`, `%`	Multiplicação, multiplicação de matrizes, divisão, divisão pelo piso, resto[6]
`+`, `-`	Adição e subtração
`<<`, `>>`	Deslocamentos
`&`	E (AND) bit a bit
`^`	OU EXCLUSIVO (XOR) bit a bit
`|`	OU (OR) bit a bit
`in`, *`not in`*, *`is`*, *`is not`*, `<`, `<=`, `>`, `>=`, `!=`, `==`	Comparações, incluindo testes de pertinência e testes de identidade
`not x`	NEGAÇÃO (NOT) booleana
`and`	E (AND) booleano
`or`	OU (OR) booleano
`if` – `else`	Expressão condicional
`lambda`	Expressão lambda
`:=`	Expressão de atribuição

[5] O operador de potência `**` liga-se com menos força do que um operador aritmético ou unário bit a bit à sua direita, ou seja, `2**-1` é `0.5`.
[6] O operador `%` também é usado para formatação de strings; a mesma precedência se aplica.

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             # variável de classe
inst = Cls()
inst.x = inst.x + 1   # escreve inst.x como 4 deixando Cls.x como 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).

  Para objetos definidos pelo usuário, o método *`__setitem__()`* é chamado com argumentos apropriados.

- 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.

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 é atualizado e, em seguida, x[i] é atualizado
print(x)
```

Ver também

PEP 3132 - Desempacotamento estendido de iterável
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 desempacotamento) 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 instrução 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.

O alvo da atribuição é considerado "simples" se consistir em um único nome que não esteja entre parênteses. Para alvos de atribuição simples, 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.

Se o alvo da atribuição não for simples (um atributo, nó subscrito ou nome entre parênteses), a anotação será avaliada se estiver no escopo da classe ou do módulo, mas não será armazenada.

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.

Se o lado direito estiver presente, uma atribuição anotada executa a atribuição real antes de avaliar as anotações (quando aplicável). Se o lado direito não estiver presente para um alvo de expressão, então o interpretador avalia o alvo, exceto para a última chamada *`__setitem__()`* ou *`__setattr__()`*.

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 presumem 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    # uma função que faz nada (ainda)

class C: pass       # uma classe com nenhum método (ainda)
```

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.

Um objeto traceback (situação da pilha de execução) normalmente é criado automaticamente quando uma exceção é levantada e anexada a ele como o atributo `__traceback__`. Você pode criar uma exceção e definir seu próprio traceback em uma etapa usando o método de exceção `with_traceback()` (que retorna a mesma instância de exceção, com seu traceback definido para seu argumento), assim:

```
raise Exception("ocorreu foo").with_traceback(tracebackobj)
```

A cláusula `from` é usada para encadeamento de exceções: se fornecida, a segunda expressão, *expression*, deve ser outra classe ou instância de exceção. Se a segunda expressão for uma instância de exceção, ela será anexada à exceção levantada como o atributo `__cause__` (que é gravável). Se a expressão for uma classe de exceção, a classe será instanciada e a instância de exceção resultante será anexada à exceção levantada como o atributo `__cause__`. Se a exceção levantada não for tratada, ambas as exceções serão impressas:

```
>>> 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
```

Um mecanismo semelhante funciona implicitamente se uma nova exceção for levantada quando uma exceção já estiver sendo tratada. Uma exceção pode ser tratada quando uma cláusula *except* ou *finally*, ou uma instrução *with*, é usada. A exceção anterior é então anexada como o atributo `__context__` da nova exceção:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened")
...
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
```

O encadeamento de exceção pode ser explicitamente suprimido especificando `None` na cláusula `from`:

```
>>> 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`.

Adicionado o atributo `__suppress_context__` para suprimir a exibição automática do contexto de exceção.

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 importado e vinculado localmente
import foo.bar.baz         # foo, foo.bar e foo.bar.baz importados, foo vinculado␣
↪localmente
import foo.bar.baz as fbb  # foo, foo.bar e foo.bar.baz importados, foo.bar.baz␣
↪vinculado como fbb
from foo.bar import baz    # foo, foo.bar e foo.bar.baz importados, foo.bar.baz␣
↪vinculado como baz
from foo import attr       # foo importado e foo.attr vinculado como 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 *Importações relativas ao pacote*.

`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 os 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.

O código compilado por chamadas para as funções embutidas `exec()` e `compile()` que ocorrem em um módulo `M` contendo uma instrução future usará, por padrão, a nova sintaxe ou semântica associada com a instrução future. Isso pode ser controlado por argumentos opcionais para `compile()` – veja a documentação dessa função para detalhes.

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* faz com que os identificadores listados a serem interpretados como globais. Seria impossível atribuir a uma variável global sem `global`, embora variáveis livres possam se referir a globais sem serem declaradas globais.

A instrução *global* se aplica a todo o escopo de uma função ou corpo da classe. Uma exceção `SyntaxError` é levantada se uma variável for usada ou atribuída antes de sua declaração global no escopo.

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)*
```

Quando a definição de uma função ou classe está aninhada (incluída) nas definições de outras funções, seus escopos não locais são os escopos locais das funções envolventes. A instrução *nonlocal* faz com que os identificadores listados se refiram a nomes previamente vinculados a escopos não locais. Ele permite que o código encapsulado religue esses identificadores não locais. Se um nome estiver ligado a mais de um escopo não local, a ligação mais próxima será usada. Se um nome não estiver vinculado a nenhum escopo não local, ou se não houver escopo não local, uma exceção `SyntaxError` será levantada.

A instrução *nonlocal* se aplica a todo o escopo de uma função ou corpo da classe. Uma exceção `SyntaxError` é levantada se uma variável for usada ou atribuída antes de sua declaração nonlocal no escopo.

Ver também

PEP 3104 - Acesso a nomes em escopos externos
A especificação para a instrução *nonlocal*.

Nota do programador: *nonlocal* é uma diretiva para o analisador sintático e se aplica apenas ao código analisado junto com ele. Veja a nota para a instrução *global*.

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 *Apelidos de tipo genérico* para mais.

`type` é uma *palavra reservada contextual*.

Adicionado 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 opcionais 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             # não afeta o laço for, porque i será
                      # substituída pelo índice seguinte no 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`

As cláusulas `except` especificam um ou mais manipuladores de exceção. Quando nenhuma exceção ocorre na cláusula *`try`*, nenhum manipulador de exceção é executado. Quando uma exceção ocorre no conjunto `try`, uma busca por um manipulador de exceção é iniciada. Essa busca inspeciona as cláusulas `except` por vez até que uma seja encontrada que corresponda à exceção. Uma cláusula `except` sem expressão, se presente, deve ser a última; ela corresponde a qualquer exceção.

Para uma cláusula `except` com uma expressão, a expressão deve ser avaliada como um tipo de exceção ou uma tupla de tipos de exceção. A exceção levantada corresponde a uma cláusula `except` cuja expressão é avaliada como a classe ou uma *classe base não virtual* do objeto exceção, ou como uma tupla que contém tal classe.

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).

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

[1] A exceção é propagada para a pilha de invocação, a menos que haja uma cláusula *`finally`* que por acaso levante outra exceção. Essa nova exceção faz com que a antiga seja perdida.

```
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 Cláusula `except*`

As cláusulas `except*` são usadas para manipular `ExceptionGroups`. O tipo de exceção para correspondência é interpretado como no caso de *except*, mas no caso de grupos de exceção, podemos ter correspondências parciais quando o tipo corresponde a algumas das exceções no grupo. Isso significa que várias cláusulas `except*` podem ser executadas, cada uma manipulando parte do grupo de exceções. Cada cláusula é executada no máximo uma vez e manipula um grupo de exceções de todas as exceções correspondentes. Cada exceção no grupo é manipulada por no máximo uma cláusula `except*`, a primeira que corresponde a ela.

```
>>> 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:
...     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))
```

(continua na próxima página)

(continuação da página anterior)

```
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: eg
  +-+---------------- 1 ----------------
    | ValueError: 1
    +------------------------------------
```

Quaisquer exceções restantes que não foram manipuladas por nenhuma cláusula `except*` são levantadas novamente no final, junto com todas as exceções que foram levantadas de dentro das cláusulas `except*`. Se esta lista contiver mais de uma exceção para levantar novamente, elas serão combinadas em um grupo de exceções.

Se a exceção levantada não for um grupo de exceções e seu tipo corresponder a uma das cláusulas `except*`, ela será capturada e encapsulada por um grupo de exceções com uma string de mensagem vazia.

```
>>> try:
...     raise BlockingIOError
... except* BlockingIOError as e:
...     print(repr(e))
...
ExceptionGroup('', (BlockingIOError()))
```

Uma cláusula `except*` deve ter uma expressão correspondente; não pode ser `except*:`. Além disso, essa expressão não pode conter tipos de grupo de exceção, porque isso teria semântica ambígua.

Não é possível misturar *except* e `except*` no mesmo *try*. *break*, *continue* e *return* não pode aparecer em uma cláusula `except*`.

8.4.3 Cláusula `else`

A cláusula opcional `else` é executada se o fluxo de controle deixar o conjunto *try*, nenhuma exceção foi levantada e nenhuma instrução *return*, *continue* ou *break* foi executada. Exceções na cláusula `else` não são manipuladas pelas cláusulas *except* precedentes.

8.4.4 Cláusula `finally`

Se `finally` estiver presente, especifica um manipulador de "limpeza". A cláusula *try* é executada, incluindo quaisquer cláusulas *except* e *else*. Se uma exceção ocorrer em qualquer uma das cláusulas e não for manipulada, a exceção será salva temporariamente. A cláusula `finally` é executada. Se houver uma exceção salva, ela será levantada novamente no final da cláusula `finally`. Se a cláusula `finally` levantar outra exceção, a exceção salva será definida como o contexto da nova exceção. Se a cláusula `finally` executar uma instrução *return*, *break* ou *continue*, a exceção salva será descartada:

```
>>> def f():
...     try:
...         1/0
...     finally:
...         return 42
...
>>> f()
42
```

As informações de exceção não estão disponíveis para o programa durante a execução da cláusula `finally`.

Quando uma instrução *return*, *break* ou *continue* é executada no conjunto *try* de uma instrução `try…finally`, a cláusula `finally` também é executada "na saída".

O valor de retorno de uma função é determinado pela última instrução *return* executada. Como a cláusula `finally` sempre é executada, uma instrução `return` executada na cláusula `finally` sempre será a última executada:

```
>>> def foo():
...     try:
...         return 'try'
...     finally:
...         return 'finally'
...
>>> foo()
'finally'
```

Alterado na versão 3.8: Antes do Python 3.8, uma instrução *continue* era ilegal na cláusula `finally` devido a um problema com a implementação.

8.5 A instrução `with`

A instrução *with* é usada para envolver em um invólucro a execução de um bloco com métodos definidos por um gerenciador de contexto (veja a seção *Gerenciadores de contexto da instrução with*). Isso permite que padrões comuns de uso de *try*…*except*…*finally* sejam encapsulados para reutilização conveniente.

```
with_stmt          ::= "with" ( "(" with_stmt_contents ","? ")" | with_stmt_contents ) ":" suite
with_stmt_contents ::= with_item (","  with_item)*
with_item          ::= expression ["as" target]
```

A execução da instrução *with* com um "item" ocorre da seguinte maneira:

1. A expressão de contexto (a expressão fornecida em *with_item*) é avaliada para obter um gerenciador de contexto.
2. O *__enter__()* do gerenciador de contexto é carregado para uso posterior.
3. O *__exit__()* do gerenciador de contexto é carregado para uso posterior.
4. O método *__enter__()* do gerenciador de contexto é invocado.
5. Se um alvo foi incluído na instrução *with*, o valor de retorno de *__enter__()* é atribuído a ele.

