Manual de Referência Python

Release 2.4.2

Guido van Rossum
Fred L. Drake, Jr., editor
Tradução: Python Brasil

23 de novembro de 2005

Python Software Foundation
Email: docs@python.org **Python Brasil**
http://pythonbrasil.com.br

Tradução: Pedro Werneck, Osvaldo Santana Neto, José Alexandre Nalon, Felipe Lessa, Pedro de Medeiros, Rafael Almeida, Renata Palazzo, Rodrigo Senra e outros.

Revisão: Pedro Werneck, Osvaldo Santana Neto, Pedro de Medeiros.

Resumo

Python é uma linguagem de alto nível, orientada a objetos, de programação de alto nível e de semântica dinâmica. Sua construção baseada em estruturas de dados de alto nível, combinada com sua "tipagem" e ligações dinâmicas tornam-a muito atraente tanto para o desenvolvimento rápido de aplicações quanto para criação de *scripts* ou como linguagem para interligar componentes já existentes. Python é simples, possui uma sintaxe fácil de aprender que dá enfase à legibilidade, reduzindo o custo de manutenção do código. Python suporta módulos e pacotes, os quais encorajam a modularização e o reaproveitamento de código. O interpretador Python e a extensa biblioteca padrão estão disponíveis na forma de código fonte, e binários para as principais plataformas, e pode ser distribuído livremente.

Este manual de referência descreve a sintaxe e a "semântica principal" da linguagem. É um resumo, mas tenta ser exato e completo. A semântica de objetos de tipo interno (*built-in*) e das funções e módulos são descritas na *Referência da Biblioteca Python*. Para uma introdução informal à linguagem, veja o *Tutorial Python*. Para programadores C ou C++ existem outros dois manuais adicionais: *Extendendo e Embutindo o Interpretador Python* fornece um retrato de alto nível sobre como escrever módulos de extensão para Python, e o *Manual de Referência da API Python/C* que descreve em detalhes as *interfaces* disponíveis para programadores C/C++.

SUMÁRIO

CAPÍTULO

ONE

Introdução

Este manual de referência descreve a linguagem de programação Python. Ele não tem a intenção de servir como um tutorial.

Tentarei ser o mais preciso possível, no entanto escolhi o uso do Português no lugar de uma especificação formal para tudo, exceto para sintaxe e analise léxica. Isso deixará o documento mais compreensível para a maioria dos leitores, mas deixará brechas para ambigüidades. Consequentemente, se você veio de Marte e tentar reimplementar a linguagem apenas com esse documento, provavelmente implementará uma linguagem um pouco diferente. Por outro lado, se você está usando Python e quer saber com precisão as regras sobre uma área da linguagem em particular, provavelmente você conseguirá encontrá-las aqui. Se deseja ver uma definição mais formal da linguagem, talvez você possa se voluntariar para a tarefa — ou inventar uma máquina de clonagem :-).

É perigoso colocar muitos detalhes de implementação num documento de referência de uma linguagem — a implementação pode mudar, e outras implementações da mesma linguagem podem funcionar de maneira diferente. Por outro lado, como existe apenas uma implementação de Python usada em grande escala (apesar de já existirem implementações alternativas), é importante mencionar suas peculiaridades, especialmente onde a implementação impõe limitações adicionais. Portanto, você encontrará algumas pequenas "notas de implementação" espalhadas pelo texto.

Todas as implementações de Python vem com um certo número de módulos internos (*built-in*) e módulos padrão. Esses módulos não serão documentados aqui, mas sim no documento *Referência da Biblioteca Python*. Alguns poucos módulos serão mencionados quando eles interagirem com a definição da linguagem de modo significativo.

1.1 Alternate Implementations

Though there is one Python implementation which is by far the most popular, there are some alternate implementations which are of particular interest to different audiences.

Known implementations include:

CPython — This is the original and most-maintained implementation of Python, written in C. New language features generally appear here first.

Jython — Python implemented in Java. This implementation can be used as a scripting language for Java applications, or can be used to create applications using the Java class libraries. It is also often used to create tests for Java libraries. More information can be found at the Jython website.

Python for .NET — This implementation actually uses the CPython implementation, but is a managed .NET application and makes .NET libraries available. This was created by Brian Lloyd. For more information, see the Python for .NET home page.

IronPython — An alternate Python for .NET. Unlike Python.NET, this is a complete Python implementation that generates IL, and compiles Python code directly to .NET assemblies. It was created by Jim Hugunin, the original creator of Jython. For more information, see the IronPython website.

PyPy — An implementation of Python written in Python; even the bytecode interpreter is written in Python. This is executed using CPython as the underlying interpreter. One of the goals of the project is to encourage

experimentation with the language itself by making it easier to modify the interpreter (since it is written in Python). Additional information is available on the PyPy project's home page.

Each of these implementations varies in some way from the language as documented in this manual, or introduces specific information beyond what's covered in the standard Python documentation. Please refer to the implementation-specific documentation to determine what else you need to know about the specific implementation you're using.

1.2 Notação

As descrições do analisador léxico e de sintaxe usam uma notação gramatical BNF modificada. Elas usam o seguinte estilo de definição:

```
nome:           lc_letra (lc_letra | "_")*
lc_letra:      "a"..."z"
```

A primeira linha diz que um `nome` é um `lc_letra` seguido de uma sequencia com zero ou mais `lc_letra` e sublinhados. Um `lc_letra` por sua vez é qualquer caracter entre '`a`' e '`z`'. (Esta regra é na verdade seguida para nomes definidos nas definições léxicas e gramaticais neste documento.)

Cada regra começa com um nome (o qual é o nome definido pela regra) e um sinal `::=`. Uma barra vertical (`|`) é usada para separar alternativas; este é o menor operador de conexão nesta notação. Um asterisco (`*`) significa zero ou mais repetições do item precedente; assim como, um sinal de adição (`+`) representa uma ou mais repetições, e um elemento dentro de colchetes (`[ ]`) significa zero ou uma ocorrência (em outras palavras, o elemento interno é opcional). Os operadores `*` e `+` conectam da forma mais forte possível; parenteses são usados para agrupamento. Strings literais são definidas entre aspas-duplas. Espaços em branco são usados com a intenção de separar *tokens* (elemento léxico). Regras são geralmente definidas em apenas uma linha; regras com muitas alternativas podem ser formatadas com cada linha iniciando com uma barra vertical.

Nas definições léxicas (como no exemplo acima), duas outras convenções são usadas: dois caracteres literais separados por três pontos significam a escolha de qualquer caracter na faixa especificada (inclusive) dos caracteres ASCII. Um elemento entre os sinais de menor e maior (`<...>`) fornece uma descrição informal do símbolo definido; ex., pode ser usado para descrever o conceito de 'caracter de controle' se necessário.

Mesmo a notação utilizada sendo quase sempre a mesma, existe uma grande diferença entre o significado das definições léxica e sintática: uma definição léxica opera em caracteres individuais de uma fonte de entrada, enquanto a definição da sintaxe opera sobre um fluxo de *tokens* gerados por um analisador léxico. Todos os usos de BNF no próximo capítulo ("Análise Léxica") são definições léxicas; usos nos capítulos subseqüentes são definições sintáticas.

CAPÍTULO

TWO

Análise léxica

Um programa em Python é lido por um *parser*. A entrada de dados do parser é uma sequência de *tokens*, gerados pelo *analisador léxico*. Este capítulo descreve como essa análise divide o conteúdo de um arquivo em tokens.

Python usa o conjunto de caracteres de 7 bits ASCII para o texto de um programa. New in version 2.3: Uma declaração de página de código pode ser usada para indicar que as strings literais e os comentários usam um código diferente de ASCII.. Por questões de compatibilidade com versões anteriores, o interpretador Python vai avisar apenas se encontrar caracteres de 8 bits; pode-se corrigir o problema declarando explicitamente a página de código usada ou usando sequências de escape, se aqueles bytes forem dados binários ao invés de caracteres.

O conjunto de caracteres usado durante a execução depende dos dispositivos de entrada e saída conectados ao programa, mas geralmente é um subgrupo de ASCII.

Nota de compatibilidade futura: Pode ser tentador assumir que o conjunto de caracteres usado para caracteres de 8 bits é o ISO Latin-1 (um subgrupo do ASCIIque cobre a maioria dos idiomas ocidentais que utilizam o alfabeto latino), mas é possível que no futuro os editores de texto que usem Unicode se tornem mais populares. Normalmente, esses editores utilizam o UTF-8, que é também um subgrupo de ASCII, mas que usa de forma diferente os caracteres com código de 128 a 255. Apesar de ainda não haver nenhum consenso nesse assunto, não é recomendável assumir que qualquer um dos dois será o dominante, ainda que a implementação atual favoreça o Latin-1. Isto se aplica tanto aos caracteres usados no código-fonte quanto aos utilizados durante a execução.

2.1 Estrutura das linhas

Um programa em Python é dividido em um conjunto de *linhas lógicas*.

2.1.1 Linhas lógicas

O fim de uma linha lógica é representado pelo símbolo (token) NEWLINE. Comandos (statements) não podem ultrapassar os limites da linha lógica exceto quando uma nova linha é permitida pela sintaxe (por exemplo, entre comandos em um comando composto). Uma linha lógica é construída de uma ou mais *linhas físicas*, seguindo as regras implícitas ou explícitas de *união de linhas*.

2.1.2 Linhas físicas

A physical line is a sequence of characters terminated by an end-of-line sequence. In source files, any of the standard platform line termination sequences can be used - the UNIXform using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform.

When embedding Python, source code strings should be passed to Python APIs using the standard C conventions for newline characters (the `\n` character, representing ASCII LF, is the line terminator).

2.1.3 Comentários

Comentários iniciam com uma tralha (#) que não seja parte de uma string literal e terminam no fim da linha física. Um comentário significa o fim da linha lógica a menos que as regras implícitas de união de linhas sejam usadas. Comentários são ignorados pela sintaxe; eles não são tokens.

2.1.4 Declarações de página de código

Se um comentário na primeira ou segunda linha de um arquivo de código Python coincidir com a expressão regular definida por `coding[=:]\s*([-\w.]+)`, este comentário é processado como uma declaração de código de caracteres: o primeiro grupo da expressão é o nome da página de código. As formas recomendadas são

```
# -*- coding: <nome-do-código> -*-
```

Que também é reconhecido pelo GNU Emacs, e:

```
# vim:fileencoding=<nome-do-código>
```

Que é reconhecido pelo VIM. Além disso, se os primeiros bytes do arquivo são (`'\xef\xbb\xbf'`), o código usado é UTF-8 (isso é reconhecido pelo bloco de notas, do Microsoft Windows, entre outros editores)

Se uma página de código for declarada, o nome tem de ser reconhecido pelo interpretador. A página de código é usada para toda e qualquer análise léxica, em especial para encontrar o fim de uma string, e para interpretar o conteúdo de strings literais. Strings literais são convertidas para Unicode antes de serem analisadas, e são então convertidas de volta para o código original antes do interpretador começar a execução. A declaração de página de código tem de aparecer sozinha na linha.

2.1.5 União Explícita de linhas

Duas ou mais linhas físicas podem ser unidas em linhas lógicas usando barras invertidas (\) da seguinte forma: quando uma linha física termina em uma barra invertida que não é parte de uma string literal ou um comentário, ela é unida com a linha seguinte, formando uma única linha lógica, sem a barra e o caracter de nova linha a seguir. Por exemplo:

```
if 1900 < ano < 2100 and 1 <= mes <= 12 \
   and 1 <= dia <= 31 and 0 <= hora < 24 \
   and 0 <= minuto < 60 and 0 <= segundo < 60:   # Parece ser uma data válida
        return 1
```

Uma linha terminando em barra invertida não pode conter comentários. Barras invertidas não continuam comentários. Eles não continuam quaisquer elementos, exceto strings literais (ou seja, elementos outros que strings não podem ser divididos em várias linhas usando barras invertidas). Uma barra invertida é ilegal em qualquer outro lugar que não seja uma string literal.

2.1.6 União Implícita de linhas

Expressões em parênteses, colchetes ou chaves, podem ser divididas em mais de uma linha, sem usar a barra invertida. Exemplo:

```
nomes_de_meses = ['Januari', 'Februari', 'Maart',      # Estes são os nomes
                  'April',   'Mei',      'Juni',       # holandeses para os
                  'Juli',    'Augustus', 'September',  # meses do ano
                  'Oktober', 'November', 'December']
```

Linhas continuadas explicitamente podem conter comentários. A indentação da linha de continuação não é importante. Linhas em branco são permitidas. Não há qualquer caracter de NEWLINE entre linhas continuadas implicitamente. Elas também podem ocorrer em strings com três aspas (abaixo); neste caso, não podem conter comentários.

2.1.7 Linhas em Branco

Uma linha lógica que contenha apenas espaços, tabulações, parágrafos e possivelmente um comentário, é ignorada (ou seja, não é gerado uma NEWLINE). Durante o uso do interpretador interativo, o manuseio de linhas em branco pode diferir dependendo da implementação do interpretador. Na implementação padrão, uma linha lógica completamente em branco (isto é, que não tenha nem mesmo espaços ou comentários) finalizam um comando multi-linhas.

2.1.8 Indentação

Espaços em branco (ou tabulações) no início de uma linha lógica é usado para computar o nível de indentação da linha, que por sua vez é usado para determinar o agrupamento de comandos.

Primeiro, tabulações são substituídas (da esquerda pra direita) por um número de espaços que pode variar de um a oito, de forma que o total de caracteres seja sempre um múltiplo de oito (a mesma regra usada por sistemas UNIX). O número total de espaços precedendo o primeiro caracter válido determina o nível de indentação da linha. A indentação não pode ser dividida em várias linhas físicas usando barras invertidas; os espaços em branco até a primeira barra invertida determinam o nível de indentação.

nota: não é muito aconselhável misturar espaços e tabulações usados como indentação em um único arquivo de código fonte. Também deve-se notar que diferentes plataformas podem limitar explicitamente o nível máximo de indentação.

Um caracter *formfeed* pode estar presente no começo da linha; ele será ignorado para o cálculo de indentação descrito acima. Quando ocorrem em qualquer outro ponto dos espaços precedendo a linha, têm efeito indefinido (por exemplo, podem zerar a contagem de espaços).

O nível de indentação de linhas consecutivas é usado para gerar tokens INDENT e DEDENT, usando uma pilha, da seguinte forma:

Antes de ler a primeira linha do arquivo, um zero é inserido na pilha; ele jamais sairá dela. Os números inseridos sempre aumentarão, progressivamente, da base para o topo. No início de cada linha lógica, o nível de indentação da linha é comparado com o do topo da pilha. Se for igual, nada acontece; se for maior, é inserido na pilha, e um token INDENT é gerado. Se for menor, ele obrigatoriamente tem de ser um dos números da pilha; todos os números que são maiores que ele são removidos e para cada um deles, um token DEDENT é gerado. No final do arquivo, um token DEDENT é gerado para cada número maior que zero restante na pilha.

Segue um trecho de código que apesar de confuso, está indentado corretamente:

```
def perm(l):
        # Computa a lista de todas as permutações de 1
    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 diversos erros de indentação:

```
 def perm(l):                       # erro: primeira linha indentada
for i in range(len(l)):             # erro: sem indentação
    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                # error: indentação inconsistente
```

(Na verdade, os três primeiros erros são detectados pelo parser; apenas o último erro é encontrado durante a análise léxica — a indentação de `return r` não coincide com o nível que foi removido da pilha.)

2.1.9 Espaço em branco entre elementos

Os caracteres espaço, tabulação e *formfeed* podem ser usados para separar símbolos, exceto quando no começo de uma linha lógica ou em strings literais. Espaços são necessários entre dois símbolos somente se a sua concatenação levar a interpretá-los como um símbolo diferente (por exemplo, ab é um símbolo, mas a b são dois).

2.2 Outros símbolos

Além de NEWLINE, INDENT e DEDENT, as seguintes categorias de tokens existem: *identificadores*, *keywords*, *literais*, *operadores*, e *delimitadores*. Espaços (quando não forem terminações de linhas, discutidos mais adiante) não são tokens, mas servem para delimitá-los. Quando existir ambiguidade, um token compreende a mais longa string que forma um símbolo legal, lido da esquerda pra direita.

2.3 Identificadores e keywords

Identificadores (também chamados de *nomes*) são descritos pela seguinte definição léxica:

```
identifier  ::=  (letter|"_") (letter | digit | "_")*
letter      ::=  lowercase | uppercase
lowercase   ::=  "a"..."z"
uppercase   ::=  "A"..."Z"
digit       ::=  "0"..."9"
```

Identificadores não tem limite de comprimento, mas o uso de maiúsculas e minúsculas tem importância.

2.3.1 Keywords

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

```
and       del       for       is        raise
assert    elif      from      lambda    return
break     else      global    not       try
class     except    if        or        while
continue  exec      import    pass      yield
def       finally   in        print
```

Note que apesar do identificador `as` poder ser usado como parte da sintaxe da instrução `import`, no momento ele não é um keyword.

Em uma futura versão da linguagem, é possível que os identificadores `as` e `None` tornar-se-ão keywords.

2.3.2 Classes reservadas de identificadores

Algumas classes de identificadores (exceto keywords), têm significados especiais. Estas classes são identificadas pelo padrão da disposição do caracter underscore antes e depois do identificador:

`_*` Não é importado por '`from` *module* `import *`'. O identificador especial '_' é usado no interpretador interativo para armazenar o resultado da última expressão resolvida: ele é armazenado no módulo `__builtin__`. Quando não o interpretador não estiver em modo interativo, '_' não tem qualquer significado especial e não é definido. Veja section 6.12, "O comando import."

Note: O nome '_' é às vezes usado para internacionalização; procure a documentação para o módulo `gettext` module para mais informação sobre esta convenção.

`__*__` Nomes definidos pelo sistema. Estes nomes são definidos pelo interpretador e sua implementação (incluindo a biblioteca padrão); Assume-se que as aplicações não definam nomes adicionais usando esta convenção. O conjunto de nomes desta classe definidos pela linguagem pode ser estendido em versões futuras. Veja section 3.3, "Nomes de métodos especiais."

`__*` Nomes privados da classe. Nomes nesta categoria, quando usados dentro do contexto de uma definição de classe, são re-escritos para usar uma versão modificada, para evitar conflitos entre atributos "privados" entre a classe base e suas classes derivadas. Veja section 5.2.1, "Identificadores (Nomes)."

2.4 Literais

Literais são notações para valores constantes de alguns tipos *built-in*.

2.4.1 Strings literais

Strings literais são descritas pelas seguintes definições léxicas:

```
stringliteral    ::=  [stringprefix](shortstring | longstring)
stringprefix     ::=  "r "u "ur "R "U "UR "Ur "uR"
shortstring      ::=  "'"shortstringitem* "' '"' shortstringitem* '"'
longstring       ::=  "'''"longstringitem* "'''"
                      | '"' longstringitem* '"'
shortstringitem  ::=  shortstringchar | escapeseq
longstringitem   ::=  longstringchar | escapeseq
shortstringchar  ::=  <any character except "\"or newline or the quote>
longstringchar   ::=  <any character except "\»
escapeseq        ::=  "\«any ASCII character>
```

Uma restrição sintática não indicada por essas definições é que espaços não são permitidos entre o `stringprefix` e o resto da string literal.

Strings literais podem ser inseridas entre aspas simples (`'`) ou aspas duplas (`"`). Elas podem também ser inseridas em groupos de três aspas duplas ou simples. Uma barra-invertida (`\`) pode ser usada para escapar caracteres que de outra forma teriam algum significado especial, como newline, a própria barra-invertida ou aspas. Strings literais podem opcionalmente ter como prefixo a letra '`r`' ou '`R`'; tais strings são chamadas *raw strings* e usam regras diferentes para a interpretação de sequências de escape. Uma string prefixada pelo character '`u`' or '`U`' torna-se uma string Unicod, que utiliza o conjunto de caracteres Unicode, definido pela norma ISO 10646. Algumas sequências adicionais, descritas adiante, estão disponíveis em strings Unicode. Estes dois caracteres prefixando strings podem ser utilizados simultaneamente; neste caso, '`u`' deve vir antes de '`r`'.

Em strings com três aspas, newlines ou aspas são permitidas (e mantidas), exceto por três aspas seguidas, que terminam a string.

A menos que a string tenha o prefixo '`r`' ou '`R`', sequências de escape são interpretadas segundo regras semelhantes aquelas usadas em C. As sequências reconhecidas são:

Escape Sequence	**Meaning**	**Notes**
`\`*newline*	Ignored	
`\\`	Backslash (`\`)	
`\'`	Single quote (`'`)	
`\"`	Double quote (`"`)	
`\a`	ASCII Bell (BEL)	
`\b`	ASCII Backspace (BS)	
`\f`	ASCII Formfeed (FF)	
`\n`	ASCII Linefeed (LF)	
`\N{`*name*`}`	Character named *name* in the Unicode database (Unicode only)	
`\r`	ASCII Carriage Return (CR)	
`\t`	ASCII Horizontal Tab (TAB)	
`\u`*xxxx*	Character with 16-bit hex value *xxxx* (Unicode only)	(1)
`\U`*xxxxxxxx*	Character with 32-bit hex value *xxxxxxxx* (Unicode only)	(2)
`\v`	ASCII Vertical Tab (VT)	
`\`*ooo*	Character with octal value *ooo*	(3,5)
`\x`*hh*	Character with hex value *hh*	(4,5)

Notas:

(1) Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.

(2) Any Unicode character can be encoded this way, but characters outside the Basic Multilingual Plane (BMP) will be encoded using a surrogate pair if Python is compiled to use 16-bit code units (the default). Individual code units which form parts of a surrogate pair can be encoded using this escape sequence.

(3) As in Standard C, up to three octal digits are accepted.

(4) Unlike in Standard C, at most two hex digits are accepted.

(5) In a string literal, hexadecimal and octal escapes denote the byte with the given value; it is not necessary that the byte encodes a character in the source character set. In a Unicode literal, these escapes denote a Unicode character with the given value.

Ao contrário de C , todas as sequências de escape não reconhecidas são deixadas na string inalteradas, ou seja, *a barra invertida é deixada na string*. (Este comportamente é útil na depuração do código: se uma sequência de escape foi digitada errada, a saída resultante é facilmente reconhecida.) Também é importante notar que sequências marcadas como "(Apenas Unicode)" na tabela acima, caem na categora de sequências não reconhecidas para strings que não sejam Unicode.

Quando o prefixo '`r`' ou '`R`' está presente, o caracter seguinte à barra invertida é incluído na string sem alterações, e *todas as barras são deixadas na string.* Por exemplo, `r"\n"` c é uma string literal válida que consiste de dois caracteres: uma barra invertida e um '`n`' minúsculo. Aspas simples podem ser escapadas com a barra invertida,

mas ela continua na string; por exemplo, `r"\` é uma string literal válida consistindo de dois caracteres: uma barra invertida e aspas duplas; `r"\"` não é uma string válida (mesmo uma string *crua* não pode terminar num número impar de barras invertidas). A string não pode terminar com uma barra invertida (já que ela irá escapar as aspas logo em seguida).

Quando o prefixo '`r`' ou '`R`' é usado com o prefixo '`u`' ou '`U`' prefix, então a sequência `\uXXXX` é processada enquanto *todas as outras barras invertidas são deixadas na string*. Por exemplo, a string literal `ur"\u0062\n"` consiste de três caracteres Unicode, 'LATIN SMALL LETTER B', 'REVERSE SOLIDUS' e 'LATIN SMALL LETTER N'. Barras invertidas podem ser escapadas com uma outra barra; no entanto, ambas ficam na string. Como resultado sequências de escape`\uXXXX` só são reconhecidas quando há um número ímpar de barras invertidas.

2.4.2 Concatenação de strings literais

Múltiplas strings literais adjacentes (delimitadas por espaços, possivelmente usando convenções diferentes para as aspas) são permitidas, e seu significado é o mesmo da sua concatenação. Desta forma, `"hello"'world'` é equivalente a `"helloworld"`. Esse recurso pode ser usado para reduzir o número de barras-invertidas necessárias, para repartir strings longas através de várias linhas, ou mesmo para adicionar comentários a partes de uma string, por exemplo:

```
re.compile("[A-Za-z_]"       # letra ou underscore
           "[A-Za-z0-9_]*"   # letra, dígito ou underscore
          )
```

Note que este recurso é definido no nível sintático, mas implementado no momento da compilação. O operador '+' deve ser usado para concatenar strings em tempo de execução. Note ainda que concatenação literal pode usar diferentes tipos de aspas para cada componente (até misturar strings *cruas* e strings com três aspas).

2.4.3 Literais numéricos

Há quatro tipos de literais numéricos: inteiros, longos, números de ponto flutuante e números imaginários. Não há literais complexos (eles podem ser formados adicionando um número real e um número imaginário).

Note que literais numéricos não incluem um sinal; um trecho como `-1` é na verdade uma expressão composta do operador '`-`' e do número `1`.

2.4.4 Inteiros e longos

Literais inteiros e longos são descritos pelas seguintes definições léxicas:

```
longinteger     ::=  integer ("l "L")
integer         ::=  decimalinteger | octinteger | hexinteger
decimalinteger  ::=  nonzerodigit digit* | "0"
octinteger      ::=  "0"octdigit+
hexinteger      ::=  "0"("x "X") hexdigit+
nonzerodigit    ::=  "1"..."9"
octdigit        ::=  "0"..."7"
hexdigit        ::=  digit | "a"..."f "A"..."F"
```

Apesar de tanto o caracter '`l`' e '`L`' serem permitidos como sufixo para longos, é recomendado usar sempre o '`L`', já que a letra '`l`' se parece muito com o dígito '`1`'.

Inteiros decimais que estão acima do maior inteiro representável (2147483647 quando usamos 32-bits) são aceitos como se fossem longos. [1] There is no limit for long integer literals apart from what can be stored in available memory.

[1] In versions of Python prior to 2.4, octal and hexadecimal literals in the range just above the largest representable plain integer but below the largest unsigned 32-bit number (on a machine using 32-bit arithmetic), 4294967296, were taken as the negative plain integer obtained by subtracting 4294967296 from their unsigned value.

Octais e hexadecimais se comportam de maneira semelhante, mas quando ultrapassam a maior representação possível, mas estão abaixo do maior número de 32-bits possível, 4294967296, eles aparecem como o inteiro negativo obtido da subtração de 4294967296 do seu valor original. Não há limite para longos, exceto, é claro, o quanto pode ser armazenado na memória.

Alguns exemplos de inteiros (primeira linha) e longos (segunda e terceira linhas):

```
7     2147483647                        0177
3L    79228162514264337593543950336L    0377L   0x100000000L
      79228162514264337593543950336             0xdeadbeef
```

2.4.5 Literais de ponto-flutuante

Números de ponto-flutuante são descritos pelas seguintes definições léxicas:

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

Note que as partes de um número de ponto flutuante podem parecer como inteiros octais, mas são interpretadas usando base 10. Por exemplo, '`077e010`' é legal, e denota o mesmo número que '`77e10`'. O intervalo permitido de números de ponto flutuante depende da implementação. Alguns exemplos:

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

Note que literais numéricos não incluem um sinal; um trecho como `-1` é na verdade uma expressão composta do operador '`-`' e do número `1`.

2.4.6 Literais imaginários

Números imaginários são descritos pelas seguintes definições léxicas:

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

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

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

2.5 Operadores

Os seguintes símbolos são operadores:

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

Os operadores de comparação `<>` e `!=` são formas alternativas do mesmo operador. `!=` é a forma preferida; `<>` está obsoleto.

2.6 Delimitadores

Os seguintes símbolos funcionam como delimitadores:

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

O ponto pode ocorrer ainda em números de ponto-flutuante e literais imaginários. Uma sequência de três pontos tem o significado especial de uma elipse no fatiamento de sequências. Os operadores da segunda metada da lista, servem como delimitadores, mas também executam uma operação.

Os seguintes caracteresASCII têm significado especial quando partes de outros símbolos ou são de alguma forma significantes para a análise léxica:

```
’       "       #       \
```

Os seguintes caracteres ASCII não são usados em Python. Sua ocorrência fora de strings ou comentários é um erro incondicional:

```
$       ?
```

CAPÍTULO

THREE

Modelo de Dados

3.1 Objetos, valores e tipos

Objetos são a abstração para dados em Python. Todos os dados em um programa Python são representados por objetos ou por relações entre objetos. (Em um certo sentido, e em concordância com o modelo de "programas armazenados em computadores" de Von Neumann, o código também é representado por objetos.)

Todo objeto possui uma identidade, um tipo e um valor. A *identidade* de um objeto nunca muda após sua criação; você pode imaginá-la como sendo o endereço de memória de um objeto. O operador '`is`' compara a identidade de dois objetos; a função `id()` retorna um número inteiro que representa a identidade de um objeto (atualmente implementada como sendo o seu endereço). O *tipo* de um objeto também é inalterável.[1] O tipo do objeto determina as operações que o objeto suporta (ex., "ele tem um comprimento?") e os possíveis valores para objetos deste tipo. A função `type()` retorna o tipo do objeto (que também é um objeto). O *valor* de alguns objetos pode mudar. Objetos que podem mudar seu valor são denominados *mutáveis*; objetos que não podem mudar seu valor após terem sido criados são chamados de *imutáveis*. (O valor de um objeto contêiner imutável que possui uma referência para um objeto mutável pode alterar quando o valor do último é alterado; porém, o conteúdo do primeiro ainda é considerado imutável porque a coleção de objetos deste conteúdo não pode ser modificada. Logo, imutabilidade não é estritamente o mesmo que ter um valor inalterável, é algo 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.

Objetos nunca são explicitamente destruídos; entretanto, quando eles se tornam inacessíveis podem ser coletados pelo sistema de 'coleta de lixo' (*garbage-collect*). Uma implementação pode adiar 'coletas de lixo' ou omití-las completamente — contanto que respeite a regra de que nenhum objeto ainda acessível será coletado, depende apenas da qualidade da implementação a forma exata com que a coleta é realizada. (Nota de implementação: a implementação atual usa um esquema de contagem de referências com uma detecção de referências cíclicas (opcional) que coleta a maioria dos objetos assim que eles se tornam inacessíveis, mas não garante a coleta de lixo de objetos com referências circulares. Veja a *Referência da Biblioteca Python* para informações sobre o controle de coleta de lixo de objetos com referências circulares.)

Observe que o uso de recursos de rastreamento ou de depuração pode manter objetos vivos onde normalmente seriam coletados. Note também que a captura de uma exceção com os comandos '`try...except`' pode manter os objetos ativos.

Alguns objetos contêm referências para recursos "externos" como um arquivo aberto ou janela. É de se imaginar que estes recursos são liberados quando um objeto é coletado, mas como não há garantias de que a coleta aconteça, alguns objetos fornecem uma maneira de liberar os recursos externos, normalmente um método `close()`. É extremamente recomendado que programas fechem explicitamente esses objetos. O comando '`try...finally`' fornecem uma maneira conveniente para se fazer isso.

Alguns objetos possuem referências para outros objetos; eles são chamados *contêineres*. Exemplos de contêineres são tuplas, listas e dicionários. As referências são parte do valor de um contêiner. Na maioria dos casos, quando nós falamos sobre o valor de um contêiner, nós estamos falando dos valores, não das identidades dos objetos

[1] Desde Python 2.2, foi iniciada uma unificação gradual de tipos e classes, portanto, uma ou outra afirmação feita neste manual pode não estar 100% precisa e completa: por exemplo, agora *é* possível em alguns casos mudar o tipo de um objeto, sob certas condições controladas. Até que este manual passe por uma revisão extensiva ele não deve ser considerado como autoridade definitiva com exceção das considerações acerca das "classes tradicionais", que ainda são o padrão, por motivos de compatibilidade, em Python 2.2 e 2.3

contidos; entretanto, quando nós falamos sobre a mutabilidade de um contêiner, estamos tratando apenas das identidades dos objetos imediatamente contidos. Então, se um contêiner imutável (como uma tupla) contêm uma referência para um objeto mutável, seu valor muda se o objeto mutável for alterado.

Os tipos afetam quase todos os aspectos do comportamento de um objeto. A importância da identidade é afetada do mesmo jeito: para tipos imutáveis, operações que calculam novos valores podem retornar uma referência para um objeto existente com o mesmo tipo e valor, enquanto para objetos mutáveis isto não é permitido. Ex., depois de '`a = 1; b = 1`', `a` e `b` podem ou não se referirem ao mesmo objeto com o valor um, dependendo da implementação, mas '`c = []; d = []`', `c` e `d` tem que garantir que se referem a duas listas diferentes, unicas, recém-criadas e vazias. (Note que '`c = d = []`' atribuem o mesmo objeto tanto para `c` quanto para `d`.)

3.2 A hierarquia padrão de tipos

Abaixo está uma lista de tipos que são internas ao interpretador 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 nesta hierarquia (ex., números racionais, arrays com armazenamento eficiente de inteiros, etc.).

Algumas das descrições de tipos abaixo contêm um parágrafo listando 'atributos especiais.' São atributos que fornecem acesso à implementação e cuja intenção não é de uso geral. Suas definições podem mudar no futuro.

None Este tipo tem um valor único. Existe um único objeto com este valor. Este objeto é acessado através do nome interno `None`. Ele é usado com o significado de ausência de valor em várias situações, ex., ele é retornado por funções que não retornam nada explicitamente. Seu valor lógico é falso.

NotImplemented Este tipo tem um valor único. Existe um único objeto com este valor. Este objeto é acessado através do nome interno `NotImplemented`. Métodos numéricos e métodos completos de comparação podem retornar este valor se eles não implementam a operação para os operandos fornecidos. (O interpretador tentará então uma operação refletida, ou alguma chamada alternativa, dependendo do operador.) Seu valor lógico é verdadeiro.

Ellipsis Este tipo tem um valor único. Existe um único objeto com este valor. Este objeto é acessado através do nome interno `Ellipsis`. Ele é usado para indicar a presença da sintaxe '`...`' em uma fatia. Seu valor lógico é verdadeiro.

Números São criados por números literais e retornados como resultado por operadores aritméticos e funções aritméticas internas. Objetos numéricos são imutáveis; uma vez criados seus valores nunca mudam. Números em Python são fortemente relacionados com números matemáticos, mas sujeitos às limitações da representação numérica em computadores.

Python faz distinção entre inteiros, números de ponto flutuante, e números complexos:

Inteiros Representam elementos do conjunto matemático dos números inteiros (positivos e negativos).

Existem três tipos de inteiros:

Inteiros Planos Representam números na faixa de -2147483648 até 2147483647. (A faixa pode ser maior em máquinas com um tamanho de palavra maior, mas não menor.) Quando o resultado de uma operação fica fora dessa faixa, o resultado é retornado normalmente como um inteiro longo (em alguns casos, a exceção `OverflowError` é levantada no lugar). Para propósito de deslocamento (binário) e operações com máscaras, assume-se que eles usem notação binária de 32 bits ou mais e números negativos são representados usando complemento de dois, e não escondem nenhum bit do usuário (ex., todos os 4294967296 padrões de bits diferentes correspondem a um valor diferente).

Inteiros Longos Representam números numa faixa ilimitada, sujeita apenas à memória (virtual). Para propósito de deslocamento (binário) e operações com máscaras, assume-se que usem notação binária e números negativos são representados com uma variação do complemento de dois que dá a ilusão de uma string infinita com o bit de sinal extendendo para a esquerda.

Booleanos Representam os valores lógicos False (Falso) e True (Verdadeiro). Os dois objetos que representam os valores False e True são os únicos objetos Booleanos. O tipo Booleano é um subtipo dos inteiros planos, e valores Booleanos se parecem com os valores 0 e 1, respectivamente,

em quase todos os contextos, a exceção é quando são convertidos para uma string, as strings `"False"` ou `"True"` são retornadas, respectivamente.

As regras para representação de inteiros foram feitas com a intenção de facilitar a interpretação de operações com máscara e deslocamento envolvendo inteiros negativos, e com o mínimo de problemas quando eles trocam entre os domínios dos inteiros planos e dos inteiros longos. Qualquer operação exceto o deslocamento à esquerda, se mantiver o resultado no mesmo domínio dos inteiros planos sem causar sobrecarga, irá manter o resultado no mesmo domínio dos inteiros longos ou ao usar operadores mistos.

Números de ponto flutuante Eles representam os números de ponto flutuante de precisão dupla no nível da máquina. Vai depender da arquitetura da máquina (e da implementação de C ou Java) para a definir qual é a faixa aceita e a manipulação de sobrecargas. Python não suporta números de ponto flutuante de precisão simples; a economia de processamento e de uso de memória que geralmente são a razão para usar este tipo de técnica são mínimas perto do trabalho de usar objetos em Python, então não há razão para complicar a linguagem com dois tipos de números de ponto flutuante.

Números complexos Representam os números complexos com um par de números de ponto flutuante de precisão dupla no nível da máquina. As mesmas características se aplicam como no caso dos números de ponto flutuante. As partes reais e imaginárias de um número complexo `z` podem ser obtidas através dos atributos somente de leitura `z.real` e `z.imag`. `z.real` e `z.imag`.

Seqüências Representam conjuntos finitos ordenados indexados por um número não-negativo. A função interna `len()` retorna o número de ítens da seqüência. Quando o tamanho de uma seqüência é *n*, o conjunto de índices contém os números 0, 1, ..., *n*-1. O ítem *i* de uma seqüência *a* é retornado por *a*`[`*i*`]`.

Seqüências também suportam fatiamento (*slicing*): *a*`[`*i*`:`*j*`]` retorna todos os ítens com índice *k* onde *i* <=*k* < *j*. Quando usado como uma expressão, uma fatia é uma seqüência do mesmo tipo. Isto implica que o conjunto de índices é renumerado para que comece em 0.

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

Seqüências são divididas de acordo com sua "mutabilidade":

Seqüências imutáveis Um objeto do tipo seqüência imutável não pode ser alterado após sua criação. (Se o objeto contém referências para outros objetos, estes objetos podem ser mutáveis e podem ser alterados; todavia, a coleção de objetos diretamente referenciadas por um objeto imutável não pode alterar.)

Os seguintes tipos são seqüências imutáveis:

Strings Os items de uma string são caracteres. Não há um tipo à parte para caracteres; eles são representados por uma string de um único item. Caracteres representam (pelo menos) um byte. As funções internas `chr()` e `ord()` convertem entre caracteres e números inteiros positivos representando os valores. Bytes com os valores de 0 a 127 geralmente representam o valor ASCII correspondente, mas a sua interpretação correta depende do programa. O tipo string é ainda usado para representar matrizes de dados, por exemplo, para guardar dados lidos de um arquivo.
(Em sistemas cujo conjunto de caracteres nativo não é ASCII, strings podem usar EBCDIC em sua representação interna, contanto que as funcões `chr()` e `ord()` implementem um mapeamento entre EBCDIC, e comparações de strings preservem a ordem ASCII. Ou será que talvez alguém pode propôr uma regra melhor?)

Unicode Os items de um objeto Unicode são unidades de código Unicode. São representadas por um objeto Unicode de um item e podem conter valores de 16 ou 32 bits, representando um caracter ordinal Unicode (o valor máximo é dado por `sys.maxunicode` e depende de como o interpretador Python foi configurado no momento da compilação). Pares *surrogate* podem estar presentes no objeto Unicode, e serão reportados como dois items separados. As funções internas `unichr()` e `ord()` convertem entre unidades de código e inteiros positivos, representando os ordinais Unicode, como definidos no *Unicode Standard 3.0*. Conversão de e para outros códigos são possíveis através do método Unicode `encode` e a função interna `unicode()`.

Tuples Os items de uma tupla são quaisquer objetos Python. Tuplas de dois ou mais items são formadas por uma lista de expressões, separadas por vírgula. Uma tupla de um item ('singleton') pode ser formada adicionando uma vírgula a uma expressão (uma expressão sozinha entre parênteses não cria uma tupla, já que parênteses podem ser usados para criar grupos de expressões). Uma tupla vazia pode ser criada por um par de parênteses.

Sequências mutáveis Sequências mutáveis podem ser alteradas depois de criadas. As notações de subscrição e fatiamento podem ser usadas como o alvo de atribuições ou do comando (del).

Atualmente há apenas uma sequência mutável intrínseca à linguagem:

Listas Os items de uma lista são quaisquer objetos Python. Listas são criadas colocando uma lista de expressões separadas por vírgulas entre dois colchetes. (Note que não há nenhuma regra especial necessária para criar listas de comprimento 0 ou 1.)

O módulo `array` fornece um exemplo adicional de uma sequência mutável.

Mapeamentos Mapeamentos representam conjuntos finitos de objetos, indexados por índices arbitrários. A notação `a[k]` seleciona o item indexado por `[k]` do mapeamento `a`; isto pode ser usado em expressões e como o alvo de atribuições ou do comando `del`. A função interna `len()` retorna o número de items num mapeamento.

Atualmente há apenas um único mapeamento intrínseco à linguagem:

Dicionários Dicionários () representam conjuntos finitos de objetos, indexados por índices praticamente arbitrários. O único tipo de objeto inaceitável como chave são listas ou dicionários ou outros objetos mutáveis que são comparados pelo valor e não pela identidade. A razão para isso é que uma implementação eficiente de dicionários requer que o valor do *hash* da chave permaneça constante. Tipos numéricos usados para chaves obedecem as regras normais de comparação: se na comparação dois números são considerados iguais (por exemplo, `1` e `1.0`) então qualquer um deles pode ser usado para indexar o mesmo valor em um dicionário.

Dicionários são mutáveis; eles podem ser criados pela notação `{...}` (veja a seção 5.2.6, "Visualização de dicionários").

Os módulos `dbm`,`gdbm`, `bsddb` fornecem exemplos adicionais de mapeamentos.

Tipos executáveis Estes são os tipos aos quais a operação de chamada de função (veja section **??**, "Chamadas") pode ser aplicada:

Funções Um objeto função é criado por uma definição de função (veja a section 7.5, "Definições de função"). Ele deve ser chamado com uma lista de argumentos contendo o mesmo número que itens que na definição formal da lista de parâmetros do objeto.

Special attributes:

Attribute	Meaning
`func_doc`	The function's documentation string, or `None` if unavailable
`__doc__`	Another way of spelling `func_doc`
`func_name`	The function's name
`__name__`	Another way of spelling `func_name`
`__module__`	The name of the module the function was defined in, or `None` if unavailable.
`func_defaults`	A tuple containing default argument values for those arguments that have defaults, or `None` if n
`func_code`	The code object representing the compiled function body.
`func_globals`	A reference to the dictionary that holds the function's global variables — the global namespace
`func_dict`	The namespace supporting arbitrary function attributes.
`func_closure`	`None` or a tuple of cells that contain bindings for the function's free variables.

Most of the attributes labelled "Writable" check the type of the assigned value.

Changed in version 2.4: `func_name` is now writable.

Function objects also support getting and setting arbitrary attributes, which can be used, for example, to attach metadata to functions. Regular attribute dot-notation is used to get and set such attributes. *Note that the current implementation only supports function attributes on user-defined functions. Function attributes on built-in functions may be supported in the future.*

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

Métodos Um objeto método combina uma classe, uma instância dessa classe (ou `None`) e qualquer objeto executável (normalmente uma função definida pelo usuário).

Atributos especiais (somente leitura): `im_self` é a instância da classe, `im_func` é o objeto função; `im_class` é a classe de `im_self` para métodos ligados ou a classe que pediu a execução do método para métodos desligados; `__doc__` é a documentação do método (o mesmo que `im_func.__doc__`); `__name__` é o nome do método (o mesmo que `im_func.__name__`); `__module__` é o nome do módulo em que o método foi definido, ou `None` se indisponível. Changed in version 2.2: `im_self` é usado para referir à classe que definiu o método.

Métodos também suportam o acesso (mas não a definição) de atributos arbitrários do objeto função subjacente.

Método, ou o objeto que refere a um método, podem ser criados quando acessando um atributo de uma classe (talvez atrás de uma instância dessa classe), se esse atributo for um objeto função, um objeto método definido pelo usuário ou um método da classe.

Quando o atributo é um objeto método, um objeto novo só será criado se a classe da qual ele está sendo consultado for ou a classe original armazenada no objeto, ou uma classe derivada dela; em outros casos, o objeto original é utilizado.

Quando um objeto método é criado consultando um objeto função de uma classe, o seu atribto `im_self` é `None` e o diz-se que ele é um método "desligado". Quando é criado através de uma consulta a um objeto função de uma classe através de uma de suas instâncias, seu atributo `im_self` é a instância e diz-se que é um método "ligado". Em qualquer um dos casos, o atributo `im_class` é a classe de onde a consulta ocorre e o atributo `im_func` é o objeto função original.

Quando um objeto método é criado consultando outro objeto método de uma classe ou instância, o comportamente é idêntico ao de um objeto função, exceto que o atributo `im_func` da nova instância não é o objeto original e sim o seu atributo `im_func`.

Quando um objeto método é criado consultando um método da classe, de uma classe ou instância, seu atributo `im_self` é a própria classe (o mesmo que o atributo `im_class`), e seu atributo `im_func` é o objeto função sob o método da classe.

Quando um objeto método desligado é chamado, a função subjacente (`im_func`) é chamada. O primeiro argumento deve ser uma instância da classe apropriada im_class) ou de uma classe derivada.

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

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

Note que a transformação de um objeto função para um método (ligado ou desligado) acontece cada vez que o atributo é recuperado da classe ou da instância. Em alguns casos, uma boa otimização é atribuir o atributo a uma variável local e chamar aquela variável. Note ainda que esta transformação ocorre apenas para funções definidas pelo usuário; outros objetos executáveis (e também não-executáveis) são recuperados sem transformação. É importante notar que funções definidas pelo usuário que são atributos de uma instância de uma classe não são convertidas para métodos ligados; isso ocorre *apenas* quando a função é um atributo de uma classe.

Funções geradoras Uma função ou método que usa o comando `yield` (veja a seção 6.8, "O comando `yield` ") é chamado de função geradora *generator function.* Ao ser chamado, uma função desse tipo sempre retorna um objeto iterador, que pode ser usado para executar o corpo da função; ao chamar o método `next()` do iterador, a função será executada até retornar um valor através do comando `yield`. Quando a função executa o comando `return`, ou chega no fim, uma exceção `StopIteration` é levantada e o iterador terá alcançado o fim do conjunto de valores a ser retornado.

Funções internas (*built-in*) Uma função interna é um *wrapper* em torno de uma função em C. Exemplos de funções internas são `len()` e `math.sn()` (`math` é um módulo interno padrão). O número e tipo dos argumentos são determinados pela função em C. Atributos somente-leitura especiais: `__doc__` é a string de documentação da função, ou `None`, caso esteja indisponível; `__name__` é o nome da função; `__self__` tem o valor `None` (mas veja o próximo item); `__module__` é o nome do módulo em que a função foi definida, ou `None` se indisponível.

Métodos internos (*built-in*) Estes são realmente quase a mesma coisa de uma função interna, exceto que contém um objeto passado para a função em C como um argumento extra implícito. Um exemplo de

método interno é *alist*`.append()`, assumindo que *alist* é uma lista. Neste caso, o atributo especial somente-leitura `__self__` aponta para o objeto *list*.