   > **Nota**
   >
   > A instrução *with* garante que se o método *__enter__()* retornar sem um erro, então *__exit__()* sempre será chamado. Assim, se ocorrer um erro durante a atribuição à lista de alvos, ele será tratado da mesma forma que um erro ocorrendo dentro do conjunto seria. Veja a etapa 7 abaixo.

6. O conjunto é executado.
7. O método *__exit__()* do gerenciador de contexto é invocado. Se uma exceção fez com que o conjunto fosse encerrado, seu tipo, valor e traceback são passados como argumentos para *__exit__()*. Caso contrário, três argumentos `None` são fornecidos.

   Se o conjunto foi encerrado devido a uma exceção, e o valor de retorno do método *__exit__()* foi falso, a exceção é levantada novamente. Se o valor de retorno era verdadeiro, a exceção é suprimida, e a execução continua com a instrução após a instrução *with*.

   Se o conjunto foi encerrado por qualquer motivo diferente de uma exceção, o valor de retorno de *__exit__()* é ignorado e a execução prossegue no local normal para o tipo de saída que foi realizada.

O seguinte código:

```
with EXPRESSION as TARGET:
    SUITE
```

é semanticamente equivalente a:

```
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)
```

Com mais de um item, os gerenciadores de contexto são processados como se várias instruções *with* estivessem aninhadas:

```
with A() as a, B() as b:
    SUITE
```

é semanticamente equivalente a:

```
with A() as a:
    with B() as b:
        SUITE
```

Você também pode escrever gerenciadores de contexto multi-item em várias linhas se os itens estiverem entre parênteses. Por exemplo:

```
with (
    A() as a,
    B() as b,
):
    SUITE
```

Alterado na versão 3.1: Suporte para múltiplas expressões de contexto.

Alterado na versão 3.10: Suporte para usar parênteses de agrupamento para dividir a instrução em várias linhas.

Ver também

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

8.6 A instrução `match`

Adicionado na versão 3.10.

A instrução match é usada para correspondência de padrões. Sintaxe:

```
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

Esta seção usa aspas simples para denotar *palavras reservadas contextuais*.

A correspondência de padrões aceita um padrão como entrada (seguindo `case`) e um valor de sujeito (seguindo `match`). O padrão (que pode conter subpadrões) é correspondido ao valor de assunto. Os resultados são:

- Um sucesso ou falha de correspondência (também chamado de sucesso ou falha de padrão).
- Possível vinculação de valores correspondentes a um nome. Os pré-requisitos para isso são discutidos mais adiante.

As palavras reservadas `match` e `case` são *palavras reservadas contextuais*.

Ver também

- **PEP 634** – Structural Pattern Matching: Specification
- **PEP 636** – Correspondência de padrões estruturais: Tutorial

8.6.1 Visão Geral

Aqui está uma visão geral do fluxo lógico de uma instrução match:

1. A expressão de sujeito `subject_expr` é avaliada e um valor de sujeito resultante é obtido. Se a expressão de sujeito contiver uma vírgula, uma tupla é construída usando as regras padrão.
2. Cada padrão em um `case_block` é tentado para corresponder ao valor de sujeito. As regras específicas para sucesso ou falha são descritas abaixo. A tentativa de correspondência também pode vincular alguns ou todos os nomes autônomos dentro do padrão. As regras precisas de vinculação de padrão variam por tipo de padrão e são especificadas abaixo. **As vinculações de nome feitas durante uma correspondência de padrão bem-sucedida sobrevivem ao bloco executado e podem ser usadas após a instrução match**.

   Nota

   Durante correspondências de padrões com falha, alguns subpadrões podem ter sucesso. Não confie em vinculações sendo feitas para uma correspondência com falha. Por outro lado, não confie em variáveis permanecendo inalteradas após uma correspondência com falha. O comportamento exato depende da implementação e pode variar. Esta é uma decisão intencional feita para permitir que diferentes implementações adicionem otimizações.

3. Se o padrão for bem-sucedido, o *guard* correspondente (se presente) é avaliado. Neste caso, todas as vinculações de nome são garantidas como tendo acontecido.
   - Se o *guard* for avaliado como verdadeiro ou estiver ausente, o `block` dentro de `case_block` será executado.
   - Caso contrário, o próximo `case_block` será tentado conforme descrito acima.
   - Se não houver mais blocos de caso, a instrução match será concluída.

Nota

Os usuários geralmente nunca devem confiar em um padrão sendo avaliado. Dependendo da implementação, o interpretador pode armazenar valores em cache ou usar outras otimizações que pulam avaliações repetidas.

Um exemplo de instrução match:

```
>>> flag = False
>>> match (100, 200):
...     case (100, 300):  # Não corresponde: 200 != 300
...         print('Case 1')
...     case (100, 200) if flag:  # Corresponde com sucesso, mas o guard falha...
        print('Case 2')
...     case (100, y):  # Corresponde e vincula y a 200
...         print(f'Case 3, y: {y}')
...     case _:  # Padrão não testado
...         print('Case 4, I match anything!')
...
Case 3, y: 200
```

Neste caso, `if flag` é um *guard*. Leia mais sobre isso na próxima seção.

8.6.2 Guards

```
guard ::= "if" named_expression
```

Um `guard` (que faz parte do `case`) deve ter sucesso para que o código dentro do bloco `case` seja executado. Ele assume a forma: *`if`* seguido por uma expressão.

O fluxo lógico de um bloco `case` com um `guard` é o seguinte:

1. Verifique se o padrão no bloco `case` foi bem-sucedido. Se o padrão falhou, o `guard` não é avaliado e o próximo bloco `case` é verificado.
2. Se o padrão for bem-sucedido, avalia o `guard`.
   - Se a condição `guard` for avaliada como verdadeira, o bloco de caso será selecionado.
   - Se a condição `guard` for avaliada como falsa, o bloco de caso não será selecionado.
   - Se o `guard` levantar uma exceção durante a avaliação, a exceção surgirá.

Guards podem ter efeitos colaterais, pois são expressões. A avaliação de guards deve prosseguir do primeiro ao último bloco de caso, um de cada vez, pulando blocos de caso cujos padrões não são todos bem-sucedidos. (Isto é, a avaliação de guardas deve acontecer em ordem.) A avaliação de guards deve parar quando um bloco de caso for selecionado.

8.6.3 Blocos irrefutáveis de case

Um bloco irrefutável de case é um bloco de case que corresponde a qualquer valor. Uma instrução match pode ter no máximo um bloco irrefutável de case, e ele deve ser o último.

Um bloco de case é considerado irrefutável se não tiver guard e seu padrão for irrefutável. Um padrão é considerado irrefutável se pudermos provar somente por sua sintaxe que ele sempre terá sucesso. Somente os seguintes padrões são irrefutáveis:

- *Padrões AS* cujo lado esquerdo é irrefutável
- *Padrões OR* contendo pelo menos um padrão irrefutável
- *Padrões de captura*
- *Padrões curingas*
- padrões irrefutáveis entre parênteses

8.6.4 Padrões

Nota

Esta seção usa notações de gramática para além do padrão de EBNF:

- a notação `SEP.RULE+` é uma abreviação para `RULE (SEP RULE)*`
- a notação `!RULE` é uma abreviação para uma asserção de negação antecipada.

Esta é a sintaxe de nível superior para `patterns` (padrões):

```
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
```

As descrições abaixo incluirão uma descrição "em termos simples" de o que o padrão faz para fins ilustrativos (créditos a Raymond Hettinger pelo documento que inspirou a maioria das descrições). Note que essas descrições são puramente para fins ilustrativos, e **não necessariamente** refletem a implementação subjacente. Além disso, elas não cobrem todas as formas válidas.

Padrões OR

Um padrão OR é composto por dois ou mais padrões separados por barras verticais `|`. Sintaxe:

```
or_pattern ::= "|".closed_pattern+
```

Somente o último subpadrão pode ser *irrefutável*, e cada subpadrão deve vincular o mesmo conjunto de nomes para evitar ambiguidades.

Um padrão OR testa a correspondência de cada um dos seus subpadrões, em sequência, ao valor do sujeito, até que uma delas seja bem sucedida. O padrão OR é então considerado bem sucedido. Caso contrário, se todas elas falharam, o padrão OR falhou.

Em termos simples, `P1 | P2 | ...` vai tentar fazer corresponder `P1`, se falhar vai tentar `P2`, declarando sucesso se houver sucesso em qualquer uma das tentativas, e falhando caso contrário.

Padrões AS

Um padrão AS corresponde a um padrão OR à esquerda da palavra reservada `as` de um assunto. Sintaxe:

```
as_pattern ::= or_pattern "as" capture_pattern
```

Se o padrão OR falhar, o padrão AS falhará. Caso contrário, o padrão AS vincula o assunto ao nome à direita da palavra-chave as e obtém sucesso. `capture_pattern` não pode ser um _.

Em termos simples, `P as NAME` corresponderá a `P` e, em caso de sucesso, definirá `NAME = <assunto>`.

Padrões literais

Um padrão literal corresponde à maioria dos *literais* em Python. Sintaxe:

```
literal_pattern ::= signed_number
                  | signed_number "+" NUMBER
                  | signed_number "-" NUMBER
                  | strings
                  | "None"
```

```
                     | "True"
                     | "False"
signed_number    ::= ["-"] NUMBER
```

A regra `strings` e o token `NUMBER` são definidos na *gramática Python padrão*. Strings entre aspas triplas são suportadas. Strings brutas e strings de bytes são suportadas. *Literais de strings formatadas* não são suportadas.

As formas `signed_number '+' NUMBER` e `signed_number '-' NUMBER` são para expressar *números complexos*; elas requerem um número real à esquerda e um número imaginário à direita. Por exemplo, `3 + 4j`.

Em termos simples, `LITERAL` terá sucesso somente se `<assunto> == LITERAL`. Para os singletons `None`, `True` e `False`, o operador *`is`* é usado.

Padrões de captura

Um padrão de captura vincula o valor do assunto a um nome. Sintaxe:

```
capture_pattern ::= !'_' NAME
```

Um único sublinhado _ não é um padrão de captura (é o que `!'_'` expressa). Em vez disso, ele é tratado como um *`wildcard_pattern`*.

Em um determinado padrão, um determinado nome só pode ser vinculado uma vez. Por exemplo, `case x, x: ...` é inválido enquanto `case [x] | x: ...` é permitido.

Os padrões de captura sempre são bem-sucedidos. A vinculação segue regras de escopo estabelecidas pelo operador de expressão de atribuição na **PEP 572**; o nome se torna uma variável local no escopo de função de contenção mais próximo, a menos que haja uma instrução *`global`* ou *`nonlocal`* aplicável.

Em termos simples, `NAME` sempre terá sucesso e definirá `NAME = <assunto>`.

Padrões curingas

Um padrão curinga sempre tem sucesso (corresponde a qualquer coisa) e não vincula nenhum nome. Sintaxe:

```
wildcard_pattern ::= '_'
```

_ é uma *palavra reservada contextual* dentro de qualquer padrão, mas somente dentro de padrões. É um identificador, como de costume, mesmo dentro de expressões de assunto `match`s, `guard`s e blocos `case`.

Em termos simples, _ sempre terá sucesso.

Padrões de valor

Um padrão de valor representa um valor nomeado em Python. Sintaxe:

```
value_pattern ::= attr
attr          ::= name_or_attr "." NAME
name_or_attr  ::= attr | NAME
```

O nome pontilhado no padrão é pesquisado usando as *regras de resolução de nomes* padrão do Python. O padrão é bem-sucedido se o valor encontrado for comparado igual ao valor do assunto (usando o operador de igualdade ==).

Em termos simples, `NAME1.NAME2` terá sucesso somente se `<assunto> == NAME1.NAME2`

> **Nota**
>
> Se o mesmo valor ocorrer várias vezes na mesma instrução match, o interpretador pode armazenar em cache o primeiro valor encontrado e reutilizá-lo em vez de repetir a mesma pesquisa. Esse cache é estritamente vinculado a uma determinada execução de uma determinada instrução match.

Padrões de grupo

Um padrão de grupo permite que os usuários adicionem parênteses em torno de padrões para enfatizar o agrupamento pretendido. Caso contrário, não há sintaxe adicional. Sintaxe:

```
group_pattern ::= "(" pattern ")"
```

Em termos simples, `(P)` tem o mesmo efeito que `P`.

Padrões de sequência

Um padrão de sequência contém vários subpadrões a serem correspondidos com elementos de sequência. A sintaxe é similar ao desempacotamento de uma lista ou tupla.

```
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)
```

Não há diferença se parênteses ou colchetes são usados para padrões de sequência (por exemplo, `(...)` vs `[...]`).

Nota

Um único padrão entre parênteses sem uma vírgula final (por exemplo, `(3 | 4)`) é um *padrão de grupo*. Enquanto um único padrão entre colchetes (por exemplo, `[3 | 4]`) ainda é um padrão de sequência.

No máximo um subpadrão de estrela pode estar em um padrão de sequência. O subpadrão de estrela pode ocorrer em qualquer posição. Se nenhum subpadrão de estrela estiver presente, o padrão de sequência é um padrão de sequência de comprimento fixo; caso contrário, é um padrão de sequência de comprimento variável.

A seguir está o fluxo lógico para corresponder um padrão de sequência com um valor de assunto:

1. Se o valor do assunto não for uma sequência[2], o padrão de sequência falhará.
2. Se o valor do assunto for uma instância de `str`, `bytes` ou `bytearray`, o padrão de sequência falhará.
3. As etapas subsequentes dependem se o padrão de sequência é fixo ou de comprimento variável.

   Se o padrão de sequência for de comprimento fixo:

   1. Se o comprimento da sequência do assunto não for igual ao número de subpadrões, o padrão da sequência falha
   2. Subpadrões no padrão de sequência são correspondidos aos seus itens correspondentes na sequência de assunto da esquerda para a direita. A correspondência para assim que um subpadrão falha. Se todos

[2] Na correspondência de padrões, uma sequência é definida como uma das seguintes:
- uma classe que herda de `collections.abc.Sequence`
- uma classe Python que foi registrada como `collections.abc.Sequence`
- uma classe embutida que tem seu bit `Py_TPFLAGS_SEQUENCE` (CPython) definido
- uma classe que herda de qualquer uma das anteriores

As seguintes classes de biblioteca padrão são sequências:
- `array.array`
- `collections.deque`
- `list`
- `memoryview`
- `range`
- `tuple`

Nota

Valores de assunto do tipo `str`, `bytes` e `bytearray` não correspondem aos padrões de sequência.

os subpadrões tiverem sucesso em corresponder ao seu item correspondente, o padrão de sequência é bem-sucedido.

Caso contrário, se o padrão de sequência for de comprimento variável:

1. Se o comprimento da sequência do assunto for menor que o número de subpadrões não-estrela, o padrão da sequência falha.
2. Os principais subpadrões não estelares são correspondidos aos seus itens correspondentes, como nas sequências de comprimento fixo.
3. Se a etapa anterior for bem-sucedida, o subpadrão estrela corresponde a uma lista formada pelos itens de assunto restantes, excluindo os itens restantes correspondentes aos subpadrões não-estrela que seguem o subpadrão estrela.
4. Os subpadrões não-estrela restantes são correspondidos aos seus itens de assunto correspondentes, como em uma sequência de comprimento fixo.

> **Nota**
>
> O comprimento da sequência de assunto é obtido via `len()` (ou seja, via protocolo `__len__()`). Esse comprimento pode ser armazenado em cache pelo interpretador de forma similar a *padrões de valor*.

Em termos simples, `[P1, P2, P3, … , P<N>]` corresponde somente se tudo o seguinte acontecer:

- verifica se `<subject>` é uma sequência
- `len(subject) == <N>`
- `P1` corresponde a `<subject>[0]` (observe que esta correspondência também pode vincular nomes)
- `P2` corresponde a `<subject>[1]` (observe que esta correspondência também pode vincular nomes)
- … e assim por diante para o padrão/elemento correspondente.

Padrões de mapeamento

Um padrão de mapeamento contém um ou mais padrões de chave-valor. A sintaxe é similar à construção de um dicionário. Sintaxe:

```
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
```

No máximo um padrão de estrela dupla pode estar em um padrão de mapeamento. O padrão de estrela dupla deve ser o último subpadrão no padrão de mapeamento.

Chaves duplicadas em padrões de mapeamento não são permitidas. Chaves literais duplicadas levantarão um `SyntaxError`. Duas chaves que de outra forma têm o mesmo valor levantarão `ValueError` em tempo de execução.

A seguir está o fluxo lógico para comparar um padrão de mapeamento com um valor de assunto:

1. Se o valor do assunto não for um mapeamento[3], o padrão de mapeamento falhará.
2. Se cada chave fornecida no padrão de mapeamento estiver presente no mapeamento de assunto, e o padrão para cada chave corresponder ao item correspondente do mapeamento de assunto, o padrão de mapeamento será bem-sucedido.

[3] Na correspondência de padrões, um mapeamento é definido como uma das seguintes:
- uma classe que herda de `collections.abc.Mapping`
- uma classe Python que foi registrada como `collections.abc.Mapping`
- uma classe embutida que tem seu bit `Py_TPFLAGS_MAPPING` (CPython) definido
- uma classe que herda de qualquer uma das anteriores

As classes de biblioteca padrão `dict` e `types.MappingProxyType` são mapeamentos.

3. Se chaves duplicadas forem detectadas no padrão de mapeamento, o padrão será considerado inválido. Uma exceção `SyntaxError` é levantada para valores literais duplicados; ou `ValueError` para chaves nomeadas do mesmo valor.

> **Nota**
>
> Os pares de chave-valor são correspondidos usando o formato de dois argumentos do método `get()` do assunto do mapeamento. Os pares de chave-valor correspondidos já devem estar presentes no mapeamento e não devem ser criados em tempo de uso via `__missing__()` ou *`__getitem__()`*.

Em termos simples, `{KEY1: P1, KEY2: P2, ... }` corresponde somente se tudo o seguinte acontecer:

- verifica se `<assunto>` é um mapeamento
- `KEY1 in <assunto>`
- `P1` corresponde a `<assunto>[KEY1]`
- … e assim por diante para o par KEY/elemento correspondente.

Padrões de classe

Um padrão de classe representa uma classe e seus argumentos nomeados e posicionais (se houver). Sintaxe:

```
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
```

O mesmo argumento nomeado não deve ser repetido em padrões de classe.

A seguir está o fluxo lógico para corresponder a um padrão de classe com um valor de assunto:

1. Se `name_or_attr` não for uma instância do tipo embutido `type` , levanta `TypeError`.
2. Se o valor do assunto não for uma instância de `name_or_attr` (testado via `isinstance()`), o padrão de classe falhará.
3. Se nenhum argumento de padrão estiver presente, o padrão é bem-sucedido. Caso contrário, as etapas subsequentes dependem se os padrões de argumento posicional ou nomeado estão presentes.

   Para vários tipos embutidos (especificados abaixo), um único subpadrão posicional é aceito, o qual corresponderá a todo o assunto; para esses tipos, os padrões de argumentos nomeados também funcionam como para outros tipos.

   Se apenas padrões de argumentos nomeados estiverem presentes, eles serão processados da seguinte forma, um por um:

   I. A palavra-chave é procurada como um atributo no assunto.

   - Se isso levantar uma exceção diferente de `AttributeError`, a exceção será exibida.
   - Se isso levantar `AttributeError`, o padrão de classe falhou.
   - Caso contrário, o subpadrão associado ao padrão de argumento nomeado é correspondido ao valor de atributo do sujeito. Se isso falhar, o padrão de classe falha; se isso for bem-sucedido, a correspondência prossegue para o próximo argumento nomeado.

   II. Se todos os padrões de argumento nomeado forem bem-sucedidos, o padrão de classe será bem-sucedido.

   Se houver algum padrão posicional presente, ele será convertido em padrões de argumento nomeado usando o atributo *`__match_args__`* na classe `name_or_attr` antes da correspondência:

   I. O equivalente de `getattr(cls, "__match_args__", ())` é chamado.

- Se isso levantar uma exceção, a exceção surgirá.
- Se o valor retornado não for uma tupla, a conversão falhará e `TypeError` será levantada.
- Se houver mais padrões posicionais do que `len(cls.__match_args__)`, `TypeError` será levantada.
- Caso contrário, o padrão posicional `i` é convertido em um padrão de argumento nomeado usando `__match_args__[i]` como argumento nomeado. `__match_args__[i]` deve ser uma string; caso contrário, `TypeError` é levantada.
- Se houver argumentos nomeados duplicados, `TypeError` será levantada.

Ver também

Customizando argumentos posicionais na classe correspondência de padrão

II. Uma vez que todos os padrões posicionais foram convertidos em padrões de argumentos nomeados, a partida prossegue como se houvesse apenas padrões de argumentos nomeados.

Para os seguintes tipos embutidos, o tratamento de subpadrões posicionais é diferente:

- `bool`
- `bytearray`
- `bytes`
- `dict`
- `float`
- `frozenset`
- `int`
- `list`
- `set`
- `str`
- `tuple`

Essas classes aceitam um único argumento posicional, e o padrão ali é correspondido ao objeto inteiro em vez de um atributo. Por exemplo, `int(0|1)` corresponde ao valor `0`, mas não ao valor `0.0`.

Em termos simples, `CLS(P1, attr=P2)` corresponde somente se o seguinte acontecer:

- `isinstance(<assunto>, CLS)`
- converte `P1` em um padrão de argumento nomeado usando `CLS.__match_args__`
- Para cada argumento de palavra-chave `attr=P2`:
  - `hasattr(<assunto>, "attr")`
  - `P2` corresponde a `<assunto>.attr`
- … e assim por diante para o par argumento nomeado/elemento correspondente.

Ver também

- **PEP 634** – Structural Pattern Matching: Specification
- **PEP 636** – Correspondência de padrões estruturais: Tutorial

8.7 Definições de função

Uma definição de função define um objeto de função definido pelo usuário (veja a seção *A hierarquia de tipos padrão*):

```
funcdef                   ::= [decorators] "def" funcname [type_params] "(" [parameter_list] ")"
                              ["->" expression] ":" suite
decorators                ::= decorator+
decorator                 ::= "@" assignment_expression NEWLINE
parameter_list            ::= defparameter ("," defparameter)* "," "/" ["," [parameter_list_no_p
                              | parameter_list_no_posonly
parameter_list_no_posonly ::= defparameter ("," defparameter)* ["," [parameter_list_starargs]]
                              | parameter_list_starargs
parameter_list_starargs   ::= "*" [star_parameter] ("," defparameter)* ["," [parameter_star_kwar
                              | "*" ("," defparameter)+ ["," [parameter_star_kwargs]]
                              | parameter_star_kwargs
parameter_star_kwargs     ::= "**" parameter [","]
parameter                 ::= identifier [":" expression]
star_parameter            ::= identifier [":" ["*"] expression]
defparameter              ::= parameter ["=" expression]
funcname                  ::= identifier
```

Uma definição de função é uma instrução executável. Sua execução vincula o nome da função no espaço de nomes local atual a um objeto função (um invólucro em torno do código executável para a função). Este objeto função contém uma referência ao espaço de nomes global atual como o espaço de nomes global a ser usado quando a função é chamada.

A definição da função não executa o corpo da função; ela é executada somente quando a função é chamada.[4]

Uma definição de função pode ser encapsulada por uma ou mais expressões *decoradoras*. Expressões decoradoras são avaliadas quando a função é definida, no escopo que contém a definição da função. O resultado deve ser um chamável, que é invocado com o objeto de função como o único argumento. O valor retornado é vinculado ao nome da função em vez do objeto de função. Vários decoradores são aplicados de forma aninhada. Por exemplo, o código a seguir