Tipos Tipos, ou classes *new-style* são executáveis. Esses objetos normalmente funcionam como fábricas de novas instâncias, mas existem variações possíveis de tipos que sobrescrevem o método `__new__()`. Os argumentos da chamada são passados para o método `__new__()` e, normalmente, passados em seguida para o método `__init__()` para a inicialização da instância.

Classes tradicionais Estas classes são descritas adiante. Quando um objeto class é chamado, uma nova instância dessa classe (também descritas adiante) é criada e retornada. Isso implica em uma chamada do método `__init__()`, se ele existir, com quaisquer argumentos passados na chamada. Se não houver um método `__init__()`, a classe deve ser chamada sem argumentos.

Instâncias Instâncias são descritas logo adiante. Instâncias são executáveis somente quando a classe tem um método `__call__`; `x(argumentos)` é um atalho para `x.__call__(argumentos)`.

Módulos Módulos são importados através do comando `import` (veja section 6.12, "O comando `import`").

Um objeto módulo tem um *namespace* implementado através de um dicionário (é esse o dicionário referenciado pelo atributo func_globals de funções definidas no módulo). Referências a atributos são traduzidas em procuras a esse dicionário, por exemplo: `m.x` equivale à `m.__dict__["x"]`. Um objeto módulo não contem o objeto código usado para sua inicialização (já que não é mais necessário depois dela).

Atribuições atualizam o dicionário do *namespace* do módulo, por exemplo., '`m.x = 1`' equivale a '`m.__dict__["x"] = 1`'.

Atributos especiais somente para leitura: `__dict__` é o *namespace* do módulo, representado por um dicionário.

Atributos pré-definidos: `__name__` é o nome do módulo; `__doc__` é a string de documentação do módulo, ou `None` se indisponível; `__file__` é o caminho do arquivo do qual o módulo foi carregado, se ele foi carregado de um arquivo. O atributo `__file__` não está presente para módulos em C que são ligados estaticamente ao interpretador; para módulos de extensões carregados dinamicamente de uma biblioteca compartilhada, é o caminho do arquivo da biblioteca.

Classes Classes são objetos criados por definições de classe (veja section7.6, "Definições de classe"). Uma classe tem um *namespace* implementado através de um objeto dicionário. Referências a atributos da classe são traduzidas em procuras a este dicionário, por exemplo: '`C.x`' é equivalente à '`C.__dict__["x"]`'. Quando o nome do atributo não é encontrado, a busca continua nas classes base. A busca é realizada da esquerda para a direita, na ordem de ocorrência na lista de classes base.

Quando uma referência de um atributo da classe (digamos, uma classe `C`) irá retornar uma função ou um método desligado cuja classe associada é `C` ou uma de suas classes base, ele é transformado em um método desligado cujo atributo `im_class` é `C`. Quando ele for retornar um método estático, ele é transformado em um objeto "embrulhado" pelo objeto original. Veja section 3.3.2 para outros exemplos de como atributos recuperados de uma classe podem diferir daqueles contidos em seu `__dict__`.

Atribuições aos atributos de uma classe alteram sempre o seu próprio dicionário, nunca o de uma classe base.

Um objeto classe pode ser executado (veja acima) para criar e retornar uma instância da classe (veja logo abaixo).

Atributos especiais: `__name__` é o nome da classe; `__module__` é o nome do módulo no qual a classe foi definida; `__dict__` é o dicionário contendo o *namespace* da classe; `__bases__` é uma tupla (talvez vazia, ou contendo um único elemento) contendo as classes base, na ordem de sua ocorrência na definição da classe; `__doc__` é a string de documentação da classe ou None quando indefinida;

Instâncias de classes Uma instância é criada executando um objeto classe (veja acima). Elas têm um *namespace* implementado como um dicionário que é o primeiro lugar em que são procuradas referências a atributos. Quando um atributo não é encontrado lá, e a classe da instância tem um atributo com aquele nome, a busca continua entre os atributos da classe. Se for encontrado um atributo na classe que seja um objeto função ou um método cuja classe associada seja a seja a classe da instância que iniciou a busca ouuma de suas classes base, este método ou função é então transformado em um objeto método ligado, cujos atributos `im_class` e `im_self` são respectivamente `C` e a própria instância. Métodos estáticos e métodos de classe são também tranformados, como se tivessem sido recuperados da mesma classe `C`; veja logo acima em "Classes". Consulte section **??** para uma outra forma na qual atributos de uma classe recuperados através

de uma instância podem diferir do objeto que esteja realmente armazenado no `__dict__` da classe. Se nenhum atributo da classe for encontrado e ela tem um método `__getattr__()`, então ele é executado para satisfazer a procura.

Ao atribuir ou apagar atributos, o dicionário da instância é atualizado, nunca o da classe. Se a classe tem um método `__setattr__()` ou `__delattr__()`, ele é chamado ao invés de atualizar o dicionário diretamente.

Instâncias podem *fingir* ser números, sequências ou mapeamentos se elas tiverem algums métodos com certos nomes especiais. Veja a seção 3.3, "Special method names."

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

Arquivos Um objeto *file* representa um arquivo aberto. Estes objetos são criados pela função interna `open()` e também por `os.popen()`, `os.fdopen()`, e os métodos `makefile()` de objetos *socket*. Os objetos `sys.stdin`, `sys.stdout` e `sys.stderr` são inicializados para arquivos, correspondendo à entrada e saída de dados padrão do interpretador, e da saída de erros. Veja em *Python Library Reference* para a documentação completa desses objetos.

Tipos internos Alguns tipos usados internamente pelo interpretador são acessíveis ao usuário. Suas definições podem mudar em futuras versões, mas são mencionadas aqui para manter um registro mais completo.

Código Objeto Códigos objetos *[code objects]* contêm os bytes compilados do código executável de Python, ou *bytecodes*. A diferença entre um código objeto para uma função objeto é que a função objeto contém uma referência explícita para o seu contexto global (o módulo em que ela foi definida), enquanto que um código objeto não contém qualquer contexto; além disso, os valores padrão dos argumentos são armazenados no objeto função, não no objeto código (porque estes representam valores calculados durante a execução). Ao contrário de objetos função, código é imutável e não contém quaisquer referências (diretas ou indiretas) para objetos mutáveis.

Atributos especiais somente para leitura: `co_name` contém o nome da função; `co_argcount` é o número de argumentos posicionais (incluindo argumentos com valores padrão); co_nlocals é o número de variáveis locais usadas pela função (incluindo argumentos); `co_varnames` é uma tupla contendo os nomes das variáveis locais (começando pelos nomes dos argumentos); `co_cellvars` é uma tupla contendo os nomes de variáveis locais que são referenciadas por funções aninhadas; `co_freevars` é uma tupla contendo os nomes de variáveis livres; `co_code` é uma string representando a sequência de instruções em *bytecode*; `co_consts` é uma tupla contendo os literais usados pelo *bytecode*; `co_names` é uma tupla contendo os nomes usados pelo *bytecode*; `co_filename` é o nome do arquivo de onde o *bytecode* foi compilado; `co_firstlineno` é o número da primeira linha da função; `co_lnotab` é uma string contendo o mapeamento dos deslocamentos dos *bytecodes* para números de linhas (para maiores detalhes procurar o código fonte do interpratador); `co_stacksize` é o tamanho exigido da pilha (incluindo as variáveis locais); `co_flags` é uma codificação inteira de *flags* para o interpretador.

Os seguintes bits de *flag* são definidos como `co_flags`: o bit `0x04` é definido se a função usa a sintaxe '`*arguments`' para aceitar um número arbitrário de posiçõesç o bit `0x08` é definido se a função usa a sintaxe '`**keywords`' para aceitar argumentos chaves-valores; o bit `0x20` é definido se a função é um gerador.

Futuras declarações de características ('`from __future__ import division`') também usam bits nas `co_flags` para indicar se um código-objeto foi compilado com uma característica particular habilitada: o bit `0x2000` é definido se a função foi compilada com divisões futuras habilitadas; os bits `0x10` e `0x1000` foram utilizados em versões anteriores de Python.

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

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

Objetos quadro [*frame*] Objetos quadro representam execução de quadros. Eles podem ocorrer em objetos [*traceback*] (veja abaixo).

Atributos especiais somente de leitura: `f_back` é anterior ao [*stack frame*] (próximo ao chamador), ou `None` se estiver abaixo na [*stack frame*]; `f_code` é o código objeto que está sendo executado neste quadro; `f_locals` é o diretório utilizado para pesquisa de variáveis locais; `f_globals` é utilizado para variáveis globais; `f_builtins` é uma flag indicando se a função está executando em

modo de execução restrita; `f_lasti` dá a instrução precisa (isto é um índice para dentro da string de [*bytecode*] do código objeto.

Atributos especiais de escrita: `f_trace`, se não for `None`, é uma função chamada no início de cada linha do código fonte (isto é usado pelo depurador); `f_exc_type`, `f_exc_value`, `f_exc_traceback` representa a última exceção levantada no quadro pai deste, se ao menos uma outra exceção qualquer já foi levantada no quadro atual (em todos os outros casos eles contêm None); `f_lineno` é o número atual de linhas do quadro — escrever para este de dentro de uma *trace* salta para as linhas dadas (somente para quadros acima). Um depurador pode implementar um comando Jump (como Set Next Statment) por escrever para f_lineno.

Objetos *traceback* Objetos *traceback* representam um rastro na pilha de exceção. Um objeto *traceback* é criado quando uma exceção ocorre. Quando a busca por um operador de exceção desenrola a pilha de execução, em cada nível um objeto *traceback* é inserido na frente do atual *traceback*. Quando um operador de exceção entra, o rastro na pilha é colocado à disposição do programa. (Consulte section 7.4, "O comando `try`.")

Ele está acessível como `sys.exc_traceback`, e também como o terceiro item do retorno da tupla por `sys.exc_info()`. Recentemente é preferido a interface, desde que ele funcione corretamente quando o programa está utilizando múltiplas *threads*.

Quando o programa não contém operador adequado, o rastro da pilha é escrito (bem formatado) para uma mensagem de erro padrão; se o interpretador está interativo, ele é também colocado à disposição para o usuário como

`sys.last_traceback`.

Atributos especiais somente de leitura: `tb_next` é o próximo nível no rastro da pilha (próximo do quadro onde a exceção ocorreu), ou `None` se não existir próximo nível; `tb_frame` aponta para o quadro de execução do nível atual; `tb_lineno` dá o número da linha onde a exceção ocorreu; `tb_lasti` indica a exata instrução. O número da linha e a última instrução no *traceback* pode ser diferente do número do seu quadro objeto se a cláusula a exceção ocorreu no comando `try` sem que ocorra cláusula de exceção ou com uma cláusula de finalização.

Objetos fatias Objetos fatias são usados para representar fatias quando a *extended slice syntax* é usado. Esta é uma fatia usando duas colunas, ou múltiplas fatias ou elipses separadas por vírgulas, por exemplo, `a[i:j:step]`, `a[i:j,k:l]`, ou `a[..., i:j]`. Eles também são criados pela função embutida `slice()`.

Atributos especiais somente de leitura: `start` é o limite mais baixo; `stop` é o limite mais alto; `step` é o valor do passo; cada atributo é `None` se omitido. Estes atributos podem ter qualquer tipo.

Objetos fatias suportam um método:

`indices`(*self, length*)

Este método pega um simples argumento inteiro *length* e computa informação sobre as fatias extendidas que o objeto fatia descreveria se aplicada para uma seqüência de itens *length*. Ele retorna uma tupla de três inteiros; respectivamente estes são os índices *start* e *stop* e *step* ou transpõem o comprimento da fatia. Perdas ou fora de limites de índices são manipulados de uma maneira consistente com fatias regulares. Novidades na versão New in version 2.3.

Objetos de métodos estáticos Objetos de métodos estáticos fornecem uma maneira de derrotar a transformação da função dos objetos para métodos de objetos descritos acima. Um objeto de método estático é um empacotador em torno de qualquer outro objeto, normalmente um método de objeto definido pelo usuário. Quando um objeto de método estático é retirado de uma classe ou uma instância de classe, o valor realmente retornado do objeto é o objeto empacotado, que não é objeto de qualquer outra transformação. Objetos de métodos estáticos não são chamados por si próprios, embora os objetos sejam normalmente embrulhados. Objetos de métodos estáticos são criados pelo construtor `staticmethod()` embutido.

Objetos de métodos de classe Um objeto de método de classe, como um objeto de método estático, é um empacotador em torno de outro objeto que altera a maneira em que este objeto é retirado das classes e instâncias de classes. O comportamento dos objetos dos métodos das classes acima tal como retirada é descrita abaixo, em "Métodos definidos pelo usuário". Objetos de métodos de classes são criados pelo construtor embutido `classmethod()`.

3.3 Nomes de métodos especiais

Uma classe pode implementar certas operações que são chamadas por sintaxe especial (tais como operações aritméticas ou acesso a posições e fatiamento em contêiners) pela definição de métodos com nomes especiais. Esta é a maneira de Python para *sobrecarga de operador*, permitindo às classes que definam o 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 equivalente ao `x.__getitem__(i)`. Exceto onde mencionado, tentar executar uma operação lança uma exceção quando nenhum método apropriado é definido.

Quando a implementação de uma classe emula qualquer tipo embutido, é importante que a emulação somente seja implementada para o grau em que ela faz sentido para o objeto que esteja sendo modelado. Por exemplo, algumas seqüências podem funcionar bem com a retirada de elementos individuais, mas extrair uma fatia pode não fazer sentido. (Um exemplo disto é a interface `NodeList` no Documento de Modelo de Objetos da W3C).

3.3.1 Personalização básica

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

Called to create a new instance of class *cls*. `__new__()` is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of `__new__()` should be the new object instance (usually an instance of *cls*).

Typical implementations create a new instance of the class by invoking the superclass's `__new__()` method using '`super(`*currentclass*`, `*cls*`).__new__(`*cls*`[, ...])`' with appropriate arguments and then modifying the newly-created instance as necessary before returning it.

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

If `__new__()` does not return an instance of *cls*, then the new instance's `__init__()` method will not be invoked.

`__new__()` is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation.

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

Chamado quando a instância é criada. Os argumentos são aqueles passados para a expressão que chama o construtor da classe. Se uma classe base possui um método `__init__()`, o método `__init__()` da classe derivada, se existir, deve explicitamente chamá-lo para garantir uma correta inicialização da parte da instância que está na classe base; por exemplo: '`BaseClass.__init__(`*self*`, [`*args*`...])`'. Como uma característica especial dos construtores, estes não devem retornar nenhum valor; isto irá fazer com que uma `TypeError` seja lançada em tempo de execução.

`__del__`(*self*)

Chamado quando uma instância está para ser destruída. Também conhecido como destrutor. Se uma classe base possui um método `__del__()`, o método `__del__()` da classe derivada, se existir, deve explicitamente chamá-lo para garantir uma correta deleção da parte da instância que está na classe base. Note que é possível (mas não recomendado!) que o método `__del__()` adie a destruição da instância criando uma nova referência para ela. O método pode ser chamado em outro momento quando esta nova referência é deletada. Não é garantido que métodos `__del__()` sejam chamados para objetos que ainda existam quando o interpretador finaliza.

Note: '`del x`' não chama diretamente `x.__del__()` — a expressão decrementa a contagem de referências para `x` em um, e `__del__()` somente será chamado quando o contador de referências de `x`'s chegar a zero. Algumas situações comuns para evitar que o contador de referências de um objeto chegue a zero incluem: referências circulares entre objetos (ex., uma lista duplamente encadeada ou uma estrutura de dados em árvore com ponteiros pai e filho); uma referência para o objeto na pilha de uma função que captura uma excessão (o *traceback* armazenado em `sys.exc_traceback` mantém a pilha viva); ou uma referência para o objeto em uma pilha que lançou uma excessão não capturada no modo interativo (o *traceback* armazenado em mantém a pilha viva). A primeira situação só pode ser remediada através da

quebra explícita dos círculos; as duas últimas situações podem ser resolvidas armezenando-se `None` em `sys.exc_traceback` ou `sys.last_traceback`. Referências circulares que forem lixo são detectadas quando a opção de detector de ciclos estiver habilitada (o padrão), mas apenas podem ser limpas se não existir nenhum método `__del__()` em nível do Python envolvido. Veja a documentação do módulo `gc` para obter mais informações sobre como métodos `__del__()` são tratados pelo detector de ciclos, particularmente na descrição do valor `garbage`.

Devido a circunstâncias precárias sob as quais os métodos `__del__()` são invocados, excessões que ocorrerem durante sua execução são ignoradas, e ao invés um *(*warning) (aviso) é impresso em `sys.stderr`. Além disso, quando `__del__()` é invocado em resposta a um módulo sendo deletado (ex. quando a execução do programa termina), outros objetos globais referenciados pelo método `__del__()` já podem ter sido deletados. Por essa razão, métodos `__del__()` devem fazer mínimo necessário para manter invariantes externas. A partir da versão 1.5, Python garante que objetos globais cujo nome começa com um simples *underscore* (_) são deletados de seus módulos antes que outros objetos globais forem deletados; se não existir mais nenhuma referência para tais objetos globais, ajuda a garantir que módulos importados ainda estejam disponíveis no momento que o método `__del__()` for chamado.

`__repr__`(*self*)
Chamado pela função interna `repr()` e por conversões em string (aspas reversas) para computar a representação "oficial" em string de um objeto. Se for possível, deve ser 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 na forma '*<...alguma descriç˜ao útil...>*' deve ser retornada. O valor retornado deve ser um objeto string. Se uma classe define `__repr__()` mas não `__str__()`, então `__repr__()` é também usado quando uma representação "informal" em string de uma instância da classe for requerida.

Isto é tipicamente usado para depuração, então é importante que a representação seja ricamente informativa e não ambígua.

`__str__`(*self*)
Chamado pela função interna `str()` e pela expressão `print` para computar a representação "informal" em string de um objeto. Se difere de `__repr__()` no que esta não precisa ser uma expressão Python válida: ao invés, uma representação mais conveniente e concisa deve ser usada. O valor de retorno deve ser um objeto string.

`__lt__`(*self, other*)
`__le__`(*self, other*)
`__eq__`(*self, other*)
`__ne__`(*self, other*)
`__gt__`(*self, other*)
`__ge__`(*self, other*)
New in version 2.1. Estes são amplamente chamados de método para "comparação rica", e são chamados pelos operadores de comparação em preferência ao método `__cmp__()` abaixo. A correspondência entre os símbolos de operadores e os nomes dos métodos são as seguintes: *x*<*y* chama *x*.`__lt__`(*y*), *x*<=*y* chama *x*.`__le__`(*y*), *x*==*y* chama *x*.`__eq__`(*y*), *x*!=*y* e *x*<>*y* chamam *x*.`__ne__`(*y*), *x*>*y* chama *x*.`__gt__`(*y*), e *x*>=*y* chama *x*.`__ge__`(*y*). Estes métodos podem retornar qualquer valor, mas se o operador de comparação é usado em um contexto Booleano, o valor de retorno deve ser interpretável como um valor Booleano, se não `TypeError` será lançada. Por convenção, `False` para falso e `True` para true.

Não há nenhum relacionamento implícito entre os operadores de comparação. A validade de *x*==*y* não implica que *x*!=*y* é falso. Similarmente, quando se define `__eq__()`, `__ne__` também deve ser definido, de modo que os operadores se comportem como esperado.

Não existe nenhuma versão especificamente reflexiva (argumento de troca) desses métodos (para serem usadas quando o argumento da esquerda não suporta a operação, mas o da direita suporta); ao invés, `__lt__()` e `__gt__()` são reflexivos um do outro, `__le__()` e `__ge__()` são reflexivos um do outro, e `__eq__()` e `__ne__()` são suas próprias reflexões.