```
@f1(arg)
@f2
def func(): pass
```

é aproximadamente equivalente a

```
def func(): pass
func = f1(arg)(f2(func))
```

exceto que a função original não está temporariamente vinculada ao nome `func`.

Alterado na versão 3.9: Funções podem ser decoradas com qualquer `assignment_expression` válida. Anteriormente, a gramática era muito mais restritiva; veja **PEP 614** para detalhes.

Uma lista de *parâmetros de tipo* pode ser dada entre colchetes entre o nome da função e o parêntese de abertura para sua lista de parâmetros. Isso indica aos verificadores de tipo estático que a função é genérica. Em tempo de execução, os parâmetros de tipo podem ser recuperados do atributo `__type_params__` da função. Veja *Funções genéricas* para mais.

Alterado na versão 3.12: Listas de parâmetros de tipo são novas no Python 3.12.

Quando um ou mais *parâmetros* têm a forma *parameter* = *expression*, diz-se que a função tem "valores de parâmetro padrão". Para um parâmetro com um valor padrão, o *argumento* correspondente pode ser omitido de uma chamada, em cujo caso o valor padrão do parâmetro é substituído. Se um parâmetro tiver um valor padrão, todos os parâmetros seguintes até "`*`" também devem ter um valor padrão — esta é uma restrição sintática que não é expressa pela gramática.

[4] Um literal de string que aparece como a primeira instrução no corpo da função é transformado no atributo `__doc__` da função e, portanto, no *docstring* da função.

Os valores de parâmetro padrão são avaliados da esquerda para a direita quando a definição da função é executada. Isso significa que a expressão é avaliada uma vez, quando a função é definida, e que o mesmo valor "pré-calculado" é usado para cada chamada. Isso é especialmente importante para entender quando um valor de parâmetro padrão é um objeto mutável, como uma lista ou um dicionário: se a função modifica o objeto (por exemplo, anexando um item a uma lista), o valor de parâmetro padrão é efetivamente modificado. Isso geralmente não é o que se pretendia. Uma maneira de contornar isso é usar `None` como o padrão e testá-lo explicitamente no corpo da função, por exemplo:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

A semântica de chamada de função é descrita em mais detalhes na seção *Chamadas*. Uma chamada de função sempre atribui valores a todos os parâmetros mencionados na lista de parâmetros, seja de argumentos posicionais, de argumentos nomeados ou de valores padrão. Se o formato "`*identifier`" estiver presente, ele será inicializado para uma tupla que recebe quaisquer parâmetros posicionais excedentes, padronizando para a tupla vazia. Se o formato "`**identifier`" estiver presente, ele será inicializado para um novo mapeamento ordenado que recebe quaisquer argumentos nomeados excedentes, padronizando para um novo mapeamento vazio do mesmo tipo. Parâmetros após "`*`" ou "`*identifier`" são parâmetros somente-nomeados e podem ser passados somente por argumentos nomeados. Parâmetros antes de "`/`" são parâmetros somente-posicionais e podem ser passados somente por argumentos posicionais.

Alterado na versão 3.8: A sintaxe do parâmetro de função `/` pode ser usada para indicar parâmetros somente-posicionais. Veja a **PEP 570** para detalhes.

Os parâmetros podem ter uma *anotação* do formato "`: expressão`" após o nome do parâmetro. Qualquer parâmetro pode ter uma anotação, mesmo aqueles do formato `*identificador` ou `**identificador`. (Como um caso especial, parâmetros do formato `*identificador` podem ter uma anotação "`: *expressão`".) As funções podem ter uma anotação "return" do formato "`-> expressão`" após a lista de parâmetros. Essas anotações podem ser qualquer expressão Python válida. A presença de anotações não altera a semântica de uma função. Os valores de anotação estão disponíveis como valores de um dicionário indexados pelos nomes dos parâmetros no atributo `__annotations__` do objeto função. Se a importação de `annotations` de `__future__` for usada, as anotações serão preservadas como strings em tempo de execução, o que permite avaliação adiada. Caso contrário, elas são avaliadas quando a definição da função é executada. Nesse caso, as anotações podem ser avaliadas em uma ordem diferente daquela em que aparecem no código-fonte.

Alterado na versão 3.11: Parâmetros do formato "`*identificador`" podem ter uma anotação "`: *expressão`". Veja **PEP 646**.

Também é possível criar funções anônimas (funções não vinculadas a um nome), para uso imediato em expressões. Isso usa expressões lambda, descritas na seção *Lambdas*. Observe que a expressão lambda é meramente uma abreviação para uma definição de função simplificada; uma função definida em uma instrução "`def`" pode ser passada adiante ou atribuída a outro nome, assim como uma função definida por uma expressão lambda. O formato "`def`" é, na verdade, mais poderoso, pois permite a execução de várias instruções e anotações.

Nota do programador: Funções são objetos de primeira classe. Uma instrução "`def`" executada dentro de uma definição de função define uma função local que pode ser retornada ou passada adiante. Variáveis livres usadas na função aninhada podem acessar as variáveis locais da função que contém o "def". Veja a seção *Nomeação e ligação* para detalhes.

Ver também

PEP 3107 - Anotações de função
A especificação original para anotações de funções.

PEP 484 - Dicas de tipo
Definição de um significado padrão para anotações: dicas de tipo.

PEP 526 - Sintaxe para anotações de variáveis
Capacidade de fornecer dica de tipo para declarações de variáveis, incluindo variáveis de classe e variáveis de instância.

PEP 563 - Avaliação postergada de anotações
Suporte para referências futuras dentro de anotações, preservando anotações em um formato de string em tempo de execução em vez de avaliação antecipada.

PEP 318 - Decoradores para funções e métodos
Decoradores de função e método foram introduzidos. Decoradores de classe foram introduzidos na **PEP 3129**.

8.8 Definições de classe

Uma definição de classe define um objeto classe (veja a seção *A hierarquia de tipos padrão*):

```
classdef    ::= [decorators] "class" classname [type_params] [inheritance] ":" suite
inheritance ::= "(" [argument_list] ")"
classname   ::= identifier
```

Uma definição de classe é uma instrução executável. A lista de herança geralmente fornece uma lista de classes base (veja *Metaclasses* para usos mais avançados), então cada item na lista deve ser executada como um objeto classe que permite extensão via subclasse. Classes sem uma lista de herança herdam, por padrão, da classe base `object`; portanto,

```
class Foo:
    pass
```

equivale a

```
class Foo(object):
    pass
```

O conjunto da classe é então executado em um novo quadro de execução (veja *Nomeação e ligação*), usando um espaço de nomes local recém-criado e o espaço de nomes global original. (Normalmente, o conjunto contém principalmente definições de função.) Quando o conjunto da classe termina a execução, seu quadro de execução é descartado, mas seu espaço de nomes local é salvo.[5] Um objeto classe é então criado usando a lista de herança para as classes base e o espaço de nomes local salvo para o dicionário de atributos. O nome da classe é vinculado a este objeto classe no espaço de nomes local original.

A ordem em que os atributos são definidos no corpo da classe é preservada no `__dict__` da nova classe. Observe que isso é confiável somente logo após a classe ser criada e somente para classes que foram definidas usando a sintaxe de definição.

A criação de classes pode ser bastante personalizada usando *metaclasses*.

As classes também podem ser decoradas: assim como na decoração de funções,

```
@f1(arg)
@f2
class Foo: pass
```

é aproximadamente equivalente a

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

[5] Um literal de string que aparece como a primeira instrução no corpo da classe é transformado no item `__doc__` do espaço de nomes e, portanto, no *docstring* da classe.

As regras de execução para as expressões de decorador são as mesmas que para decoradores de função. O resultado é então vinculado ao nome da classe.

Alterado na versão 3.9: Classes podem ser decoradas com qualquer *assignment_expression* válida. Anteriormente, a gramática era muito mais restritiva; veja **PEP 614** para detalhes.

Uma lista de *parâmetros de tipo* pode ser dada entre colchetes imediatamente após o nome da classe. Isso indica aos verificadores de tipo estático que a classe é genérica. Em tempo de execução, os parâmetros de tipo podem ser recuperados do atributo `__type_params__` da classe. Veja *Classes genéricas* para mais.

Alterado na versão 3.12: Listas de parâmetros de tipo são novas no Python 3.12.

Nota do programador: Variáveis definidas na definição de classe são atributos de classe; elas são compartilhadas por instâncias. Atributos de instância podem ser definidos em um método com `self.nome = valor`. Atributos de classe e instância são acessíveis por meio da notação "`self.nome`", e um atributo de instância oculta um atributo de classe com o mesmo nome quando acessado dessa forma. Atributos de classe podem ser usados como padrões para atributos de instância, mas usar valores mutáveis pode levar a resultados inesperados. *Descritores* podem ser usados para criar variáveis de instância com diferentes detalhes de implementação.

Ver também

PEP 3115 - Metaclasses no Python 3000
A proposta que alterou a declaração de metaclasses para a sintaxe atual e a semântica de como as classes com metaclasses são construídas.

PEP 3129 - Decoradores de classe
A proposta que adicionou decoradores de classe. Decoradores de função e método foram introduzidos na **PEP 318**.

8.9 Corrotinas

Adicionado na versão 3.5.

8.9.1 Definição de função de corrotina

```
async_funcdef ::= [decorators] "async" "def" funcname "(" [parameter_list] ")"
                  ["->" expression] ":" suite
```

A execução de corrotinas do Python pode ser suspensa e retomada em muitos pontos (consulte *coroutine*). As expressões *await*, *async for* e *async with* só podem ser usadas no corpo de uma função de corrotina.

Funções definidas com a sintaxe `async def` são sempre funções de corrotina, mesmo que não contenham palavras reservadas `await` ou `async`.

Ocorre uma `SyntaxError` se usada uma expressão `yield from` dentro do corpo de uma função de corrotina.

Um exemplo de uma função de corrotina:

```
async def func(param1, param2):
    faz_algo()
    await alguma_corrotina()
```

Alterado na versão 3.7: `await` e `async` agora são palavras reservadas; anteriormente, elas só eram tratadas como tal dentro do corpo de uma função de corrotina.

8.9.2 A instrução `async for`

```
async_for_stmt ::= "async" for_stmt
```

Um *iterável assíncrono* fornece um método `__aiter__` que retorna diretamente um *iterador assíncrono*, que pode chamar código assíncrono em seu método `__anext__`.

A instrução `async for` permite iteração conveniente sobre iteráveis assíncronos.

O seguinte código:

```
async for TARGET in ITER:
    SUITE
else:
    SUITE2
```

É semanticamente equivalente a:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True

while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        SUITE
else:
    SUITE2
```

Veja também *__aiter__()* e *__anext__()* para detalhes.

Ocorre uma `SyntaxError` se usada uma instrução `async for` fora do corpo de uma função de corrotina.

8.9.3 A instrução `async with`

```
async_with_stmt ::= "async" with_stmt
```

Um *gerenciador de contexto assíncrono* é um *gerenciador de contexto* que é capaz de suspender a execução em seus métodos *enter* e *exit*.

O seguinte código:

```
async with EXPRESSÃO as ALVO:
    COMANDOS
```

é semanticamente equivalente a:

```
manager = (EXPRESSÃO)
aenter = type(manager).__aenter__
aexit = type(manager).__aexit__
value = await aenter(manager)
hit_except = False

try:
    ALVO = value
    COMANDOS
except:
    hit_except = True
    if not await aexit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        await aexit(manager, None, None, None)
```

Veja também *__aenter__()* e *__aexit__()* para detalhes.