Argumentos em comparações ricas nunca sofrem coerção. Um método de comparação rica deve retornar `NotImplemented` se não suportar a operação para um dado par de argumentos.

`__cmp__`(*self, other*)
Chamado por operadores de comparação se comparações ricas (veja acima) não estiverem definidas. Deve retornar um valor negativo se `self < other`, zero se `self == other`, um inteiro positivo se `self < other`. Se nenhuma das operações `__cmp__()`, `__eq__()` ou `__ne__()` estiverem definidas,

instâncias da classe são comparadas pela identidade do objeto (“endereço”). Veja também a descrição de `__hash__()` para algumas importantes notas sobre a criação de objetos que suportam operações personalizadas e são usáveis como chaves de dicionários. (Nota: a restrição onde excessões não são propagadas por `__cmp__()` foram removidas desde Python 1.5.)

`__rcmp__`(*self, other*)
Changed in version 2.1: Não mais suportado.

`__hash__`(*self*)
Chamado em operações com o objeto chave usado em dicionário , e pela função interna `hash()`. Deve retornar um inteiro de 32 bits usável como um valor hash em operações com dicionários. A única propriedade requerida é que objetos que possuem igualdade de comparação devem ter o mesmo valor *hash*; é recomendado que de algum modo sejam colocados juntos (ex., usando ou exclusivo), os valores *hash* para componentes que também são objetos comparáveis. Se uma classe não define um método `__cmp__()` não deve definir uma operação `__hash__()` também; se ele define `__cmp__()` ou `__eq__()` mas não `__hash__()`, suas instâncias não serão usáveis como chaves de dicionários. Se uma classe define objetos mutáveis e implementa o método `__cmp__()` ou `__eq__()`, ela não deve implementar `__hash__()`, uma vez que a implementação de dicionário requer que o valor *hash* seja imutável (se o valor *hash* do objeto mudar, estará em um uso errado do *hash*).

`__nonzero__`(*self*)
Chamado para implementar o valor de teste de verdade, e a operação interna `bool()`; deve retornar `False` ou `True`, ou seus valores inteiros `0` or `1`. Quando este método não está definido, `__len__()` é chamado, se estiver definido (veja abaixo). Se uma classe não define `__len__()` nem `__nonzero__()`, todas as suas instâncias são consideradas verdadeiras.

`__unicode__`(*self*)
Chamado para implementar a função interna `unicode()`; deve retornar um objeto Unicode. Quando este método não está definido, conversões em string serão tentadas, e o resultado da conversão em string é convertido em Unicode usando a codificação padrão do sistema.

3.3.2 Personalizando o acesso a atributos

Os métodos seguintes podem ser definidos para customizar o modo de acesso a atributos (uso de, atribuição para, ou deleção de `x.name`) em instâncias de classes.

`__getattr__`(*self, name*)
Chamado quando a busca por um atributo não o encontra nos lugares usuais (i.e., não é um atributo da instância nem foi encontrado na árvore da classe em `self`). `name` é o nome do atributo. Este método deve retornar o valor do atributo (computado) ou lançar uma excessão `AttributeError`.

Note que se o atributo é encontrado através do mecanismo normal, `__getattr__()` não é chamado. (Essa é uma intencional assimetria entre `__getattr__()` e `__setattr__()`.) Isso é feito em ambos por questões de eficiência e porque de outro modo `__setattr__()` não teria como acessar outros atributos da instância. Note que pelo menos para variáveis de instância, você pode “falsificar” totalmente o controle não inserindo quaisquer valores no dicionário de atributos da instância (mas ao invés inserí-los em outro objeto). Veja o método `__getattribute__()` abaixo para um modo de atualmente obter controle total em classes *new-style*.

`__setattr__`(*self, name, value*)
Chamado quando ocorre uma atribuição a algum atributo. Este método é chamado ao invés do mecanismo normal (i.e. armazenar o valor no dicionário da instância). *name* é o nome do atributo, *value* é o valor a ser atribuído a ele.

Se `__setattr__()` pretende atribuir um valor a um atributo de uma instância, ele não deve simplesmente executar ‘`self.`*name* `= value`’ — isso causaria uma chamada recursiva a si mesmo. Ao invés, deve inserir o valor no dicionário de atributos da instância, ex., ‘`self.__dict__[`*name*`] = value`’. Para classes *new-style*, ao invés de acessar o dicionário da instância, deve chamar o método da classe base com o mesmo nome, por exemplo, ‘`object.__setattr__(self, name, value)`’.

`__delattr__`(*self, name*)
Similar a `__setattr__()`, mas para a deleção de um atributo ao invés de atribuição. Este só deve ser

implementado se '`del obj.`*name*' é significativo para o objeto.

Mais sobre acesso a atributos para classes *new-style*

Os métodos seguintes só se aplicam para classes *new-style*.

`__getattribute__`(*self, name*)
Chamado incondicionalmente para implementar acesso a atributos para instâncias da classe. Se a classe também define `__getattr__`, ele não será chamado a menos que `__getattribute__` chame-o explicitamente ou levante uma exceção `AttributeError`. Este método deve retornar o valor (computado) do atributo ou lançar um excessão `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 se necessário, por exemplo, '`object.__getattribute__(self, name)`'.

Implementando Descritores

Os métodos seguintes apenas se aplicam quando a instância da classe contendo o método (a chamada classe *descriptor (descritora)*) aparece no dicionário de outra classe *new-style*, conhecida como classe *proprietária*. Nos exemplos abaixo, "o atributo" refere-se ao atributo cujo nome é a chave da propriedade na classe proprietária' `__dict__`. Descritores só podem ser implementados ocmo classes *new-style*.

`__get__`(*self, instance, owner*)
Chamado para ler o atributo da classe proprietária (acesso ao atributo da classe) ou de uma instância dessa classe (acesso ao atributo da instância). *owner* é sempre a classe proprietária, enquanto *instance* é a instância através da qual o atributo está sendo acessado, ou `None` quando o atributo é acessado através de *owner*. Este método deve retornar o valor (computado) do atributo ou lançar uma excessão `AttributeError`.

`__set__`(*self, instance, value*)
Chamado para atribuir um novo valor *value* ao atributo em uma instância *instance* da classe proprietária.

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

Invocando Descritores

Em geral, um descritor é um atributo de um objeto com "comportamento de ligação", no qual o acesso a atributos foi sobrescrito por métodos no protocolo de descritor: `__get__()`, `__set__()`, e `__delete__()`. Se qualquer destes métodos estão definidos para um objeto, ele é chamado de descritor.

O comportamento padrão para acessar atributos é ler, atribuir, ou deletar o atributo do dicionário de um objeto. Para uma instância, `a.x` possui uma corrente de localização começando com `a.__dict__['x']`, depois `type(a).__dict__['x']`, e continuando através das classes base de `type(a)`, excluindo-se as metaclasses.

Entretanto, se o valor procurado é um objeto que define um dos métodos descritores, então Python pode sobrescrever o comportamento padrão e invocar ao invés o método descritor. Onde isto irá ocorrer na corrente de precedncia depende de onde os métodos descritores estão definidos e como eles são chamados. Note que descritores são apenas invocados para objetos ou classes *new-style* (aquelas que extendem `object()` ou `type()`).

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

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

Ligação de Instância
: Se ligada a uma instância de um objeto *new-style*, `a.x` é transformado na chamada: `type(a).__dict__['x'].__get__(a, type(a))`.

Ligação de Classe
: Se ligada a uma classe *new-style*, `A.x` é transformada na chamada: `A.__dict__['x'].__get__(None, A)`.

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

Para ligações de instância, a precedência de invocação de descritor depende de quais métodos descritores estão definidos. Descritores de dados definem tanto `__get__()` quanto `__set__()`. Descritores que não representam dados (*non-data*) possuem apenas o método `__get__()`. Descritores de dados sempre sobrescrevem uma redefinição em um dicionário de instância. Em constraste, descritores que não representam dados podem ser sobrescritos pelas instâncias.

Métodos Python (incluindo `staticmethod()` e `classmethod()`) são implementados como descritores *non-data*. Deste modo, instâncias podem redefinir e sobrescrever métodos. Isto permite a instâncias individuais adiquirirem comportamentos que as diferem de outras instâncias da mesma classe.

A função `property()` é implementadas como um descritor de dados. Deste modo, instãncias não podem sobrescrever o comportamento de uma propriedade.

__slots__

Por padrão, instâncias de ambas as classes antigas e *new-style* possuem um dicionário para armazenamento de atributos. Isto disperdiça espaço para objetos contendo muito poucas variáveis de instância. O consumo de espaço pode se tornar prejudicial ao se criar um grande número de instâncias.

O padrão pode ser sobrescrito pela definição de *__slots__* em classes *new-style*. A declaração de *__slots__* toma uma sequência de variáveis de instância e reserva apenas espaço suficiente em cada instância para guardar um valor para cada variável. Espaço é economizado porque *__dict__* não é criado para cada instância.

`__slots__`
: A esta variável de classe pode ser atribuída uma string, um interável, ou uma sequência de strings com nomes variáveis usados pelas instâncias. Se definido em uma classe *new-style*, *__slots__* reserva espaço para as variáveis declaradas e previne a automática criação de *__dict__* e *__weakref__* para cada instância. New in version 2.2.

Notas ao usar *__slots__*

- Se uma variável *__dict__*, instâncias não podem atribuir novas variáveis não listadas na definição de *__slots__*. Tentativas de atribuir à uma variável cujo nome não tenha sido listado lançam `AttributeError`. Se a atribuição dinâmica de novas variáveis é desejada, então adicione `'__dict__'` para a sequência de string na declaração de *__slots__*. Changed in version 2.3: Antigamente, adicionando `'__dict__'` à declaração de *__slots__* não habilitaria a atribuição de novos atributos não especificamente listados na sequência de nomes de variáveis de instância.
- Sem uma variável *__weakref__* para cada instância, classes que definem *__slots__* não suportam referências fracas para suas instãncias. Se o suporte a referência fraca é necessário, então adicione `'__weakref__'` à sequência de string na declaração de *__slots__*. Changed in version 2.3: Antigamentem, adicionando `'__weakref__'` à declaração de *__slots__* não habilitaria o suporte para referências fracas.
- *__slots__* é implementado em nível de classe pela criação de descritores (3.3.2) para cada nome de variável. Como resultado, atributos de classe não podem ser usados para atribuir valores padrão para variáveis de instância definidos por *__slots__*; de outro modo, o atributo da classe sobrescreveria a atribuição do descritor.
- Se uma classe define um slot também definido na classe base, a variável de instância definida pelo slot da classe base é inacessível (exceto retornando seu descritor diretamente da classe base). Isto rende um comportamento indefinido do programa. No futuro, uma verificação deve ser adicionada para prevenir isso.
- A ação da declaração de *__slots__* é limitada à classe onde ele está definido. Como resultado, subclasses terão um *__dict__* a menos que elas também definam *__slots__*.
- *__slots__* não funcionam para classes derivadas de tipos internos de “tamanho variável” como `long`, `str` and `tuple`.
- Qualquer iterável não string pode ser atribuído à *__slots__*. Mapas também podem ser usados; entretando, no futuro, comportamento especial pode ser adicionado aos valores correspondentes a cada chave.

3.3.3 Personalizando a criação de classes

Por padrão, classes *new-style* são construídas usando `type()`. Uma definição de classe é lida dentro de um *namespace* separado e o valor do nome da classe é anexado ao resultado de `type(name, bases, dict)`.

Quando a definição de uma classe é lida, se *__metaclass__* está definida então a classe executável (*callable*) atribuída a eke será chamada ao invés de `type()`. Isto permite que classes e funções sejam escritas para monitorar ou alterar o processo de criação da classe:

- Modificando o dicionário da classe previamente à criação da classe.
- Retornando uma instância de outra classe – essencialmente realizando a regra de uma uma função fábrica.

`__metaclass__`
Esta variável pode ser qualquer classe executável (*callable*) que aceite os argumentos `name`, `bases`, e `dict`. Na criação da classe, o executável é usado ao invés do interno `type()`. New in version 2.2.

A metaclasse apropriadas é determinada pelas seguintes regras de precedência:

- Se `dict['__metaclass__']` existe, é usado.
- Caso contrário, se existe pelo menos uma classe base, sua metaclasse é usada (procura-se por um atributo *__class__* primeiro e se não encontrado, usa seu tipo).
- Caso contrário, se uma variável global chamada __metaclass__ existe, pe usada.
- Caso contrário, o estilo antigo, com metaclasse clássica (types.ClassType) é usado.

O uso potencial de metaclasses não possui limites. Algumas idéias que tem sido exploradas incluem *logging*, verificação de interface, delegação automática, criação de propriedades automática, proxies, *frameworks*, e *locking*/sincronização de recursos automático.

3.3.4 Emulando objetos executáveis (*callable*)

`__call__`(*self*[, *args...*])
Chamado quando a instância é "executada" como uma função; se este método está definido, *x*`(arg1, arg2, ...)` é um atalho para *x*`.__call__(arg1, arg2, ...)`.

3.3.5 Emulando tipos contêiner

Os métodos seguintes podem ser definidos para implementar objetos contêiner. Contêineres em geral são sequências (como listas e tuplas) ou mapas (como dicionários), mas podem representar outros contêiners do mesmo modo. O primeiro conjunto de métodos e usado tanto para emular uma sequência quando para emular um mapa; a diferença é que para uma sequência, as chaves permitidas devem ser inteiros *k*, para `0 <=` *k* `<` *N* onde *N* é o tamanho da sequência, ou objetos fatia, que definem uma faixa de itens. (Para compatibilidade com versões anteriores, o método `__getslice__()` (veja abaixo) também de ser definido para lidar com fatias simples, mas não extendidas. Também é recomendado que mapas provenham os métodos `keys()`, `values()`, `items()`, `has_key()`, `get()`, `clear()`, `setdefault()`, `iterkeys()`, `itervalues()`, `iteritems()`, `pop()`, `popitem()`, `copy()`, e `update()` se comportando de modo similar aos dicionários padrão de Python. O módulo `UserDict` provê uma classe `DictMixin` para ajudar na criação daqueles métodos de um conjunto base de `__getitem__()`, `__setitem__()`, `__delitem__()`, e `keys()`. Sequências mutáveis devem prover métodos `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()` e `sort()`, como os objetos lista padrões de Python. Finalmente, tipos sequência devem implementar adição (no sentido de concatenação) e multiplicação (no sentido de repetição) pela definição dos métodos `__add__()`, `__radd__()`, `__iadd__()`, `__mul__()`, `__rmul__()` e `__imul__()` descritos abaixo; eles não devem definir `__coerce__()` ou outros operadores numéricos. É recomendado que ambos os mapas e sequências implementem o método `__contains__()` para permitir o uso eficiente do operador `in`; para mapas, `in` deve ser equivalente a `has_key()`; para sequências, deve buscar através dos valores. Adicionalmente

é recomendado que ambos mapas e sequências implementem o método `__iter__()` para permitir interação eficiente através do contêiner; para mapas, `__iter__()` deve ser o mesmo que `iterkeys()`; para sequências, deve iterar sobre seus valores.

`__len__`(*self*)
Chamado para implementar a função interna `len()`. Além disso, um objeto que não tenha definido um método `__nonzero__()` e cujo método `__len__()` retorne zero será considerado falso em um contexto Booleano.

`__getitem__`(*self, key*)
Chamado para implementar a avaliação de *self*`[`*key*`]`. Para tipos sequência, as chaves aceitas devem ser inteiros e objetos fatia. Note que a interpretação especial de índices negativos (se a classe deseja emular um tipo sequência) deve ser feita no método `__getitem__()`. Se *key* é de um tipo inapropriado, `TypeError` deve ser lançado; se for um valor fora do conjunto de índices da sequência (depois de qualquer interpretação especial de valores negativos), `IndexError` deve ser lançado. Para mapeamentos, se a chave (*key*) estiver faltando (não estiver no conteiner), `KeyError` deve ser levantada. **Note:** `for` loops esperam que `IndexError` seja lançado para índices ilegais para permitir uma detecção adequada do final da sequência.

`__setitem__`(*self, key, value*)
Chamado para implememyar atribuição para *self*`[`*key*`]`. Se aplicam os mesmos comentários que `__getitem__()`. Este método somente deve ser implementado para mapas se os objetos nos valores para as chaves dadas suportam alterações, ou se novas chaves podem ser adicionadas, ou para sequências se elementos podem ser substituídos. As mesmas excessões do método devem ser lançadas para valores impróprios de *key*.

`__delitem__`(*self, key*)
Chamado para implementar deleção de *self*`[`*key*`]`. Se aplicam os mesmos comentários que `__getitem__()`. Este método somente deve ser implementado para mapas se os objetos suportam remoção de chaves, ou para sequências se elementos podem ser removidos dela. As mesmas excessões do método devem ser lançadas para valores impróprios de *key*. `__getitem__()` method.

`__iter__`(*self*)
Este método é chamado quando um iterador é requerido para o contêiner. Este método deve retornar um novo objeto iterador que possa iterar sobre todos os objetos do contêiner. Para mapas, deve iterar sobre as chaves do contêiner, e também deve estar disponível como no método `iterkeys()`.

Objetos iteradores também precisam implementar este método; neste caso é requerido que eles retornem a si próprios. Para mais informações sobre objetos iteradores, veja "Iterator Types" em *Referência da Biblioteca Python*.

Os operadores de teste de existência de um membro (`in` and `not in`) são normalmente implementados como uma iteração através da sequência. Entretanto, objetos contêiner podem suprir o seguinte método especial como uma implementação mais eficiente, que também não requer que o objeto seja uma sequência.

`__contains__`(*self, item*)
Chamado para implementar operadores de teste de existência de um membro. Deve retornar verdadeiro se *item* está em *self*, falso caso contrário. Para objetos mapa, deve considerar as chaves do mapas ao invés dos valores ou dos pares chave-ítem.

3.3.6 Métodos adicionais para emulação de tipos sequência

Os seguintes métodos opcionais podem ser definidos para incrementar a emulação de objetos sequência. Métodos de sequências imutáveis devem no máximo definir `__getslice__()`; sequências mutáveis podem definir todos os três métodos.

`__getslice__`(*self, i, j*)
Deprecated since release 2.0. Suporte de objetos fatia como parâmetros para o método `__getitem__()`.

Chamado para implemenyar a avaliação de *self*`[`*i*`:`*j*`]`. O objeto retornado deve ser do mesmo tipo de *self*. Note que omitindo *i* ou *j* na expressão fatia os valores serão substituídos por zero ou `sys.maxint`, respectivamente. Se índices negativos são usados na fatia, o tamanho da sequência é adicionado àquele índice. Se a instância não implementa o método `__len__()`, `AttributeError` é lençada. Nenhuma

garantia é feita de que índices ajustados deste modo não continuem negativos. Índices maiores do que o tamanho da sequência não são modificados. Se `__getslice__()` não for encontrado, um objeto fatia é criado, e passado para `__getitem__()`.

`__setslice__`(*self, i, j, sequence*)
Chamado para implementar atribuição a *self* [*i*:*j*]. Se aplicam os mesmos comentários de *i* e *j* em `__getslice__()`. Este método está obsoleto. Se `__setslice__()` não for encontrado, ou para fatia extendida na forma *self* [*i*:*j*:*k*], um objeto fatia é criado, e passado para `__setitem__()`, ao invés de chamar `__setslice__()`.

`__delslice__`(*self, i, j*)
Chamado para implementar a deleção de *self* [*i*:*j*]. Se aplicam os mesmos comentários de *i* e *j* em `__getslice__()`. Este método está obsoleto. Se `__delslice__()` não for encontrado, ou para fatia extendida na forma *self* [*i*:*j*:*k*], um objeto fatia é criado, e passado para `__delitem__()`, ao invés de chamar `__delslice__()`.

Note que estes métodos apenas são invocados quando uma única fatia com uma única vírgula é usada, e o método de fatiamento está disponível. Para operações de fatiamento involvendo notação de fatiamento extendida, ou na ausência de métodos de fatiamento, `__getitem__()`, `__setitem__()` ou `__delitem__()` é chamado com um objeto fatia como argumento.

O seguinte exemplo demonstra como fazer com que seu programa ou módulo seja compatível com versões antigas de Python (assumindo que os métodos `__getitem__()`, `__setitem__()` and `__delitem__()` suportem objetos fatia como argumentos):