Ocorre uma `SyntaxError` se usada uma instrução `async with` fora do corpo de uma função de corrotina.

Ver também

PEP 492 - Corrotina com sintaxe de async e wait
A proposta que tornou as corrotinas um conceito autônomo em Python e adicionou sintaxe de suporte.

8.10 Listas de parâmetros de tipo

Adicionado na versão 3.12.

Alterado na versão 3.13: Foi adicionado suporte para valores padrão (veja **PEP 696**).

```
type_params  ::= "[" type_param ("," type_param)* "]"
type_param   ::= typevar | typevartuple | paramspec
typevar      ::= identifier (":" expression)? ("=" expression)?
typevartuple ::= "*" identifier ("=" expression)?
paramspec    ::= "**" identifier ("=" expression)?
```

Funções (incluindo *corrotinas*), *classes* e *apelidos de tipo* podem conter uma lista de parâmetros de tipo:

```
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]
```

Semanticamente, isso indica que a função, classe ou apelido de tipo é genérico sobre uma variável de tipo. Essas informações são usadas principalmente por verificadores de tipo estático e, em tempo de execução, objetos genéricos se comportam muito como suas contrapartes não genéricas.

Parâmetros de tipo são declarados entre colchetes (`[]`) imediatamente após o nome da função, classe ou apelido de tipo. Os parâmetros de tipo são acessíveis dentro do escopo do objeto genérico, mas não em outro lugar. Assim, após uma declaração `def func[T](): pass`, o nome `T` não está disponível no escopo do módulo. Abaixo, a semântica de objetos genéricos é descrita com mais precisão. O escopo de parâmetros de tipo é modelado com uma função especial (tecnicamente, um *escopo de anotação*) que encapsula a criação do objeto genérico.

Funções genéricas, classes e apelidos de tipo têm um atributo `__type_params__` listando seus parâmetros de tipo.

Existem três espécies de parâmetros de tipo:

- `typing.TypeVar`, introduzido por um nome simples (por exemplo, `T`). Semanticamente, isso representa um único tipo para um verificador de tipos.
- `typing.TypeVarTuple`, introduzido por um nome prefixado com um único asterisco (por exemplo, `*Ts`). Semanticamente, isso representa uma tupla de qualquer número de tipos.
- `typing.ParamSpec`, introduzido por um nome prefixado com dois asteriscos (por exemplo, `**P`). Semanticamente, isso representa os parâmetros de um chamável.

As declarações de `typing.TypeVar` podem definir delimitações (*bounds*) e restrições (*constraints*) com dois pontos (`:`) seguidos por uma expressão. Uma única expressão após os dois pontos indica uma delimitação (por exemplo, `T: int`). Semanticamente, isso significa que o `typing.TypeVar` pode representar apenas tipos que são um subtipo desse limite. Uma tupla de expressões entre parênteses após os dois pontos indica um conjunto de restrições (por exemplo, `T: (str, bytes)`). Cada membro da tupla deve ser um tipo (novamente, isso não é imposto em tempo de execução). Variáveis de tipo restrito podem assumir apenas um dos tipos na lista de restrições.

Para `typing.TypeVar`s declarados usando a sintaxe de lista de parâmetros de tipo, a delimitação e as restrições não são avaliados quando o objeto genérico é criado, mas somente quando o valor é acessado explicitamente por meio dos atributos `__bound__` e `__constraints__`. Para fazer isso, as delimitações ou restrições são avaliados em um *escopo de anotação* separado.

`typing.TypeVarTuple`s e `typing.ParamSpec`s não pode ter delimitações e restrições.

Todas as três espécies de parâmetros de tipo também podem ter um *valor padrão*, que é usado quando o parâmetro de tipo não é fornecido explicitamente. Isso é adicionado anexando um único sinal de igual (=) seguido por uma expressão. Como os limites e restrições de tipos variáveis, o valor padrão não é avaliado quando o objeto é criado, mas apenas quando o atributo `__default__` do parâmetro de tipo é acessado. Para esse fim, o valor padrão é avaliado em um *escopo de anotação* separado. Se nenhum valor padrão for especificado para um parâmetro de tipo, o atributo `__default__` é definido como o objeto sinalizador especial `typing.NoDefault`.

O exemplo a seguir indica o conjunto completo de declarações de parâmetros de tipo permitidas:

```
def overly_generic[
   SimpleTypeVar,
   TypeVarWithDefault = int,
   TypeVarWithBound: int,
   TypeVarWithConstraints: (str, bytes),
   *SimpleTypeVarTuple = (int, float),
   **SimpleParamSpec = (str, bytearray),
](
   a: SimpleTypeVar,
   b: TypeVarWithDefault,
   c: TypeVarWithBound,
   d: Callable[SimpleParamSpec, TypeVarWithConstraints],
   *e: SimpleTypeVarTuple,
): ...
```

8.10.1 Funções genéricas

Funções genéricas são declaradas assim:

```
def func[T](arg: T): ...
```

Essa sintaxe é equivalente a:

```
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()
```

Aqui, `annotation-def` indica um *escopo de anotação*, que não é vinculado de verdade a nenhum nome em tempo de execução. (Uma outra liberdade foi tomada na tradução: essa sintaxe não faz ocorrer um acesso de atributo no módulo `typing`; ao invés disso é criada uma instância de `typing.TypeVar` diretamente.)

As anotações das funções genéricas são lidas dentro do escopo de anotação usado para declarar os parâmetros de tipo, mas os valores padrão dos argumentos e decoradores da função não são.

Este exemplo ilustra as regras de escopo para estes casos, bem como para outras espécies de parâmetros de tipo:

```
@decorator
def func[T: int, *Ts, **P](*args: *Ts, arg: Callable[P, T] = some_default):
    ...
```

Tirando a *avaliação preguiçosa* da delimitação da `TypeVar`, isso é equivalente a:

```
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)
    return func
func = decorator(TYPE_PARAMS_OF_func())
```

Os nomes com maiúsculas como `DEFAULT_OF_arg` não são realmente vinculados em tempo de execução.

8.10.2 Classes genéricas

Classes genéricas são declaradas assim:

```
class Bag[T]: ...
```

Essa sintaxe é equivalente a:

```
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()
```

Aqui novamente `annotation-def` (não é uma palavra reservada real) indica um *escopo de anotação*, e o nome `TYPE_PARAMS_OF_Bag` não é realmente vinculado em tempo de execução.

Classes genéricas herdam implicitamente de `typing.Generic`. As classes base e os argumentos nomeados de classes genéricas são avaliados dentro do escopo de tipo para os parâmetros de tipo, e os decoradores são avaliados fora desse escopo. Isso é ilustrado por este exemplo:

```
@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):
```

(continua na próxima página)

(continuação da página anterior)

```
        __type_params__ = (T,)
        ...
    return Bag
Bag = decorator(TYPE_PARAMS_OF_Bag())
```

8.10.3 Apelidos de tipo genérico

A instrução *type* também pode ser usada para criar um apelido de tipo genérico:

```
type ListOrSet[T] = list[T] | set[T]
```

Exceto pela *avaliação preguiçosa* do valor, isso é equivalente a:

```
annotation-def TYPE_PARAMS_OF_ListOrSet():
    T = typing.TypeVar("T")

    annotation-def VALUE_OF_ListOrSet():
        return list[T] | set[T]
    # Na realidade, o valor é avaliado preguiçosamente
    return typing.TypeAliasType("ListOrSet", VALUE_OF_ListOrSet(), type_params=(T,
↪))
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

Aqui, `annotation-def` (não é uma palavra reservada real) indica um *escopo de anotação*. Os nomes em maiúsculas como `TYPE_PARAMS_OF_ListOrSet` não são realmente vinculados em tempo de execução.

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
```

(continua na próxima página)

(continuação da página anterior)

```
#   rule:
# rule_name[return_type]: expression
#   If the return type is omitted, then a void * is returned in C and an Any in
#   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.
# &&e
#   Eager parse e. The parser will not backtrack and will immediately
#   fail with SyntaxError if e cannot be parsed.
#

# 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
```

(continua na próxima página)

(continuação da página anterior)

```
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
    | 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:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
```

(continua na próxima página)

(continuação da página anterior)

```
    | '<<='
    | '>>='
    | '**='
    | '//='

return_stmt:
    | 'return' [star_expressions]

raise_stmt:
    | 'raise' expression ['from' expression ]
    | 'raise'

global_stmt: 'global' ','.NAME+

nonlocal_stmt: 'nonlocal' ','.NAME+

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
# ---------------
```

(continua na próxima página)

(continuação da página anterior)

```
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:
    | 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]
```

(continua na próxima página)

(continuação da página anterior)

```
    | '*' ',' 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:
    | 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
```

(continua na próxima página)

(continuação da página anterior)

```
# --------------

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+ ','? ')' ':' [TYPE_COMMENT] 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
# ----------------

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
```

(continua na próxima página)

(continuação da página anterior)

```
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 !('+' | '-')
    | 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
```

(continua na próxima página)

(continuação da página anterior)

```
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:
    | '*' 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
```

(continua na próxima página)

(continuação da página anterior)

```
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:
    | invalid_type_params
    | '[' type_param_seq ']'

type_param_seq: ','.type_param+ [',']

type_param:
    | NAME [type_param_bound] [type_param_default]
    | '*' NAME [type_param_starred_default]
    | '**' NAME [type_param_default]

type_param_bound: ':' expression
type_param_default: '=' expression
type_param_starred_default: '=' star_expression

# EXPRESSIONS
# -----------

expressions:
    | expression (',' expression )+ [',']
    | expression ','
    | expression

expression:
    | disjunction 'if' disjunction 'else' expression
    | disjunction
    | lambdef

yield_expr:
```

(continua na próxima página)

(continuação da página anterior)

```
    | '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
    | isnot_bitwise_or
    | is_bitwise_or
```

(continua na próxima página)

(continuação da página anterior)

```
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
```

(continua na próxima página)

(continuação da página anterior)

```
    | await_primary

# 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:
```

(continua na próxima página)

(continuação da página anterior)

```
    | 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

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:
    | '{' annotated_rhs '='? [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:
```

(continua na próxima página)

(continuação da página anterior)

```
    | '[' [star_named_expressions] ']'

tuple:
    | '(' [star_named_expression ',' [star_named_expressions]  ] ')'

set: '{' star_named_expressions '}'

# Dicts
# -----

dict:
    | '{' [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+
```

(continua na próxima página)

(continuação da página anterior)

```
    | ','.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
# ==================

# 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:
```

(continua na próxima página)

(continuação da página anterior)

```
    | 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 ')'
    | '(' [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 console *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`.

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*). ABCs 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 ABCs 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 ABCs 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 (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:

```
chamavel(argumento1, argumento2, argumentoN)
```

Uma *função*, e por extensão um *método*, é um chamável. Uma instância de uma classe que implementa o método *__call__()* também é um chamável.

função de retorno
Também conhecida como callback, é uma função sub-rotina que é passada como um argumento a ser executado em algum ponto no futuro.

classe
Um modelo para criação de objetos definidos pelo usuário. Definições de classe normalmente contém definições de métodos que operam sobre instâncias da classe.

variável de classe
Uma variável definida em uma classe e destinada a ser modificada apenas no nível da classe (ou seja, não em uma instância da classe).

variável de clausura
Uma *variável livre* referenciada de um *escopo aninhado* que é definida em um escopo externo em vez de ser resolvida em tempo de execução a partir dos espaços de nomes embutido ou globais. Pode ser explicitamente definida com a palavra reservada `nonlocal` para permitir acesso de gravação, ou implicitamente definida se a variável estiver sendo somente lida.

Por exemplo, na função `interna` no código a seguir, tanto `x` quanto `print` são *variáveis livres*, mas somente `x` é uma *variável de clausura*:

```
def externa():
    x = 0
    def interna():
        nonlocal x
        x += 1
        print(x)
    return interna
```

Devido ao atributo `codeobject.co_freevars` (que, apesar do nome, inclui apenas os nomes das variáveis de clausura em vez de listar todas as variáveis livres referenciadas), o termo mais geral *variável livre* às vezes é usado mesmo quando o significado pretendido é se referir especificamente às variáveis de clausura.

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.

contexto
Este termo tem diferentes significados dependendo de onde e como ele é usado. Alguns significados comuns:

- O estado ou ambiente temporário estabelecido por um *gerenciador de contexto* por meio de uma instrução `with`.
- A coleção de ligações de chave-valor associadas a um objeto `contextvars.Context` específico e acessadas por meio de objetos `ContextVar`. Veja também *variável de contexto*.
- Um objeto `contextvars.Context`. Veja também *contexto atual*.

protocolo de gerenciamento de contexto
Os métodos `__enter__()` e `__exit__()` chamados pela instrução `with`. Veja **PEP 343**.

gerenciador de contexto
Um objeto que implementa o *protocolo de gerenciamento de contexto* e controla o ambiente visto em uma instrução `with`. Veja **PEP 343**.

variável de contexto
Uma variável cujo valor depende de qual contexto é o *contexto atual*. Os valores são acessados por meio de objetos `contextvars.ContextVar`. Variáveis de contexto são usadas principalmente para isolar o estado entre tarefas assíncronas simultâneas.

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.

contexto atual
O *contexto* (objeto `contextvars.Context`) que é usado atualmente pelos objetos `ContextVar` para acessar (obter ou definir) os valores de *variáveis de contexto*. Cada thread tem seu próprio contexto atual. Frameworks para executar tarefas assíncronas (veja `asyncio`) associam cada tarefa a um contexto que se torna o contexto atual sempre que a tarefa inicia ou retoma a execução.

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
Abreviatura de "documentation string" (string de documentação). Uma string literal que aparece como pri-

meira expressão numa classe, função ou módulo. Ainda que sejam ignoradas quando a suíte é executada, é reconhecida pelo compilador que a coloca no atributo `__doc__` da classe, função ou módulo que a encapsula. Como ficam disponíveis por meio de introspecção, docstrings são o lugar canônico para documentação do objeto.

tipagem pato
Também conhecida como *duck-typing*, é um estilo de programação que não verifica o tipo do objeto para determinar se ele possui a interface correta; em vez disso, o método ou atributo é simplesmente chamado ou utilizado ("Se se parece com um pato e grasna como um pato, então deve ser um pato.") Enfatizando interfaces ao invés de tipos específicos, o código bem desenvolvido aprimora sua flexibilidade por permitir substituição polimórfica. Tipagem pato evita necessidade de testes que usem `type()` ou `isinstance()`. (Note, porém, que a tipagem pato pode ser complementada com o uso de *classes base abstratas*.) Ao invés disso, são normalmente empregados testes `hasattr()` ou programação *EAFP*.

EAFP
Iniciais da expressão em inglês "easier to ask for forgiveness than permission" que significa "é mais fácil pedir perdão que permissão". Este estilo de codificação comum no Python presume a existência de chaves ou atributos válidos e captura exceções caso essa premissa se prove falsa. Este estilo limpo e rápido se caracteriza pela presença de várias instruções `try` e `except`. A técnica diverge do estilo *LBYL*, comum em outras linguagens como C, por exemplo.

expressão
Uma parte da sintaxe que pode ser avaliada para algum valor. Em outras palavras, uma expressão é a acumulação de elementos de expressão como literais, nomes, atributos de acesso, operadores ou chamadas de funções, todos os quais retornam um valor. Em contraste com muitas outras linguagens, nem todas as construções de linguagem são expressões. Também existem *instruções*, as quais não podem ser usadas como expressões, como, por exemplo, `while`. Atribuições também são instruções, não expressões.

módulo de extensão
Um módulo escrito em C ou C++, usando a API C do Python para interagir tanto com código de usuário quanto do núcleo.

f-string
Literais string prefixadas com `'f'` ou `'F'` são conhecidas como "f-strings" que é uma abreviação de *formatted string literals*. Veja também **PEP 498**.

objeto arquivo
Um objeto que expõe uma API orientada a arquivos (com métodos tais como `read()` ou `write()`) para um recurso subjacente. Dependendo da maneira como foi criado, um objeto arquivo pode mediar o acesso a um arquivo real no disco ou outro tipo de dispositivo de armazenamento ou de comunicação (por exemplo a entrada/saída padrão, buffers em memória, soquetes, pipes, etc.). Objetos arquivo também são chamados de *objetos arquivo ou similares* ou *fluxos*.

Atualmente há três categorias de objetos arquivo: *arquivos binários* brutos, *arquivos binários* em buffer e *arquivos textos*. Suas interfaces estão definidas no módulo `io`. A forma canônica para criar um objeto arquivo é usando a função `open()`.

objeto arquivo ou similar
Um sinônimo do termo *objeto arquivo*.

tratador de erros e codificação do sistema de arquivos
Tratador de erros e codificação usado pelo Python para decodificar bytes do sistema operacional e codificar Unicode para o sistema operacional.

A codificação do sistema de arquivos deve garantir a decodificação bem-sucedida de todos os bytes abaixo de 128. Se a codificação do sistema de arquivos falhar em fornecer essa garantia, as funções da API podem levantar `UnicodeError`.

As funções `sys.getfilesystemencoding()` e `sys.getfilesystemencodeerrors()` podem ser usadas para obter o tratador de erros e codificação do sistema de arquivos.

O *tratador de erros e codificação do sistema de arquivos* são configurados na inicialização do Python pela função `PyConfig_Read()`: veja os membros `filesystem_encoding` e `filesystem_errors` do `PyConfig`.

Veja também *codificação da localidade*.

localizador
Um objeto que tenta encontrar o *carregador* para um módulo que está sendo importado.

Existem dois tipos de localizador: *localizadores de metacaminho* para uso com `sys.meta_path`, e *localizadores de entrada de caminho* para uso com `sys.path_hooks`.

Veja *Localizadores e carregadores* e `importlib` para muito mais detalhes.

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**.

threads livres
Um modelo de threads onde múltiplas threads podem simultaneamente executar bytecode Python no mesmo interpretador. Isso está em contraste com a *trava global do interpretador* que permite apenas uma thread por vez executar bytecode Python. Veja **PEP 703**.

variável livre
Formalmente, conforme definido no *modelo de execução de linguagem*, uma variável livre é qualquer variável usada em um espaço de nomes que não seja uma variável local naquele espaço de nomes. Veja *variável de clausura* para um exemplo. Pragmaticamente, devido ao nome do atributo `codeobject.co_freevars`, o termo também é usado algumas vezes como sinônimo de *variável de clausura*.

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 soma_dois_numeros(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 (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 laço, um intervalo, e uma cláusula `if` opcional. A expressão combinada gera valores para uma função encapsuladora:

```
>>> sum(i*i for i in range(10))         # soma dos quadrados 0, 1, 4, ... 81
285
```

função genérica
Uma função composta por várias funções implementando a mesma operação para diferentes tipos. Qual implementação deverá ser usada durante a execução é determinada pelo algoritmo de despacho.

Veja também a entrada *despacho único* no glossário, o decorador `functools.singledispatch()`, e a **PEP 443**.

tipo genérico
Um *tipo* que pode ser parametrizado; tipicamente uma *classe contêiner* tal como `list` ou `dict`. Usado para *dicas de tipo* e *anotações*.

Para mais detalhes, veja tipo apelido genérico, **PEP 483**, **PEP 484**, **PEP 585**, e o módulo `typing`.

GIL
Veja *trava global do interpretador*.

trava global do interpretador
O mecanismo utilizado pelo interpretador *CPython* para garantir que apenas uma thread execute o *bytecode* Python por vez. Isto simplifica a implementação do CPython ao fazer com que o modelo de objetos (incluindo tipos embutidos críticos como o `dict`) ganhem segurança implícita contra acesso concorrente. Travar todo o interpretador facilita que o interpretador em si seja multitarefa, às custas de muito do paralelismo já provido por máquinas multiprocessador.

No entanto, alguns módulos de extensão, tanto da biblioteca padrão quanto de terceiros, são desenvolvidos de forma a liberar a GIL ao realizar tarefas computacionalmente muito intensas, como compactação ou cálculos de hash. Além disso, a GIL é sempre liberado nas operações de E/S.

A partir de Python 3.13, o GIL pode ser desabilitado usando a configuração de construção `--disable-gil`. Depois de construir Python com essa opção, o código deve ser executado com a opção `-X gil=0` ou a variável de ambiente `PYTHON_GIL=0` deve estar definida. Esse recurso provê um desempenho melhor para aplicações com múltiplas threads e torna mais fácil o uso eficiente de CPUs com múltiplos núcleos. Para mais detalhes, veja **PEP 703**.

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 *Invalidação de bytecode em cache*.

hasheável
Um objeto é *hasheável* se tem um valor de hash que nunca muda durante seu ciclo de vida (precisa ter um método `__hash__()`) e pode ser comparado com outros objetos (precisa ter um método `__eq__()`). Objetos hasheáveis que são comparados como iguais devem ter o mesmo valor de hash.

A hasheabilidade faz com que um objeto possa ser usado como uma chave de dicionário e como um membro de conjunto, pois estas estruturas de dados utilizam os valores de hash internamente.

A maioria dos objetos embutidos imutáveis do Python são hasheáveis; containers mutáveis (tais como listas ou dicionários) não são; containers imutáveis (tais como tuplas e frozensets) são hasheáveis apenas se os seus

elementos são hasheáveis. Objetos que são instâncias de classes definidas pelo usuário são hasheáveis por padrão. Todos eles comparam de forma desigual (exceto entre si mesmos), e o seu valor hash é derivado a partir do seu `id()`.

IDLE
Um ambiente de desenvolvimento e aprendizado integrado para Python. idle é um editor básico e um ambiente interpretador que vem junto com a distribuição padrão do Python.

imortal
Objetos imortais são um detalhe da implementação do CPython introduzida na **PEP 683**.

Se um objeto é imortal, sua *contagem de referências* nunca é modificada e, portanto, nunca é desalocado enquanto o interpretador está em execução. Por exemplo, `True` e `None` são imortais no CPython.

imutável
Um objeto que possui um valor fixo. Objetos imutáveis incluem números, strings e tuplas. Estes objetos não podem ser alterados. Um novo objeto deve ser criado se um valor diferente tiver de ser armazenado. Objetos imutáveis têm um papel importante em lugares onde um valor constante de hash seja necessário, como por exemplo uma chave em um dicionário.

caminho de importação
Uma lista de localizações (ou *entradas de caminho*) que são buscadas pelo *localizador baseado no caminho* por módulos para importar. Durante a importação, esta lista de localizações usualmente vem a partir de `sys.path`, mas para subpacotes ela também pode vir do atributo `__path__` de pacotes-pai.

importação
O processo pelo qual o código Python em um módulo é disponibilizado para o código Python em outro módulo.

importador
Um objeto que localiza e carrega um módulo; Tanto um *localizador* e o objeto *carregador*.

interativo
Python tem um interpretador interativo, o que significa que você pode digitar instruções e expressões no prompt do interpretador, executá-los imediatamente e ver seus resultados. Apenas execute `python` sem argumentos (possivelmente selecionando-o a partir do menu de aplicações de seu sistema operacional). O interpretador interativo é uma maneira poderosa de testar novas ideias ou aprender mais sobre módulos e pacotes (lembre-se do comando `help(x)`). Para saber mais sobre modo interativo, veja tut-interac.

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.

O CPython não aplica consistentemente o requisito de que um iterador defina `__iter__()`. E também observe que o CPython com threads livres não garante a segurança do thread das operações do iterador.

função chave
Uma função chave ou função colação é um chamável que retorna um valor usado para ordenação ou classificação. Por exemplo, `locale.strxfrm()` é usada para produzir uma chave de ordenação que leva o locale em consideração para fins de ordenação.

Uma porção de ferramentas no Python aceitam funções chave para controlar como os elementos são ordenados ou agrupados. Algumas delas incluem `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` e `itertools.groupby()`.

Há várias maneiras de se criar funções chave. Por exemplo, o método `str.lower()` pode servir como uma função chave para ordenações insensíveis à caixa. Alternativamente, uma função chave ad-hoc pode ser construída a partir de uma expressão *lambda*, como `lambda r: (r[0], r[2])`. Além disso, `operator.attrgetter()`, `operator.itemgetter()` e `operator.methodcaller()` são três construtores de função chave. Consulte o guia de Ordenação para ver exemplos de como criar e utilizar funções chave.

argumento nomeado
Veja *argumento*.

lambda
Uma função de linha anônima consistindo de uma única *expressão*, que é avaliada quando a função é chamada. A sintaxe para criar uma função lambda é `lambda [parameters]: expression`

LBYL
Iniciais da expressão em inglês "look before you leap", que significa algo como "olhe antes de pisar". Este estilo de codificação testa as pré-condições explicitamente antes de fazer chamadas ou buscas. Este estilo contrasta com a abordagem *EAFP* e é caracterizada pela presença de muitas instruções *if*.

Em um ambiente multithread, a abordagem LBYL pode arriscar a introdução de uma condição de corrida entre "o olhar" e "o pisar". Por exemplo, o código `if key in mapping: return mapping[key]` pode falhar se outra thread remover *key* do *mapping* após o teste, mas antes da olhada. Esse problema pode ser resolvido com travas ou usando a abordagem EAFP.

analisador léxico
Nome formal para o *tokenizador*; veja *token*.

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. Ele deve definir os métodos `exec_module()` e `create_module()` para implementar a interface `Loader`. Um carregador é normalmente retornado por um *localizador*. Veja também:

- *Localizadores e carregadores*
- `importlib.abc.Loader`
- **PEP 302**

codificação da localidade
No Unix, é a codificação da localidade do LC_CTYPE, que pode ser definida com `locale.setlocale(locale.LC_CTYPE, new_locale)`.

No Windows, é a página de código ANSI (ex: `"cp1252"`).

No Android e no VxWorks, o Python usa `"utf-8"` como a codificação da localidade.

`locale.getencoding()` pode ser usado para obter a codificação da localidade.

Veja também *tratador de erros e codificação do sistema de arquivos*.

método mágico
Um sinônimo informal para um *método especial*.

mapeamento
Um objeto contêiner que tem suporte a pesquisas de chave arbitrária e implementa os métodos especificados nas `collections.abc.Mapping` ou `collections.abc.MutableMapping` classes base abstratas. Exemplos incluem `dict`, `collections.defaultdict`, `collections.OrderedDict` e `collections.Counter`.

localizador de metacaminho
Um *localizador* retornado por uma busca de `sys.meta_path`. Localizadores de metacaminho são relacionados a, mas diferentes de, *localizadores de entrada de caminho*.

Veja `importlib.abc.MetaPathFinder` para os métodos que localizadores de metacaminho implementam.

metaclasse
A classe de uma classe. Definições de classe criam um nome de classe, um dicionário de classe e uma lista de classes base. A metaclasse é responsável por receber estes três argumentos e criar a classe. A maioria das linguagens de programação orientadas a objetos provê uma implementação default. O que torna o Python especial é o fato de ser possível criar metaclasses personalizadas. A maioria dos usuários nunca vai precisar deste recurso, mas quando houver necessidade, metaclasses possibilitam soluções poderosas e elegantes. Metaclasses têm sido utilizadas para gerar registros de acesso a atributos, para incluir proteção contra acesso concorrente, rastrear a criação de objetos, implementar singletons, dentre muitas outras tarefas.

Mais informações podem ser encontradas em *Metaclasses*.

método
Uma função que é definida dentro do corpo de uma classe. Se chamada como um atributo de uma instância daquela classe, o método receberá a instância do objeto como seu primeiro *argumento* (que comumente é chamado de `self`). Veja *função* e *escopo aninhado*.

ordem de resolução de métodos
Ordem de resolução de métodos é a ordem em que os membros de uma classe base são buscados durante a pesquisa. Veja python_2.3_mro para detalhes do algoritmo usado pelo interpretador do Python desde a versão 2.3.

módulo
Um objeto que serve como uma unidade organizacional de código Python. Os módulos têm um espaço de nomes contendo objetos Python arbitrários. Os módulos são carregados pelo Python através do processo de *importação*.

Veja também *pacote*.

spec de módulo
Um espaço de nomes que contém as informações relacionadas à importação usadas para carregar um módulo. Uma instância de `importlib.machinery.ModuleSpec`.

Veja também *Especificações de módulo*.

MRO
Veja *ordem de resolução de métodos*.

mutável
Objeto mutável é aquele que pode modificar seus valor mas manter seu `id()`. Veja também *imutável*.

tupla nomeada
O termo "tupla nomeada" é aplicado a qualquer tipo ou classe que herda de tupla e cujos elementos indexáveis também são acessíveis usando atributos nomeados. O tipo ou classe pode ter outras funcionalidades também.

Diversos tipos embutidos são tuplas nomeadas, incluindo os valores retornados por `time.localtime()` e `os.stat()`. Outro exemplo é `sys.float_info`:

```
>>> sys.float_info[1]                    # acesso indexado
1024
>>> sys.float_info.max_exp               # acesso a campo nomeado
1024
>>> isinstance(sys.float_info, tuple)    # tipo de tupla
True
```

Algumas tuplas nomeadas são tipos embutidos (tal como os exemplos acima). Alternativamente, uma tupla nomeada pode ser criada a partir de uma definição de classe regular, que herde de `tuple` e que defina campos nomeados. Tal classe pode ser escrita a mão, ou ela pode ser criada herdando `typing.NamedTuple` ou com uma função fábrica `collections.namedtuple()`. As duas últimas técnicas também adicionam alguns métodos extras, que podem não ser encontrados quando foi escrita manualmente, ou em tuplas nomeadas embutidas.

espaço de nomes
O lugar em que uma variável é armazenada. Espaços de nomes são implementados como dicionários. Existem os espaços de nomes local, global e nativo, bem como espaços de nomes aninhados em objetos (em métodos). Espaços de nomes suportam modularidade ao prevenir conflitos de nomes. Por exemplo, as funções `__builtin__.open()` e `os.open()` são diferenciadas por seus espaços de nomes. Espaços de nomes também auxiliam na legibilidade e na manutenibilidade ao torar mais claro quais módulos implementam uma função. Escrever `random.seed()` ou `itertools.izip()`, por exemplo, deixa claro que estas funções são implementadas pelos módulos `random` e `itertools` respectivamente.

pacote de espaço de nomes
Um *pacote* que serve apenas como contêiner para subpacotes. 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`.

Pacotes de espaço de nomes permitem que vários pacotes instaláveis individualmente tenham um pacote pai comum. Caso contrário, é recomendado usar um *pacote regular*.

Para mais informações, veja **PEP 420** e *Pacotes de espaço de nomes*.

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*.

escopo otimizado
Um escopo no qual os nomes das variáveis locais de destino são conhecidos de forma confiável pelo compilador quando o código é compilado, permitindo a otimização do acesso de leitura e gravação a esses nomes. Os espaços de nomes locais para funções, geradores, corrotinas, compreensões e expressões geradoras são otimizados desta forma. Nota: a maioria das otimizações de interpretador são aplicadas a todos os escopos, apenas aquelas que dependem de um conjunto conhecido de nomes de variáveis locais e não locais são restritas a escopos otimizados.

pacote
Um *módulo* Python é capaz de conter submódulos ou recursivamente, subpacotes. Tecnicamente, um pacote é um módulo Python com um atributo `__path__`.

Veja também *pacote regular* e *pacote de espaço de nomes*.

parâmetro
Uma entidade nomeada na definição de uma *função* (ou método) que específica um *argumento* (ou em alguns casos, argumentos) que a função pode receber. Existem cinco tipos de parâmetros:

- *posicional-ou-nomeado*: especifica um argumento que pode ser tanto *posicional* quanto *nomeado*. Esse é o tipo padrão de parâmetro, por exemplo *foo* e *bar* a seguir:

```
def func(foo, bar=None): ...
```

- *somente-posicional*: especifica um argumento que pode ser fornecido apenas por posição. Parâmetros somente-posicionais podem ser definidos incluindo o caractere `/` na lista de parâmetros da definição da função após eles, por exemplo *somentepos1* e *somentepos2* a seguir:

```
def func(somentepos1, somentepos2, /, posicional_ou_nomeado): ...
```

- *somente-nomeado*: especifica um argumento que pode ser passado para a função somente por nome. Parâmetros somente-nomeados podem ser definidos com um simples parâmetro var-posicional ou um `*` antes deles na lista de parâmetros na definição da função, por exemplo *somente_nom1* and *somente_nom2* a seguir:

```
def func(arg, *, somente_nom1, somente_nom2): ...
```

- *var-posicional*: especifica que uma sequência arbitrária de argumentos posicionais pode ser fornecida (em adição a qualquer argumento posicional já aceito por outros parâmetros). Tal parâmetro pode ser definido colocando um `*` antes do nome do parâmetro, por exemplo *args* a seguir:

```
def func(*args, **kwargs): ...
```

- *var-nomeado*: especifica que, arbitrariamente, muitos argumentos nomeados podem ser fornecidos (em adição a qualquer argumento nomeado já aceito por outros parâmetros). Tal parâmetro pode definido colocando-se `**` antes do nome, por exemplo *kwargs* no exemplo acima.

Parâmetros podem especificar tanto argumentos opcionais quanto obrigatórios, assim como valores padrão para alguns argumentos opcionais.

Veja também o termo *argumento* no glossário, a pergunta do FAQ 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* padrão que procura por um *caminho de importação* de módulos.

objeto caminho ou similar
Um objeto representando um caminho de sistema de arquivos. Um objeto caminho ou similar é ou um objeto `str` ou `bytes` representando um caminho, ou um objeto implementando o protocolo `os.PathLike`. Um objeto que suporta o protocolo `os.PathLike` pode ser convertido para um arquivo de caminho do sistema `str` ou `bytes`, através da chamada da função `os.fspath()`; `os.fsdecode()` e `os.fsencode()` podem ser usadas para garantir um `str` ou `bytes` como resultado, respectivamente. Introduzido na **PEP 519**.

PEP
Proposta de melhoria do Python. Uma PEP é um documento de design que fornece informação para a comunidade Python, ou descreve uma nova funcionalidade para o Python ou seus predecessores ou ambientes. PEPs devem prover uma especificação técnica concisa e um racional para funcionalidades propostas.

PEPs têm a intenção de ser os mecanismos primários para propor novas funcionalidades significativas, para coletar opiniões da comunidade sobre um problema, e para documentar as decisões de design que foram adicionadas ao Python. O autor da PEP é responsável por construir um consenso dentro da comunidade e documentar opiniões dissidentes.

Veja **PEP 1**.

porção
Um conjunto de arquivos em um único diretório (possivelmente armazenado em um arquivo zip) que contribuem para um pacote de espaço de nomes, conforme definido em **PEP 420**.

argumento posicional
Veja *argumento*.

API provisória
Uma API provisória é uma API que foi deliberadamente excluída das bibliotecas padrões com compatibilidade retroativa garantida. Enquanto mudanças maiores para tais interfaces não são esperadas, contanto que elas sejam marcadas como provisórias, mudanças retroativas incompatíveis (até e incluindo a remoção da interface) podem ocorrer se consideradas necessárias pelos desenvolvedores principais. Tais mudanças não serão feitas gratuitamente – elas irão ocorrer apenas se sérias falhas fundamentais forem descobertas, que foram esquecidas anteriormente a inclusão da API.

Mesmo para APIs provisórias, mudanças retroativas incompatíveis são vistas como uma "solução em último caso" - cada tentativa ainda será feita para encontrar uma resolução retroativa compatível para quaisquer problemas encontrados.

Esse processo permite que a biblioteca padrão continue a evoluir com o passar do tempo, sem se prender em erros de design problemáticos por períodos de tempo prolongados. Veja **PEP 411** para mais detalhes.

pacote provisório
Veja *API provisória*.

Python 3000
Apelido para a linha de lançamento da versão do Python 3.x (cunhada há muito tempo, quando o lançamento da versão 3 era algo em um futuro muito distante.) Esse termo possui a seguinte abreviação: "Py3k".

Pythônico
Uma ideia ou um pedaço de código que segue de perto as formas de escritas mais comuns da linguagem Python, ao invés de implementar códigos usando conceitos comuns a outras linguagens. Por exemplo, um formato comum em Python é fazer um laço sobre todos os elementos de uma iterável usando a instrução `for`. Muitas outras linguagens não têm esse tipo de construção, então as pessoas que não estão familiarizadas com o Python usam um contador numérico:

```
for i in range(len(comida)):
    print(comida[i])
```

Ao contrário do método mais limpo, Pythônico:

```
for parte in comida:
    print(parte)
```

nome qualificado
Um nome pontilhado (quando 2 termos são ligados por um ponto) que mostra o "path" do escopo global de um módulo para uma classe, função ou método definido num determinado módulo, conforme definido pela **PEP 3155**. Para funções e classes de nível superior, o nome qualificado é o mesmo que o nome do objeto:

```
>>> class C:
...     class D:
...         def metodo(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.metodo.__qualname__
'C.D.metodo'
```

Quando usado para se referir a módulos, o *nome totalmente qualificado* significa todo o caminho pontilhado para o módulo, incluindo quaisquer pacotes pai, por exemplo: `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

contagem de referências
O número de referências a um objeto. Quando a contagem de referências de um objeto cai para zero, ele é desalocado. Alguns objetos são *imortais* e têm contagens de referências que nunca são modificadas e, portanto, os objetos nunca são desalocados. A contagem de referências geralmente não é visível para o código Python, mas é um elemento-chave da implementação do *CPython*. Os programadores podem chamar a função `sys.getrefcount()` para retornar a contagem de referências para um objeto específico.

pacote regular
Um *pacote* tradicional, como um diretório contendo um arquivo `__init__.py`.

Veja também *pacote de espaço de nomes*.

REPL
Um acrônimo para "read–eval–print loop", outro nome para o console *interativo* do interpretador.

__slots__
Uma declaração dentro de uma classe que economiza memória pré-declarando espaço para atributos de instâncias, e eliminando dicionários de instâncias. Apesar de popular, a técnica é um tanto quanto complicada de acertar, e é melhor se for reservada para casos raros, onde existe uma grande quantidade de instâncias em uma aplicação onde a memória é crítica.

sequência
Um *iterável* com suporte para acesso eficiente a seus elementos através de índices inteiros via método especial *__getitem__()* e que define o método *__len__()* que devolve o tamanho da sequência. Alguns tipos de sequência embutidos são: `list`, `str`, `tuple`, e `bytes`. Note que `dict` também tem suporte para *__getitem__()* e `__len__()`, mas é considerado um mapeamento e não uma sequência porque a busca usa uma chave *hasheável* arbitrária em vez de inteiros.

A classe base abstrata `collections.abc.Sequence` define uma interface mais rica que vai além de apenas *__getitem__()* e *__len__()*, adicionando `count()`, `index()`, *__contains__()*, e *__reversed__()*. Tipos que implementam essa interface podem ser explicitamente registrados usando `register()`. Para mais documentação sobre métodos de sequências em geral, veja Operações comuns de sequências.

compreensão de conjunto
Uma maneira compacta de processar todos ou parte dos elementos em iterável e retornar um conjunto com os resultados. `results = {c for c in 'abracadabra' if c not in 'abc'}` gera um conjunto de strings `{'r', 'd'}`. Veja *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.

suavemente descontinuado
Uma API suavemente descontinuada não deve ser usada em código novo, mas é seguro para código já existente usá-la. A API continua documentada e testada, mas não será aprimorada mais.

A descontinuação suave, diferentemente da descontinuação normal, não planeja remover a API e não emitirá avisos.

Veja PEP 387: Descontinuação suave.

método especial
Um método que é chamado implicitamente pelo Python para executar uma certa operação em um tipo, como uma adição por exemplo. Tais métodos tem nomes iniciando e terminando com dois underscores. Métodos especiais estão documentados em *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*.

token
Uma pequena unidade de código-fonte, gerada pelo *analisador léxico* (também chamado de *tokenizador*). Nomes, números, strings, operadores, quebras de linha e similares são representados por tokens.

O módulo `tokenize` expõe o analisador léxico do Python. O módulo `token` contém informações sobre os vários tipos de tokens.

aspas triplas
Uma string que está definida com três ocorrências de aspas duplas (") ou apóstrofos ('). Enquanto elas não fornecem nenhuma funcionalidade não disponível com strings de aspas simples, elas são úteis para inúmeras razões. Elas permitem que você inclua aspas simples e duplas não escapadas dentro de uma string, e elas podem utilizar múltiplas linhas sem o uso de caractere de continuação, fazendo-as especialmente úteis quando escrevemos documentação em docstrings.

tipo
O tipo de um objeto Python determina qual classe de objeto ele é; cada objeto tem um tipo. Um tipo de objeto é acessível pelo atributo *`__class__`* ou pode ser recuperado com `type(obj)`.

apelido de tipo
Um sinônimo para um tipo, criado através da atribuição do tipo para um identificador.

Apelidos de tipo são úteis para simplificar *dicas de tipo*. Por exemplo:

```
def remove_tons_de_cinza(
        cores: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

pode tornar-se mais legível desta forma:

```
Cor = tuple[int, int, int]

def remove_tons_de_cinza(cores: list[Cor]) -> list[Cor]:
    pass
```

Veja `typing` e **PEP 484**, a qual descreve esta funcionalidade.

dica de tipo
Uma *anotação* que especifica o tipo esperado para uma variável, um atributo de classe, ou um parâmetro de função ou um valor de retorno.

Dicas de tipo são opcionais e não são forçadas pelo Python, mas elas são úteis para *verificadores de tipo estático*. Eles também ajudam IDEs a completar e refatorar código.

Dicas de tipos de variáveis globais, atributos de classes, e funções, mas não de variáveis locais, podem ser acessadas usando `typing.get_type_hints()`.

Veja `typing` e **PEP 484**, a qual descreve esta funcionalidade.

novas linhas universais
Uma maneira de interpretar fluxos de textos, na qual todos estes são reconhecidos como caracteres de fim de linha: a convenção para fim de linha no Unix `'\n'`, a convenção no Windows `'\r\n'`, e a antiga convenção no Macintosh `'\r'`. Veja **PEP 278** e **PEP 3116**, bem como `bytes.splitlines()` para uso adicional.

anotação de variável
Uma *anotação* de uma variável ou um atributo de classe.

Ao fazer uma anotação de uma variável ou um atributo de classe, a atribuição é opcional:

```
class C:
    campo: 'anotação'
```

Anotações de variáveis são normalmente usadas para *dicas de tipo*: por exemplo, espera-se que esta variável receba valores do tipo `int`:

```
contagem: int = 0
```

A sintaxe de anotação de variável é explicada na seção *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 esta documentação

A documentação do Python é gerada a partir de fontes reStructuredText usando Sphinx, um gerador de documentação criado originalmente para Python e agora mantido como um projeto independente.

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 autor 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 do 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

Python foi criado no início dos anos 1990 por Guido van Rossum no Stichting Mathematisch Centrum (CWI, veja https://www.cwi.nl) na Holanda como sucessor de uma linguagem chamada ABC. Guido continua sendo o principal autor do Python, embora inclua muitas contribuições de outros.

Em 1995, Guido continuou seu trabalho em Python na Corporation for National Research Initiatives (CNRI, veja https://www.cnri.reston.va.us) em Reston, Virgínia, onde lançou várias versões do software.

Em maio de 2000, Guido e a equipe de desenvolvimento do núcleo Python mudaram-se para BeOpen.com para formar a equipe BeOpen PythonLabs. Em outubro do mesmo ano, a equipe PythonLabs mudou-se para a Digital Creations, que se tornou Zope Corporation. Em 2001, a Python Software Foundation (PSF, veja https://www.python.org/psf/) foi formada, uma organização sem fins lucrativos criada especificamente para possuir Propriedade Intelectual relacionada ao Python. A Zope Corporation era um membro patrocinador da 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? (1)
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	sim (2)
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

(1) 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.

(2) De acordo com Richard Stallman, 1.6.1 não é compatível com GPL, porque sua licença tem uma cláusula de escolha de lei. De acordo com a CNRI, no entanto, o advogado de Stallman disse ao advogado da CNRI que 1.6.1 "não é incompatível" com a GPL.

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 a Python Software Foundation License Versão 2.

A partir do Python 3.8.6, exemplos, receitas e outros códigos na documentação são licenciados duplamente sob o Licença PSF versão 2 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 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") 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 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python 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 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.

4. PSF is making Python 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 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, 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.
```

(continua na próxima página)

(continuação da página anterior)

```
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, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 ACORDO DE LICENCIAMENTO DA BEOPEN.COM PARA PYTHON 2.0

ACORDO DE LICENCIAMENTO DA BEOPEN DE FONTE ABERTA DO PYTHON VERSÃO 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CONTRATO DE LICENÇA DA CNRI PARA O PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.
```

(continua na próxima página)

(continuação da página anterior)

```
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 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
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)
```

(continua na próxima página)

(continuação da página anterior)

```
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.
```

(continua na próxima página)

(continuação da página anterior)

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Serviços de soquete assíncrono

Os módulos `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
```

(continua na próxima página)

(continuação da página anterior)

```
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 codec `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
```

(continua na próxima página)

(continuação da página anterior)

```
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:
```

(continua na próxima página)

(continuação da página anterior)

```
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.
```

(continua na próxima página)

(continuação da página anterior)

```
</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` e `ssl` 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.
```

(continua na próxima página)

(continuação da página anterior)

```
   "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
   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,
```

(continua na próxima página)

(continuação da página anterior)

```
   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
       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
```

(continua na próxima página)

(continuação da página anterior)

```
   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,
```

(continua na próxima página)

(continuação da página anterior)

```
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

A extensão C `_decimal` subjacente ao módulo `decimal` é construída usando uma cópia incluída da biblioteca libmpdec, a menos que a construção esteja configurada com `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 Conjunto de testes C14N do W3C

O conjunto de testes C14N 2.0 no pacote `test` (`Lib/test/xmltestdata/c14n-20/`) foi recuperado do site do W3C em https://www.w3.org/TR/xml-c14n2-testcases/ e é distribuído sob a licença BSD de 3 cláusulas:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
```

(continua na próxima página)

(continuação da página anterior)

```
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
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 mimalloc

Licença MIT:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

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.3.20 asyncio

Partes do módulo `asyncio` são incorporadas do uvloop 0.16, que é distribuído sob a licença MIT:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.21 Global Unbounded Sequences (GUS)

O arquivo `Python/qsbr.c` é adaptado do esquema de recuperação de memória segura "Global Unbounded Sequences" do FreeBSD em subr_smr.c. O arquivo é distribuído sob a licença BSD de 2 cláusulas:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

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 unmodified, 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 "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 BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

APÊNDICE D

Direitos autorais

Veja: *História e Licença* para informações completas de licença e permissões.

Índice

Não alfabético

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

U

V

W

X

Y

Z