```
class MyClass:
    ...
    def __getitem__(self, index):
        ...
    def __setitem__(self, index, value):
        ...
    def __delitem__(self, index):
        ...

    if sys.version_info < (2, 0):
        # They won't be defined if version is at least 2.0 final

        def __getslice__(self, i, j):
            return self[max(0, i):max(0, j):]
        def __setslice__(self, i, j, seq):
            self[max(0, i):max(0, j):] = seq
        def __delslice__(self, i, j):
            del self[max(0, i):max(0, j):]
    ...
```

Note as chamadas a `max()`; elas são necessárias para tratar índices negativos antes que os métodos `__*slice__()` sejam chamados. Quando índices negativos são usados, os métodos `__*item__()` os recebem como são fornecidos, mas os métodos `__*slice__()` recebem uma forma "preparada" dos valores dos índices. Para cada valor de índice negativo, o tamanho da sequência é adicionado ao índice antes de chamar o método (o que ainda pode resultar em um índice negativo); este é o tratamento habitual de índices negativos pelas tipos sequência internos, e espera-se que os métodos `__*item__()` procedam deste modo. Entretanto, uma vez que eles já devem ter feito isto, índices negativos não podem ser passados; eles devem ser forçados para os limites da sequência antes de serem passados ao métodos `__*item__()`. Chamando `max(0, i)` covenientemente retorna o valor apropriado.

3.3.7 Emulando tipos numéricos

Os métodos seguintes podem ser definidos para emular objetos numéricos. Métodos correspondentes a operações que não são suportadas pelo tipo particular de número implementado (ex., operações bit a bit para números não inteiros) devem ser deixadas indefinidas.

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

Estes métodos são chamados para implementar as operações aritméticas binárias (`+`, `-`, `*`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). Por exemplo, para avaliar a expressão *x*+*y*, onde *x* é uma instância da classe que possui um método `__add__()`, *x*`.__add__(`*y*`)`. O método `__divmod__()` deve ser equivalente ao uso de `__floordiv__()` e `__mod__()`; ele não deve ser relacionado com `__truediv__()` (descrito abaixo). Note que `__pow__()` deve ser definido para aceitar um terceiro argumento opcional se pretende-se suportar aa versão ternária de `pow()`.

__div__(*self, other*)
__truediv__(*self, other*)

O operador de divisão (`/`) é implementado por estes métodos. O método `__truediv__()` é usado quando `__future__.division` está ativado, caso contrário, `__div__()` é usado. Se somente um destes dois métodos está definido, o objeto não irá suportar divisão no contexto alternativo; `TypeError` será lançada ao invés.

__radd__(*self, other*)
__rsub__(*self, other*)
__rmul__(*self, other*)
__rdiv__(*self, other*)
__rtruediv__(*self, other*)
__rfloordiv__(*self, other*)
__rmod__(*self, other*)
__rdivmod__(*self, other*)
__rpow__(*self, other*)
__rlshift__(*self, other*)
__rrshift__(*self, other*)
__rand__(*self, other*)
__rxor__(*self, other*)
__ror__(*self, other*)

Estes métodos são chamados para implmentar as operações aritméticas binárias (`+`, `-`, `*`, `/`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) com os operadores refletidos (trocados). Estas funções apenas são chamadas se o operador da esquerda não suporta a operação correspondente. Por exemplo, para avaliar a expressão *x*`-`*y*, onde *y* é uma instância da classe que tem um método `__rsub__()`, *y*`.__rsub__(`*x*`)` é chamado. Note que a função ternária `pow()` não tentará chamar `__rpow__()` (as regras de coerção se tornariam muito complicadas).

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

Estes métodos são chamados para implementar as operações aritméticas de aumento (`+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). Estes métodos devem tentar realizar a operação de modo *(*in-place) (modificando *self*) e retornar o resultado (o que poderia ser, mas não necessariamente, *self*). Se um método específico não está definido, a operação de aumento cai nos métodos normais. Por exemplo, para avaliar a expressão *x*`+=`*y*, onde *x* é uma instância da classe que possui um método `__iadd__()`, *x*`.__iadd__(`*y*`)` é chamado. Se *x* é uma instância de uma classe que não define um método `__iadd__()`, *x*`.__add__(`*y*`)` e *y*`.__radd__(`*x*`)` são considerados, como na avaliação de *x*+*y*.

`__neg__`(*self*)
`__pos__`(*self*)
`__abs__`(*self*)
`__invert__`(*self*)
Chamado para implementar as operações aritméticas unárias (`-`, `+`, `abs()` and `~`).

`__complex__`(*self*)
`__int__`(*self*)
`__long__`(*self*)
`__float__`(*self*)
Chamado para implementar as funções internas `complex()`, `int()`, `long()`, and `float()`. Deve retornar um valor do tipo apropriado.

`__oct__`(*self*)
`__hex__`(*self*)
Chamado para implementar as funções internas `oct()` e `hex()`. Deve retornar um valor em string.

`__coerce__`(*self, other*)
Chamado para implementar o "modo misto" de números aritméticos. Deve retornar uma tupla de dois elementos contendo *self* e *other* convertidos a um tipo numérico comum, ou `None` se a conversão é impossível. Quando o tipo comum seria do tipo de `other`, é insuficiente retornar `None`, uma vez que o interpretador também irá perguntar ao outro objeto para tentar a coerção. (mas às vezes, se a implementação do outro tipo não pode ser é útil fazer a conversão para o outro tipo aqui). Retornar o valor `NotImplemented` é equivalente a retornar `None`.

3.3.8 Regras de coerção

Esta seção costumava documentar as regras de coerção. À medida que a linguagem evoluiu, estas regras se tornaram difíceis de documentar com precisão; descrever aquilo que uma determinada versão de uma implementação em particular faz é indesejável. Ao invés disso, aqui segue um guia informal de algumas regras. Em Python 3.0, coerção não será mais suportada.

- Se o operando esquerdo de um operador % é uma string ou um objeto Unicode, nenhuma coerção ocorre. Ao invés disso, é executada a operação de formatação da string.
- Não é mais recomendável definir um operador de coerção. Operações em tipos que não definem coerção passam os argumentos originais para a operação.
- Classes *new-style* (aquelas derivadas de `object`) nunca invocam o método `__coerce__()` em resposta a um operador binário; o único momento em que `__coerce__()` é chamado é quando a função interna `coerce()` é usada.
- Para a maioria das aplicações, um operador que retorna `NotImplemented` é tratado da mesma forma que um que realmente não esteja implementado.
- Logo abaixo, os métodos `__op__()` e `__rop__()` são usados para simbolizar os nomes genéricos de métodos correspondentes a um operador; o método `__iop__()` é usado para o operador correspondente que realiza a operação no lugar (*in-place*), sem criar um novo objeto. Por exemplo, para o operador '`+`', `__add__()` e `__radd__()` são usados para as variáveis esquerda e direita do operador binário, e `__iadd__()` para a variante que realiza a operação no lugar (*in-place*).
- Para objetos *x* e *y*, primeiro tentamos*x*`.__op__(`*y*`)` . Se este não estiver implementado ou retornar `NotImplemented`, tenta-se *y*`.__rop__(`*x*`)`. Se este também não estiver implementado ou retornar `NotImplemented`, uma exceção `TypeError` é levantada. Há uma exceção para isso no próximo item:

- Se o operando esquerdo é uma instância de um tipo interno ou de uma classe *new-style*, e o operando direito é uma instância de uma subclasse apropriada, daquele tipo ou classe, o método `__rop__()` do operando à direita é tentado antes do método `__op__()` do operando à esquerda. Isto é feito para que uma subclasse possa redefinir completamente os operadores binários. De outra forma, o método __op__ do operando esquerdo sempre aceitaria o outro operando: quando uma instância de uma determinada classe é esperada, uma instância de uma subclasse dela é sempre aceitável.

- Quando qualquer um dos operandos define uma regra de coerção, esta regra é chamada antes do método `__op__()` ou `__rop__()` ser chamado, mas não antes disso. Se a coerção retorna um objeto de um tipo diferente do operando cuja coerção foi invocada, parte do processo é refeita usando o objeto novo.

- Quando um operador *in-place* (como '`+=`') é usado, se o operando da esquerda implementa `__iop__()`, ele é invocado sem nenhuma coerção. Quando a operação cai em `__op__()` e/ou `__rop__()`, se aplicam as regras normais de coerção.

- Em *x*+*y*, se *x* é uma sequência que implementa concatenação, x e y são então concatenadas.

- Em *x***y*, se um operador for uma sequência que implementa repetições, e a outra for um inteiro (`int` ou `long`), é invocada a repetição da sequência.

- Comparações ricas (implementadas pelos métodos `__eq__()` dentre outros) nunca usam coerção. Comparações de três vias (implementadas por `__cmp__()`) usam coerção sob as mesmas condições que os outros operadores binários usam.

- Na implementação atual, os tipos numéricos internos `int`, `long` e `float` não usam coerção; o tipo `complex` por outro lado usa. A diferença pode se tornar aparente quando extendemos estes tipos. Todo o tempo, o tipo `complex` pode ser corrigido para evitar a coerção. Todos estes tipos implementam um método `__coerce__()`, para uso da função interna `coerce()`.

CAPÍTULO

FOUR

Modelo de execução

4.1 Nomeando e ligando

Nomes são referências a objetos e são introduzidos por operações de ligação.Cada ocorrência de um nome no texto de um programa se refere à *ligação* estabelecida a ele no bloco de função mais interno que o contém.

Um *bloco* é uma parte do código de um programa que é executada como uma unidade. São blocos: um módulo, o corpo de uma função, a definição de uma classe. Cada comando digitado no interpretador interativo é um bloco. Assim como um arquivo de script (um arquivo passado como a entrada padrão ao interpretador ou especificado na primeira linha na linha de comando do interpretador) e um comando de script (um comando especificado na linha de comando do interpretador usando a opção '**-c**'). Um arquivo lido pela função built-in `execfile()` também é um bloco de código. Um argumento do tipo string passado para a função built-in `eval()` e para o comando `exec` é um bloco de código, a mesma coisa uma expressão lida e executada pela função built-in `input()`.

Um bloco de código é executado em um *quadro de execução*. Um quadro contém informações administrativas (usadas na depuração) e determina como e onde a execução continuará após o término da execução do bloco de código.

O *escopo* define a visibilidade de um nome em um bloco. Se uma variável local é definida em um bloco, seu escopo o inclui. Se a definição ocorre dentro de uma função o escopo se estende a qualquer bloco contido nessa, a não ser que o bloco contido faça uma nova ligação ao nome. O escopo de nomes definidos em um bloco de classe é limitado a esse, ele não se estende aos métodos.

Quando um nome é usado em um bloco de código é utilizada a definição do escopo mais próximo que encapsula o nome. O conjunto de todos escopos visiveis a um bloco de código é o *ambiente* do bloco.

Se um nome é ligado em um bloco ele é uma variável local, caso o nome seja ligado em nivel modular ele é uma variável global (as variáveis ligadas em nivel modular são tanto locais quanto locais.) Se uma variável for usada em um bloco de código em que não foi definida essa é uma *variável livre*.

Quando um nome não é encontrado, uma exceção `NameError` é levantada. Se um nome se referir a uma variável local que não foi ligada, uma exceção `UnboundLocalError` é levantada. `UnboundLocalError` é uma subclasse de `NameError`.

Os seguintes ligam nomes: parâmetros de uma função, comandos `import`, definições de classes e funções (esses ligam o nome da classe ou função no bloco que está sendo definido) e alvos que são identificadores se estiverem acontecendo em uma definição, no cabeçalho de um loop `for`, ou na segunda posição do cabeçalho do comando `except`. O comando `import` escrito desta maneira: "'`from ...import *`"' liga todos os nomes definidos no módulo importado, exceto aqueles que começam com travessão. Essa forma do comando só pode ser utilizada em nível modular.

Seguindo essa idéia, o alvo do comando `del` também pode ser considerado uma ligação (apesar da terminologia mais correta ser desligar o nome). É ilegal desligar um nome que é referenciado pelo escopo externo: o compilador irá reportar uma exceção `SyntaxError`.

Designações ou comandos import devem ocorrer dentro do bloco definido por uma classe ou função ou em nivel modular (o mais alto nivel de um código).

Se uma operação de ligação de nome ocorre em qualquer ponto de um bloco, todo uso desse nome é tratado como uma referência ao bloco atual.. Isso pode levar a erros quando um nome é usado em um bloco antes de ser ligado.

Python não possui declarações de variáveis e permite que a ligação de um nome ocorra em qualquer lugar de um bloco de código. As variáveis locais de um pedaço de código podem ser determinadas apenas olhando as operações de ligação que ocorreram no bloco.

No caso do comando global, todos os usos do nome especificado no comando se referem à ligação feita para aquele nome no namespace de mais alto nível. Nomes são adicionados a esse namespace após a procura por nomes no namespace global (namespace do modulo contendo o bloco de código) e o namespace builtin (namespace do módulo `__builtin__`). O global é buscado primeiro e, se o nome não for encontrado nele, o namespace builtin é buscado. O comando global deve preceder qualquer uso do nome desejado.

O namespace built-in (embutido) associado com a execução de um bloco de código pode ser visualizado usando o nome `__builtins__` em seu namespace global. Esse nome deve ser um dicionário ou um módulo (no caso anterior o dicionário do módulo foi usado). Normalmente o namespace `__builtins__` é um dicionário do módulo embutido `__builtin__` (nota: sem 's'). Se não for, o modo de execução restrita esta sendo usado.

O namespace de um módulo é criado automaticamente da primeira vez que um módulo é importado. O módulo principal de um script é sempre chamado de `__main__`.

A definição de uma classe é um comando executável que usa e define nomes seguindo as regras usuais de formação de um nome. O namespace da definição de uma classe torna-se o dicionário de atributo dessa. Nomes definidos no escopo da classe não são visiveis para os métodos.

4.1.1 Interação com aspectos dinâmicos

Existem vários casos em que comandos Python, quando usados em conjunto com escopos aninhados que possuem variáveis livres são ilegais.

Se houver referências de uma variável em um escopo superior, é ilegal deletar o nome. Caso isso aconteça um erro ocorrerá durante a compilação.

Se uma forma mais radical do import — '`import *`' — é usada em uma função e caso ela contenha ou seja um bloco aninhado com variáveis livres o compilador vai levantar uma exceção SyntaxError.

Se o comando `exec` for usado em uma função contendo variáveis livres ou um bloco aninhado com esse tipo de variável, o compilador levantará uma exceção `SyntaxError`, a não ser que o namespace local seja explicitamente especificado no `exec`. (Em outras palavras, '`exec obj`' seria ilegal, mas '`exec obj in ns`' seria legal.)

As funções `eval()`, `execfile()` e `input()` e o comando `exec` não tem acesso a todos nomes resolvidos no ambiente. Os nomes podem ser resolvidos no namespace global e local do requisitor. As variáveis locais não são resolvidas no namespace mais próximo, mas no namespace global. [1] O comando `exec` e as funções `eval()` e `execfile()` possuem argumentos opcionais para substituir o namespace global e local. Se apenas um namespace for especificado ele é usado como global e local.

4.2 Exceções

Exceções são meios de quebrar o fluxo normal de um código para lidar com erros ou outras condições excepcionais. Uma exceção é *levantada* assim que um erro é detectado, ela pode ser *manipulada* pelo código que a cerca ou pelo código que invocou direta ou indiretamente o bloco onde o erro aconteceu.

O interpretador Python levanta uma exceção quando detecta um erro durante o tempo de execução (como uma divisão por zero). Um programa Python pode levantar uma exceção explicitamente com o comando `raise`. Manipuladores de exceção são especificados com o comando `try` ... `except`. O comando `try` ... `finally` especifica um codigo de limpeza que não lida com a exceção, mas que é sempre executado, ocorrendo ou não uma exceção.

Python usa o modelo de "finalização" na manipulação de erros: um manipulador de exceções pode descobrir o que houve de errado e continuar a execução em um nível mais externo, mas ele não pode consertar a causa do erro ou tentar novamente a operação que causou erro (a não ser que o pedaço de código que causou o erro seja re-executado desde o começo).

[1] Essa limitação acontece porque o código que é executado por essas operações não esta disponível no momento que o modulo é compilado.

Quando uma exceção não é tratada, o interpretador encerra a execução do programa ou retorna para seu loop interativo principal. Em qualquer caso ele imprime na tela o o rastro da pilha de chamada, a não ser quando a exceção é `SystemExit`.

Exceções são identificadas por instâncias de classes. A seleção de uma cláusula except é baseada na identidade do objeto: a cláusula `except` deve se referenciar à mesma classe ou uma classe base da exceção que foi levantada. A instância pode ser recebida pelo manipulador e pode carregar informação adicional sobre a condição excepcional que ocorreu.

Exceções também podem ser identificadas por strings, caso em que a cláusula `except` é selecionada pela identidade do objeto. Um valor arbitrário pode ser passado junto com a string identificadora, que é passada para o manipulador.

Deprecated since release 2.5. Strings não devem ser usadas como exceções em código novo. Elas não funcionarão em uma futura versão de Python. Código antigo deve ser reescrito para usar classes.

Mensagens para exceções não fazem parte da API do Python. Seu conteúdo pode mudar de uma versão para outra sem qualquer aviso e não é confiável para código que poderá ser executado em diversas versões do interpretador

Veja também a descrição do comando `try` em section 7.4 e do comando `raise` em section 6.9.

CAPÍTULO

FIVE

Expressões

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

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

```
name  ::=  othername
```

e não é dada nenhuma semântica, a semântica da forma `name` é a mesma de `othername`.

5.1 Conversões Aritméticas

Quando uma descrição de um operador aritmético abaixo usa a frase “os argumentos numéricos são convertidos para um tipo comum”, os argumentos são convertidos usando as regras de conversão listadas no final do capítulo 3. Se ambos os argumentos forem de tipos numéricos padrão, as seguintes conversões são aplicadas:

- Se um dos argumentos é um número complexo, o outro é convertido para complexo;
- caso contrário, se um dos argumentos é um número de ponto flutuante, o outro é convertido para ponto flutuante;
- caso contrário, se um dos argumentos é um inteiro longo, o outro é convertido para inteiro longo;
- caso contrário, ambos devem se inteiros simples e nenhuma conversão é necessária.

Algumas regras adicionais se aplicam a certos operadores (por exemplo, um argumento do tipo string à esquerda do operador ‘%’). Extensões podem definir suas próprias regras de conversão.

5.2 Átomos

Átomos (*atoms*) são os elementos mais básicos das expressões. Os átomos mais simples são identificadores (*identifiers*) ou literais (*literals*). Expressões delimitadas por aspas invertidas, parênteses, colchetes ou chaves (*enclosures*) também são categorizadas sintaticamente como átomos. A sintaxe dos átomos é:

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

5.2.1 Identificadores (Nomes)

Um identificador na forma de um átomo é um nome. Veja section 4.1 para documentação sobre nomeação e ligação.

Quando o nome é ligado a um objeto, a interpretação do átomo retorna este objeto. Quando um nome não é ligado, a tentativa de avaliá-lo levanta uma exceção `NameError`.

Modificação de nomes privados Quando um identificador que ocorre textualmente na definição de uma classe começa com dois ou mais caracteres sublinhados (*underscore*) e não termina com dois ou mais sublinhados, este é considerado um *nome privado* desta classe. Nomes privados são transformados para uma forma mais longa antes que seu código seja gerado. A transformação insere o nome da classe antes do nome, com os sublinhados iniciais removidos, e um único sublinhado é inserido antes no nome da classe. Por exemplo, o identificador `__spam`, ocorrendo em um classe chamada `Ham`, será transformado em `_Ham__spam`. Essa transformação é independente do contexto sintático no qual o identificador é usado. Se o nome transformado é extremamente longo (maior do que 255 caracteres), pode ocorrer o truncamento definido pela implementação. Se o nome da classe é composto somente por sublinhados, não é feita nenhuma transformação.

5.2.2 Literais

Python suporta strings literais e vários literais numéricos.

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

A interpretação de um literal retorna um objeto do tipo fornecido (string, inteiro, inteiro longo, número de ponto flutuante, número complexo) com o valor fornecido. Este valor pode ser aproximado no caso de literais de ponto flutuante e imaginários (complexos). Veja section 2.4 para mais detalhes.

Todos os literais correspondem a tipos de dados imutáveis e, portanto, a identidade do objeto é menos importante do que o seu valor. Múltiplas interpretações de literais com o mesmo valor (tanto a mesma ocorrência no corpo do programa quanto uma ocorrência diferente) podem resultar no mesmo objeto ou em um objeto diferente com o mesmo valor.

5.2.3 Estruturas entre parênteses

Um estrutura entre parênteses (*parenth_form*) é uma lista opcional de expressões contida entre parênteses:

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

Uma lista de expressões entre parênteses retorna o que quer que a lista de expressões retorne: se a lista contém pelo menos uma vírgula, esta retorna uma tupla; caso contrário, retorna a única expressão que compõe a lista de expressões.

Um par de parênteses vazio retorna uma tupla vazia. Uma vez que as tuplas são imutáveis, as regras para literais se aplicam (isto é, duas ocorrências de uma tupla vazia podem ou não retornar o mesmo objeto).

Note que as tuplas não são formadas pelos parênteses, mas pelo uso do operador vírgula. A exceção é a tupla vazia, para a qual os parênteses são obrigatórios – permitir a presença de um "nada" sem parênteses nas expressões causaria ambigüidades e permitiria que erros comuns de digitação passassem despercebidos.

5.2.4 Representações de lista

Uma representação de lista (*list display*) é uma série – possivelmente vazia – de expressões contidas entre colchetes:

```
test          ::=  and_test ( "or"and_test )* | lambda_form
testlist      ::=  test ( ","test )* [ ","]
list_display  ::=  "["[listmaker] "]"
listmaker     ::=  expression ( list_for | ( ","expression )* [","] )
list_iter     ::=  list_for | list_if
list_for      ::=  "for"expression_list "in"testlist [list_iter]
list_if       ::=  "if"test [list_iter]
```

Uma representação de lista retorna um novo objeto lista. Seu conteúdo é especificado fornecendo ou uma lista de expressões ou pela compreensão de uma lista. Quando uma lista separada por vírgulas é fornecida, seus elementos

são interpretados da esquerda para a direita e colocados no objeto lista nessa ordem. Quando a compreensão de uma lista é fornecida, esta consiste de uma expressão seguida por pelo menos uma cláusula `for` e zero ou mais cláusulas `for` ou `if`. Nesse caso, os elementos da nova lista são aqueles que seriam produzidos considerando cada uma das cláusulas `for` ou `if` como um bloco, aninhando da esquerda para a direita e interpretando a expressão para produzir um elemento da lista a cada vez que o bloco mais interno é alcançado.

5.2.5 Generator expressions

A generator expression is a compact generator notation in parentheses:

```
generator_expression  ::=  "("test genexpr_for ")"
genexpr_for           ::=  "for"expression_list "in"test [genexpr_iter]
genexpr_iter          ::=  genexpr_for | genexpr_if
genexpr_if            ::=  "if"test [genexpr_iter]
```

A generator expression yields a new generator object. It consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. The iterating values of the new generator are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to yield a value that is reached the innermost block for each iteration.

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

The parentheses can be omitted on calls with only one argument. See section **??** for the detail.

5.2.6 Representações de dicionário

Uma representação de dicionário (*dictionary display*) é uma série – possivelmente vazia – de pares chave/valor (*key/datum*) contidos entre chaves:

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

Uma representação de dicionário retorno um novo objeto dicionário.

Os pares chave/valor são interpretados da esquerda para a direita para definir as entradas do dicionário: cada objeto chave é usado como uma chave dentro do dicionário para armazenar os valores correspondentes.

As restrições dos tipos de valores das chaves são listadas anteriormente nessa seção 3.2. (Resumindo, o tipo da chave deve ser indexável, o que exclui todos os tipos de objetos mutáveis.) Conflitos entre chaves duplicadas não são detectados; prevalecem os últimos valores (textualmente os mais à direita na representação) armazenados para um dado valor de chave.

5.2.7 Conversões para string

Uma conversão para string (*string conversion*) é uma lista de expressões contidas entre aspas reversas:

```
string_conversion  ::=  "‘"expression_list "‘"
```

Em uma conversão para string a lista de expressões contida é interpretada e o objeto resultante é convertido em uma string de acordo com regras específicas para cada tipo.

Se o objeto é uma string, um número, `None`, ou então uma tupla, lista ou dicionário contendo apenas objetos de um desses tipos, a string resultante é uma expressão Python válida que pode ser passada para a função interna `eval()` para gerar uma expressão com o mesmo valor (ou uma aproximação, caso estejam envolvidos números de ponto flutuante).

(Particularmente, a conversão de uma string adiciona aspas sobre ela e converte caracteres “estranhos” para seqüencias de escape que podem ser impressas sem problemas.)

Objetos recursivos (por exemplo, listas ou dicionários que contêm uma referência a si próprios, direta ou indiretamente) usam ‘`...`’ para indicar uma referência recursiva, e o resultado não pode ser passado para a função `eval()` para obter um valor igual (em vez disso, é levantada uma exceção `SyntaxError`).

A função interna `repr()` realiza exatamente a mesma conversão nos seus argumentos que a colocação entre parênteses e entre aspas invertidas fazem. A função interna `str()` realiza uma conversão similar, mas mais amigável.

5.3 Primárias

Primárias (*primaries*) representam as operações mais fortemente ligadas da linguagem. Sua sintaxe é:

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

5.3.1 Referências a atributos

Uma referência a um atributo (*attribute reference*) é uma primária seguida por um ponto e um nome:

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

A interpretação de uma primária deve retornar um objeto cujo tipo suporte referências a atributos, por exemplo, um módulo, uma lista ou uma instância. É feita então uma requisição a este objeto para retornar o atributo cujo nome é o identificador. Se este atributo não estiver disponível, é levantada uma exceção `AttributeError`. Caso contrário, o tipo e o valor do objeto retornado são determinados pelo objeto. Múltiplas chamadas a uma mesma referência a um atributo podem retornar objetos diferentes.

5.3.2 Subscrições

Uma subscrição (*subscription*) seleciona um item de uma seqüência (string, tupla ou lista) ou de um mapeamento (dicionário):

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

A interpretação da primária deve retornar um objeto do tipo seqüência ou mapeamento.

Se a primária for um mapeamento, a interpretação da lista de expressões deve retornar um objeto cujo valor é uma das chaves do mapeamento, e a subscrição seleciona o valor do mapeamento que corresponde àquela chave. (A lista de expressões é uma tupla, a não ser que contenha exatamente um item).

Se a primária for uma seqüência, a interpretação da expressão (ou lista de expressões) deve retornar um inteiro simples. Se este valor for negativo, o tamanho da seqüência é adicionado ao mesmo (para que, por exemplo, `x[-1]` selecione o último item de `x`). O valor resultante deve ser um inteiro não negativo menor do que o número de itens da seqüência, e a subscrição seleciona o item cujo índice é esse valor (contando a partir do zero).

Os itens de uma string são caracteres. Um caractere não é um tipo de dado separado, mas uma string com exatamente um caractere.

5.3.3 Fatiamentos

O fatiamento (*slicing*) seleciona uma faixa de itens de uma seqüência (como por exemplo uma string, uma tupla ou uma lista). Fatiamentos podem ser usados como expressões ou como alvos em comandos de atribuição ou deleção. A sintaxe do fatiamento é:

```
slicing           ::=  simple_slicing | extended_slicing
simple_slicing    ::=  primary "["short_slice "]"
extended_slicing  ::=  primary "["slice_list "]"
slice_list        ::=  slice_item ","slice_item)* [","]
slice_item        ::=  expression | proper_slice | ellipsis
proper_slice      ::=  short_slice | long_slice
short_slice       ::=  [lower_bound] ":"[upper_bound]
long_slice        ::=  short_slice ":"[stride]
lower_bound       ::=  expression
upper_bound       ::=  expression
stride            ::=  expression
ellipsis          ::=  "..."
```

Há uma ambigüidade na sintaxe formal aqui: qualquer coisa que tem a forma de uma lista de expressões também tem a forma de uma lista de fatias, então qualquer subscrição pode ser interpretada como um fatiamento. Em vez de complicar mais a sintaxe, esta ambigüidade é removida através da definição de que, nesse caso, a interpretação como subscrição tem prioridade sobre a interpretação como fatiamento (isso se aplica se a lista de fatias não contém fatias válidas e nem elipses). Similarmente, quando a lista de fatias tem exatamente uma fatia curta (*short slice*), sem vírgula no final, a interpretação como fatiamento simples tem prioridade sobre a interpretação como fatiamento estendido.

A semântica do fatiamento simples é a seguinte: a interpretação da primária deve retornar uma seqüência. A interpretação dos limites inferior (*lower bound*) e superior (*upper bound*), se estes estiverem presentes, deve retornar inteiros simples; os valores padrão são zero e `sys.maxint`, respectivamente. Se qualquer um dos limites for negativo, é adicionado a este o tamanho da seqüência. O fatiamento então seleciona todos os itens com índice *k* tais que *i* `<=` *k* `<` *j*, onde *i* e *j* são os limites inferior e superior especificados. Essa seleção pode retornar uma seqüência vazia. Se *i* e *j* estão fora da faixa de índices válidos, isso não causa um erro (tais itens não existem, portanto não são selecionados).

A semântica do fatiamento estendido é a seguinte: a primária deve ser um mapeamento, o qual é indexado com uma chave que é construída a partir da lista de fatias: se a lista de fatias contém pelo menos uma vírgula, a chave é uma tupla contendo a conversão dos itens dessa lista; caso contrário, a chave é a conversão do único item da lista de fatias. A conversão de um item da lista de fatias que é uma expressão é a própria expressão. A conversão de uma elipse da lista de fatias é o objeto pré-definido `Ellipsis`. A conversão de uma fatia corretamente formulada é um objeto do tipo fatia (ver section 3.2), cujos atributos `start`, `stop` e `stride` são os valores das expressões dadas como limite inferior, limite superior e passo, respectivamente, substituindo `None` pelas expressões que faltam.

5.3.4 Chamadas

Uma chamada (*call*) chama um objeto executável (por exemplo, uma função) com uma série – possivelmente vazia – de argumentos:

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

Pode haver uma vírgula depois dos argumentos e keywords, mas isso não afeta a semântica.

A interpretação da primária deve retornar um objeto executável (funções definidas pelo usuário, funções internas, métodos de objetos internos, objetos do tipo classe, métodos de instâncias de classes e em alguns casos as próprias instâncias de classes são executáveis; extensões podem definir tipos de objetos executáveis adicionais). Todas as expressões do argumento são interpretadas antes que a chamada seja feita. Veja section 7.5 para saber sobre a

sintaxe de listas de parâmetros formais.

Se há argumentos nomeados, estes primeiro são convertidos para argumentos posicionais, como segue. Primeiro, é criada uma lista de posições em branco para os parâmetros formais. Se há N argumentos posicionais, estes são colocados nas N primeiras posições. A seguir, para cada argumento nomeado, o identificador é usado para determinar a posição correspondente (se o identificador é igual ao nome do primeiro parâmetro formal, é usada a primeira posição, e assim por diante). Se a posição já está preenchida, é levantada uma exceção `TypeError`. Caso contrário, o valor do argumento é colocado na posição, preenchendo-a (mesmo que a expressão seja `None`, a posição é preenchida). Quando todos os argumentos tiverem sido processados, as posições que ainda estão em branco são preenchidas com o valor padrão correspondente na definição da função. (Os valores padrão são calculados, uma única vez, quando a função é definida; portanto, um objeto mutável, como uma lista ou um dicionário, usado como valor padrão será compartilhado por todas as chamadas que não especificam um valor de argumento para a posição correspondente; isso deveria geralmente ser evitado.) Se há posições em branco para as quais não é especificado nenhum valor padrão, é levantada uma exceção `TypeError`. Caso contrário, a lista de posições preenchidas é usada com argumento para a chamada.

Se há mais argumentos posicionais do que posições para os parâmetros formais, é levantada uma exceção `TypeError`, a não ser que haja um parâmetro usando a sintaxe '`*identificador`'; nesse caso, o parâmetro formal recebe uma tupla contendo os argumentos posicionais excedentes (uma uma tupla vazia se não houver nenhum argumento posicional excedente).

Se algum argumento nomeado não corresponde a nenhum parâmetro formal, é levantada uma exceção `TypeError`, a não ser que esteja presente um parâmetro formal com a sintaxe '`**identificador`'; nesse caso, esse parâmetro formal recebe um dicionário contendo os argumentos nomeados excedentes (usando os nomes como chaves e os valores do argumento como valores correspondentes a essas chaves), ou um (novo) dicionário vazio, se não houverem argumentos nomeados excedentes.

Se a sintaxe '`*expressão`' aparece na chamada da função, o resultado da interpretação de '`expressão`' deve ser uma seqüência. Os elementos dessa seqüência são tratados com se fossem argumentos posicionais adicionais; se há argumentos posicionais *x*,...,*xN*, e o resultado da interpretação de '`expressão`' é uma seqüência *y1*,...,*yM*, isso é equivalente a uma chamada com M+N argumentos posicionais *x1*,...,*xN*,*y1*,...,*yM*.

Uma conseqüência disso é que, embora a sintaxe '`*expressão`' apareça depois de quaisquer argumentos nomeados, esta é processada *antes* dos argumentos nomeados (e do argumento **expressão, se houver algum – 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 ?
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

É incomum o uso de argumentos nomeados e da sintaxe '`*expressão`' na mesma chamada, então, na prática, não acontece essa confusão.

Se a sintaxe '`**expressão`' aparece na chamada à função, a interpretação de '`expressão`' deve resultar em (uma subclasse de) um dicionário, cujo conteúdo é tratado como argumentos nomeados adicionais. No caso de um mesmo nome aparecer tanto na '`expressão`' quanto como um argumento nomeado explicitamente, é levantada uma exceção `TypeError`.

Os parâmetros formais que usam a sintaxe '`*identificador`' ou '`**identificador`' não podem ser usados como argumentos posicionais nem como nomes de argumentos nomeados. Os parâmetros formais que usam a sintaxe '`(sublista)`' não podem ser usados como nomes de argumentos nomeados; a lista mais externa corresponde a uma única posição não nomeada, e o valor do argumento é atribuído à sublista usando as regras usuais de atribuição para tuplas, após o processamento de todos os outros parâmetros.

Uma chamada sempre retorna algum valor, possivelmente `None`, a não ser que cause uma exceção. Como esse valor é computado depende do tipo do objeto chamado.

Se este for—

uma função definida pelo usuário: O bloco de código da função é executado, sendo-lhe passada a lista de argumentos. A primeira coisa que o bloco de código fará será ligar os parâmetros formais aos argumentos; isso é descrito na seção 7.5. Quando o bloco de código executa um comando `return`, é especificado o valor de retorno da chamada à função

uma função ou método interno: O resultado fica a cargo do interpretador; ver *Python Library Reference* para descrições das funções e métodos internos.

um objeto do tipo classe: É retornada uma nova instância dessa classe.

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

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

5.4 O operador de potência

O operador de potência *power* se liga mais fortemente do que os operadores unários à sua esquerda e se liga menos fortemente do que que os operadores unários à sua direita. Sua sintaxe é:

```
power  ::=  primary ["**"u_expr]
```

Assim, em uma seqüencia sem parênteses de operadores unários e operadores de potência, os operadores são interpretados da direita para a esquerda (isso não restringe a ordem de avaliação dos operandos).

O operador de potência tem a mesma semântica da função interna `pow()`, quando chamada com dois argumentos: retorna o argumento da esquerda elevado ao argumento da direita. Os argumentos numéricos são primeiro convertidos para um tipo comum. O tipo resultante é o tipo dos argumentos convertidos.

Com diferentes tipos de operandos misturados, se aplicam as regras de conversão para operadores aritméticos binários. Para argumentos do tipo inteiro e inteiro longo, o resultado é do mesmo tipo dos operandos (após a conversão), a menos que o segundo argumento seja negativo; nesse caso, todos os argumentos são convertidos para números de ponto flutuante, e é retornado como resultado um número ponto flutuante. Por exemplo, `10**2` retorna `100`, mas `10**-2` retorna `0.01`. (Esta função foi adicionada no Python 2.2. No Python 2.1 e anteriores, se ambos os argumentos fossem inteiros e o segundo argumento fosse negativo, era levantada uma exceção).

Elevar `0.0` a um a potência negativa resulta em uma exceção `ZeroDivisionError`. Elevar um valor negativo a uma potência fracionária resulta em uma exceção `ValueError`.

5.5 Unary arithmetic operations

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

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

O operador unário `-` (menos) retorna a negação do seu argumento numérico.

O operador unário `+` (mais) retorna o seu argumento numérico inalterado.

O operador unário (inversão) retorna a inversão bit-a-bit do seu argumento inteiro longo ou simples. A inversão bit-a-bit de `x` é definida como `-(x+1)`. Se aplica somente a números integrais.

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

5.6 Operações aritméticas binárias

As operações aritméticas binárias têm os níveis de prioridade convencionais. Note que algumas dessas operações também se aplicam a certos tipos não-numéricos. Fora o operador de potência, há apenas dois níveis, um para operadores multiplicativos e um para operadores aditivos:

```
m_expr  ::=  u_expr | m_expr "*"u_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) retorna o produto dos seus argumentos. Os argumentos devem ser ambos números, ou um argumento deve ser um inteiro (simples ou longo) e o outro deve ser uma seqüência. No primeiro caso, os números são convertidos para um mesmo tipo e então multiplicados. No segundo caso, é executada a repetição da seqüência; um fator de repetição negativo retorna uma seqüência vazia.

Os operadores / (divisão) e // (divisão inteira) retornam o quociente dos seus argumentos. Os argumentos numéricos são primeiro convertidos para um mesmo tipo. A divisão de inteiros simples ou longos retorna um inteiro do mesmo tipo; o resultado é o resultado da divisão matemática com a função '*floor*' aplicada ao resultado. A divisão por zero levanta uma exceção `ZeroDivisionError` exception.

O operador % (modulo) retorna o resto da divisão do primeiro argumento pelo segundo. Os argumentos numéricos são primeiro convertidos para um mesmo tipo. Um argumento direito igual a zero 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 `0.34` é igual a `4*0.7 + 0.34`.) O operador módulo sempre retorna um resultado com o mesmo sinal do segundo operando (ou zero); o valor absoluto do resultado é estritamente menor do que o valor absoluto do segundo operando[1].

Os operadores divisão inteira e módulo são conectados pela seguinte identidade: `x == (x/y)*y + (x%y)`. A divisão inteira e o módulo também são conectadas pela função inteira `divmod(x,y) == (x/y, x%y)`. Estas identidades não valem para números de ponto flutuante; identidades similares valem de forma aproximada quando `x/y` é substituído por `floor(x/y)` ou `floor(x/y) - 1`[2]. **Deprecated since release 2.3.** O operador de divisão inteira (*floor*) e a função `divmod()` não são mais definidas para números complexos. Em vez disso, converta para um número de ponto flutuante usando a função `abs()`, se for o caso.

O operador + (adição) retorna a soma dos seus argumentos. Os argumentos devem ser ou ambos números ou ambos seqüências do mesmo tipo. No primeiro caso, os números são convertidos para um mesmo tipo e então adicionados. No segundo, as seqüências são concatenadas.

O operador - (subtração) retorna a diferença entre os seus argumentos. Os argumentos numéricos são primeiro convertidos para um mesmo tipo.

5.7 Operações de deslocamento (*shifting*)

As operações de deslocamento têm prioridade mais baixa do que as operações aritméticas:

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

Estes operadores aceitam como argumentos inteiros simples ou longos. Os argumentos são convertidos para um mesmo tipo. Eles deslocam o primeiro argumento para a esquerda ou para a direita em um número de bits dado pelo segundo argumento.

Um deslocamento à direita em *n* bits é definido como um divisão por `pow(2,n)`. Um deslocamento à esquerda em *n* bits é definido como uma multiplicação por `pow(2,n)`; para inteiros simples não há verificação de *overflow* de forma que nesse caso a operação descarta bits e troca o sinal se o resultado não for menor do que `pow(2,31)`,

[1] Apesar da expressão `abs(x%y) < abs(y)` ser matematicamente verdadeira, para números de ponto flutuante a mesma pode não ser numericamente verdadeira devido ao arredondamento. Por exemplo, assumindo uma plataforma na qual o número de ponto flutuante do Python seja um número de precisão dupla no padrão IEEE 754, de modo que `-1e-100 %1e100` tenha o mesmo sinal que `1e100`. A função `fmod()` do módulo `math` retorna um resultado cujo sinal é igual ao do primeiro argumento em vez do segundo, e portanto retorna `-1e100` nesse caso. A abordagem mais adequada depende da aplicação.

[2] Se `x` é muito próximo de um inteiro exato múltiplo de `y`, é possível que `floor(x/y)` seja maior do que `(x-x%y)/y` devido ao arredondamento. Em casos desse tipo, Python retorna o último resultado, para manter a garantia de que codedivmod(x,y)[0] * y + x % y seja muito próximo de `x`.

em valor absoluto. Deslocamentos negativos causam uma exceção `ValueError`.

5.8 Operações bit-a-bit binárias

5.9 Binary bit-wise operations

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

```
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 `&` retorna o AND bit-a-bit entre seus argumentos, que devem ser inteiros simples ou longos. Os argumentos são convertidos para um mesmo tipo.

O operador `^` retorna o XOR (OU exclusivo) bit-a-bit entre seus argumentos, que devem ser inteiros simples ou longos. Os argumentos são convertidos para um mesmo tipo.

O operador `|` retorna o OR (inclusivo) bit-a-bit entre seus argumentos, que devem ser inteiros simples ou longos. Os argumentos são convertidos para um mesmo tipo.

5.10 Comparações

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

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

Comparações retornam valores booleanos: `True` (verdadeiro) ou `False` (falso).

Comparações podem ser encadeadas arbitrariamente, por exemplo, `x < y <= z` é equivalente a `x < y and y <= z`, exceto que `y` é interpretado apenas uma vez (mas em ambos os casos `z` não é interpretado quando o resultado de `x < y` resulta em falso).

Formalmente, se *a*, *b*, *c*, ..., *y*, *z* são expressões e *opa*, *opb*, ..., *opy* são operadores de comparação, então *a opa b opb c* ...*y opy z* é equivalente a *a opa b* `and` *b opb c* `and` ...*y opy z*, exceto que cada expressão é interpretada pelo menos uma vez.

Note que *a opa b opb c* não implica nenhum tipo de comparação entre *a* e *c*, de modo que, por exemplo, `x < y > z` é perfeitamente legal (embora talvez não seja muito bonito).

As formas <> e `!=` são equivalentes; para manter a consistência com o C, prefere-se `!=`; a seguir, onde é mencionado `!=`, <> também é aceito. A forma <> é considerada obsolescente.

Os operadores <, >, ==, >=, <= e `!=` comparam os valores de dois objetos. Os objetos não precisam ser do mesmo tipo. Se ambos forem números, são convertidos para o mesmo tipo. Caso contrário, objetos de tipos diferentes *sempre* são comparados como desiguais, e são ordenados consistente mas arbitrariamente.

(Essa definição pouco usual de comparação foi usada para simplificar a definição de operações como a ordenação e os operadores `in` e `not in`. Futuramente, as regras de commparações para objetos de tipos diferentes serão possivelmente modificadas.)

A comparação de objetos do mesmo tipo depende do tipo:

- Números são comparados aritmeticamente.
- Strings são comparadas lexicograficamente, usando os equivalentes numéricos da função interna `ord()` dos seus caracteres. Strings unicode e de 8 bits são totalmente interoperáveis nesse comportamento.

- Tuplas e listas são comparadas lexicograficamente usando a comparação de elementos correspondentes. Isso significa que para serem comparadas como iguais, cada elemento deve ser comparado como igual e as duas seqüências devem ser do mesmo tipo e ter o mesmo número de elementos.

  Se não forem iguais, as seqüências são ordenadas da mesma maneira que os seus primeiros elementos diferentes. Por exemplo, `cmp([1,2,x], [1,2,y])` retorna o mesmo que `cmp(x,y)`. Se o elemento correspondente não existe, a menor seqüência fica em primeiro na ordenação (por exemplo, `[1,2] < [1,2,3]`).

- Mapeamentos (dicionários) são comparados como iguais se e somente se suas listas (chave, valor) ordenadas forem comparadas como iguais.[3] Outros resultados que não a igualdade são resolvidos consistentemente, mas não são definidos de outro modo.[4]

- A maioria dos demais tipos é comparada como desigual a não ser que sejam o mesmo objeto; a escolha se um objeto é considerado maior ou menor do que o outro é feita arbitrariamente, mas consistentemente dentro de uma mesma execução de um programa.

Os operadores `in` e `not in` testam a pertinência a um conjunto. *x* `in` *s* é interpretado como verdadeiro se *x* é um membro do conjunto *s*, e como falso caso contrário. *x* `not in` *s* retorna a negação de *x* `in` *s*. O teste de pertinência a um conjunto tem sido tradicionalmente ligado a seqüências; um objeto é membro de um conjunto se este conjunto é uma seqüência e contém um elemento que é igual a este objeto. Entretanto, é possível que um objeto suporte testes de pertinência mesmo que não seja uma seqüência. Particularmente, dicionários suportam testes de pertinência como um modo mais elegante da forma *key* `in` *dict*; outros tipos de mapeamentos podem manter esta adequação.

Para os tipos tupla e lista, *x* `in` *y* é verdadeiro se e somente se existe um índice *i* tal que *x* `==` *y*`[`*i*`]` é verdadeiro.

Para os tipos Unicode e string, *x* `in` *y* é verdadeiro se e somente se *x* é uma substring de *y*. Um teste equivalente é `y.find(x) != -1`. Note que *x* e *y* não precisam ser do mesmo tipo; consequentemente, `u'ab' in 'abc'` retornará `True`. Strings vazias são sempre consideradas substrings de qualquer outra string, logo `in "abc"` retornará `True`.

Changed in version 2.3: Anteriormente, era exigido que *x* fosse uma string de tamanho `1`.

Para classes definidas pelo usuário que definem o método `__contains__()`, *x* `in` *y* é verdadeiro se e somente se *y*`.__contains__(`*x*`)` é verdadeiro.

Para classes definidas pelo usuário que não definem o método `__contains__()` e que definem o método `__getitem__()`, *x* `in` *y* é verdadeiro se e somente se existe um índice inteiro não-negativo *i* tal que *x* `==` *y*`[`*i*`]`, e todos os índices inteiros menores não causam a exceção `IndexError`. (Se qualquer outra exceção for levantada, é como se essa exceção tivesse sido causada pelo operador `in`).

O operador `not in` é definido como o valor verdade inverso de `in`.

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. *x* `is not` *y* retorna o valor verdade inverso.

5.11 Operações Booleanas

As operações Booleanas têm a prioridade mais baixa dentre todas as operações da linguagem Python:

```
expression  ::=  or_test | lambda_form
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 das operações Booleanas, e também quando as expressões são usadas em comandos de controle de fluxo, os seguintes valores são interpretados como falsos: `False`, `None`, o zero numérico de todos os tipos,

[3] A implementação computa isso eficientemente, sem construção de listas nem ordenação.

[4] Versões anteriores do Python usavam comparações lexicográficas dos listas (chave, valor) ordenadas, mas isso era muito custoso para o caso comum da comparação por igualdade. Uma versão ainda mais antiga do Python comparava dicionários somente por identidade, mas isso causava surpresas, porque as pessoas esperavam poder testar se um dicionário era vazio comparando-o com `{}`.

seqüências vazias (strings, tuplas, listas, dicionários, sets e frozensets). Todos os outros valores são interpretados como verdadeiros.

O operador `not` retorna `True` se o seu argumento é falso, `False` caso contrário.

Na expressão *x* `and` *y*, primeiro é interpretado *x*; se *x* é falso, seu valor é retornado; caso contrário, *y* é interpretado e o valor resultante é retornado.

Na expressão *x* `or` *y*, primeiro é interpretado *x*; se *x* é verdadeiro, seu valor é retornado; caso contrário, *y* é interpretado e o valor resultante é retornado

(Note que nem `and` nem `or` restringem o valor e o tipo que retornam a `False` e `True`, mas em vez disso retornam o último arugumento interpretado. Algumas vezes isso é útil, por exemplo, se `s` é uma string que deve ser substituída por um valor padrão caso seja vazia, a expressão `s or 'foo'` retorna o valor desejado. Porque o `not` precisa inventar uma valor de qualquer forma, ele não se preocupa em retornar um valor que seja do mesmo tipo que o seu argumento, então por exemplo `not 'foo'` retorna `False`, e não `''`.)

5.12 Lambdas

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

Construções lambda (expressões lambda) têm a mesma posição sintática do que expressões. Elas são uma forma rápida de criar funções anônimas; a expressão `lambda` *arguments*`:` *expression* retorna um objeto do tipo função. Os objeto não nomeado se comporta como uma função definida com

```
def name(arguments):
    return expression
```

Veja a section 7.5 para informação sobre a sintaxe das listasa de parâmetros. Note que funções criadas com construções lambda não podem conter comandos.

5.13 Listas de expressões

```
expression_list  ::=  expression ( ","expression )* [","]
```

Uma lista de expressões contendo pelo menos uma vírgula retorna uma tupla. O tamanho da tupla é o número de expressões contidas na lista. As expressões são interpretadas da esquerda para a direita.

A vígula no final é obrigatória somente para a criação de uma tupla de um único elemento (também chamada de *singleton*); em todos os demais casos é opcional. Uma única expressão sem vírgula no final não cria uma tupla, mas em vez disso retorna o valor da expressão. (Para criar uma tupla vazia, use um par de parênteses vazio: `()`.)

5.14 Ordem de interpretação

O interpretador Python interpreta as expressões da esquerda para a direita. Note que durante a interpretação de uma atribuição o lado direito é interpretado antes do lado esquerdo.

Nas linhas seguintes, as expressões serão interpretadas na ordem aritmética dos seus sufixos:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
func(expr1, expr2, *expr3, **expr4)
expr3, expr4 = expr1, expr2
```

5.15 Summary

A tabela a seguir resume as precedências de operadores em Python, da menor precedência (ligação mais fraca) à maior precedência (ligação mais forte). Os operadores na mesma caixa têm a mesma precedência. A menos que a sintaxe seja dada explicitamente, os operadores são binários. Os operadores na mesma caixa são agrupados da esquerda para a direita (exceto as comparações, incluindo os testes, os quaos tem a mesma precedência e são encadeados da esquerda para a direita – veja a seção 5.10 – e exponenciação, que se agrupam da direita para a esquerda).

Operador	**Descrição**
`lambda`	Expressão lambda
`or`	OR Booleano
`and`	AND Booleano
`not` *x*	NOT Booleano
`in`, `not in`	Testes de pertinência
`is`, `is not`	Testes de identidade
`<`, `<=`, `>`, `>=`, `<>`, `!=`, `==`	Comparações
`|`	OR bit-a-bit
`^`	XOR bit-a-bit
`&`	AND bit-abit
`<<`, `>>`	Deslocamentos
`+`, `-`	Adição e subtração
`*`, `/`, `%`	Multiplicação, divisão, resto
`+`*x*, `-`*x*	Positivo, negativo
`~`*x*	NOT bit-a-bit
`**`	Exponenciação
x`.`*atributo*	Referência a atributos
x`[`*indice*`]`	Subscrição
x`[`*indice*`:`*indice*`]`	Fatiamento
f`(`*argumentos*`...)`	Chamada de função
`(`*expressoes*`...)`	Ligação ou representação de tupla
`[`*expressoes*`...]`	Representação de lista
`{`*chave*`:`*dado*`...}`	Representação de dicionário
`‘`*expressoes*`...‘`	Conversão para string

CAPÍTULO

SIX

Instruções simples

Instruções simples são compreendidas em uma única linha lógica. Várias instruções simples podem ocorrer em uma única linha, separadas por ponto-e-vírgula. A sintaxe para essas instruções é:

```
simple_stmt  ::=  expression_stmt
                | assert_stmt
                | assignment_stmt
                | augmented_assignment_stmt
                | pass_stmt
                | del_stmt
                | print_stmt
                | return_stmt
                | yield_stmt
                | raise_stmt
                | break_stmt
                | continue_stmt
                | import_stmt
                | global_stmt
                | exec_stmt
```

6.1 Expressões

Expressões são usadas para computar e escrever um valor ou para chamar um procedimento (uma função que não retorna qualquer resultado significativo; em Python, procedimentos retornam o valor `None`). Existem outras formas permitidas e ocasionalmente úteis de se usar expressões. A sintaxe para uma expressão é definida por:

```
expression_stmt  ::=  expression_list
```

Uma expressão interpreta a lista de expressões passada (que pode ser uma única expressão) e devolve o resultado.

No modo interativo, se o valor não for `None`, ele é convertido para uma string usando a função `repr()` e a string resultante é escrita em uma linha na saída de dados padrão (veja section 6.6). (Expressões que retornem `None` não são escritas, de modo que esse procedimento não gera nenhuma saída.)

6.2 Assertivas

Assertivas são uma maneira conveniente de inserir verificações para a depuração de 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`', equivale a:

```
if __debug__:
   if not expression1: raise AssertionError, expression2
```

Essas equivalências assumem que `__debug__` e AssertionError referem às variáveis *built-in* com esses nomes. Na implementação atual, a variável `__debug__` é `True` sob circunstâncias normais e `False` quando otimização é usada. O interpretador não gera nenhum código para uma assertiva quando é usado otimização durante a compilação. Note que é desnecessário incluir o código fonte da expressão que falho na mensagem de erro; ele será mostrado no *traceback* do erro.

Atribuições à `__debug__` são ilegais. O valor dessa variável é determinado quando o interpretador inicia.

6.3 Atribuições

Atribuições são usadas para (re)ligar nomes à valores e para modificar atributos ou items de objetos mutáveis:

```
assignment_stmt  ::=  (target_list "=")+ expression_list
target_list      ::=  target (","target)* [","]
target           ::=  identifier
                      | "("target_list ")"
                      | "["target_list "]"
                      | attributeref
                      | subscription
                      | slicing
```

(Veja a seção 5.3 para a definição da sintaxe dos três últimos símbolos.)

Uma atribuição resolve a lista de expressões passada (lembre-se que pode ser uma única expressão ou uma lista de expressões separadas por vírgulas, nesse caso retornando uma tupla) e atribui o objeto resultanta a cada uma das variáveis da lista de alvos, da esquerda para a direita.

A atribuição é definida recursivamente, dependendo do formato dos alvos. Quando um alvo é parte de um objeto mutável (uma referência de um atributo, subscrição ou *slicing*) esse objeto que deve realizar a atribuição e decidir sobre sua validade, podendo levantar uma exceção se a atribuição for inaceitável. As regras observadas pelo objeto e as exceções levantadas dependem da definição do próprio objeto (veja a seção 3.2).

A atribuição de um objeto a uma lista de alvos é definida recursivamente da seguinte forma:

- Se a lista é um único alvo: o objeto é atribuido a esse alvo.
- Se a lista é uma sequência de alvos, separados por vírgulas: o objeto deve ser uma sequência com o número de items igual ao número de alvos. Os objetos são atribuídos, da esquerda para a direita aos alvos correspondentes. (esta restrição ficou mais leve a partir de Python 1.5; em versões anteriores o objeto tinha de ser uma tupla. Como strings são sequências legais, uma atribuição como '`a, b = "xy"`' é legal e a string tiver o mesmo comprimento.)

A atribuição de um objeto a um único alvo é definida da seguinte forma:

- Se o alvo é um identificador (variável, nome):
  - Se o nome não ocorre em um comando `global` no bloco de código atual: o nome é ligado ao objeto no escopo atual.
  - Em outros casos, o nome é simplesmente ligado ao objeto no escopo atual.

O nome já existente tem seu valor reatribuído. Isso pode causar a contagem de referências do objeto previamente ligado chegar a zero, provocando sua desalocação da mesmória e a chamada da destrutora (se tiver uma).

- Se o alvo é uma lista de alvos entre parênteses ou colchetes: o objeto deve ser uma sequência com o mesmo número de items, que são atribuídos aos alvos correspondentes.
- Se o alvo é uma referência de um atributo: a primeira expressão é resolvida. Ela deve retornar um objeto com atributos livres; se não for o caso, a exceção `TypeError` é levantada. Pede-se então que aquele objeto atribua o objeto ao atributo em questão; se ele não puder realizar a atribuição, é levantada uma exceção (geralmente, mas não necessariamente `AttributeError`).
- Se o alvo é uma subscrição: a primeira expressão é resolvida, devendo retornar ou uma sequência mutável (uma lista, por exemplo) ou um mapeamento (um dicionário). Em seguida, a subscrição é resolvida e os valores atribuídos.

  Se a expressão é uma sequência mutável (como uma lista), a subscrição deve retornar um número inteiro. Se for negativo, o valor do comprimento da sequência é adicionado a ele. O valor resultante deve ser um inteiro não negativo menor que o comprimento da sequência. Em seguida é requisitado que a sequência atribua o objeto ao item daquele índice. Se o item está fora do comprimento da sequência, `IndexError` é levantada (atribuição a uma sequência não pode adicionar novos items, apenas substituir items existentes).

  Se a expressão for um mapeamento (um dicionário), a subscrição deve ser de um tipo compatível com as chaves, e então pede-se que o objeto crie um par chave/dados que mapeia a subscrição ao objeto atribuído. Isto pode tanto substituir um par existente (com uma chave igual) ou inserir um novo (se a chave já não existir).
- Se o alvo for um fatiamento (“slicing”): a expressão é resolvida e deve retornar uma sequência mutável (como uma lista. O objeto atribuído deve ser uma sequência do mesmo tipo. Em seguida, caso existam, as expressões dando os limites superior e inferior (que devem ser inteiros, em geral pequenos) são resolvidas; os valores padrão são zero e o comprimento da seqência. Se qualquer um dos limites for negativo, o comprimento da sequência é adicionado a ele. Os valores resultantes são então ajustados para ficar entre zero e o comprimento da sequência, inclusive. Finalmente, o objeto é usado para substituir a fatia (“slice”) com os items da sequência atribuída. O comprimento da fatia pode ser diferente do comprimento da sequência atribuída, mudando o comprimento da sequência alvo, se ela permitir.

(Na implementação atual, assume-se que a sintaxe para alvos é idêntica à usada para expressões, e uma sintaxe inválida é rejeitada na fase de geração do código, com mensagens de erro menos detalhadas.)

AVISO: Apesar da definição de atribuição implicar que repassar variáveis entre o lado esquerdo e o lado direito da expressão é uma operação ‘segura (exemplo: ‘`a, b = b, a`’ troca duas variáveis), o mesmo *dentro* de uma coleção de variáveis sendo atribuídas não é seguro! Por exemplo, o seguinte programa escreve ‘`[0, 2]`’:

```
x = [0, 1]
i = 0
i, x[i] = 1, 2
print x
```

6.3.1 Atribuições incrementais

Uma atribuição incremental é a combinação, em um único comando, de uma operação binária e uma atribuição:

```
augmented_assignment_stmt  ::=  target augop expression_list
augop                      ::=  "+= -= "*= "/= "%= "**="
                                |  >>= <<= "&= "^= ="
```

(Veja a seção 5.3 para definições da sintaxe dos três últimos símbolos.)

Uma atribuição incremental resolve o alvo (que, ao contrário de outras operações de atribuição, não pode ser uma sequência) e a lista de expressões, realizando a operação específica em relação ao tipo dos operandos, e por fim atribui o resultado ao alvo original. O alvo é resolvido apenas uma vez.

Uma expressão `x += 1` pode ser escrita como `x = x + 1` para obter um efeito semelhante, mas não idêntico. Na versão incremental, `x` é resolvido apenas uma vez. Além disso, quando for possível, a operação é realizada *in-place*. Ao invés de criar um novo objeto e atribuído ao alvo, o objeto antigo é modificado.

Com a exceção de atribuições a tuplas e múltiplos alvos no mesmo comando, a atribuição incremental é realizada da mesma maneira que uma atribuição normal. De forma similar, com a exceção do comportamento *in-place* citado acima, a operação binária realizada pela atribuição é a mesma de uma operação normal.

Para alvos que são referências de atributos, o valor inicial é obtido com o método `getattr()` e o resultado é atribuído com o método `setattr()`. Note que os dois métodos não referem necessariamente a mesma variável. Quando `getattr()` refer a uma variável da classe, `setattr()` ainda atribui o valor a uma variável da instância. Por exemplo:

```
class A:
    x = 3    # variável da classe
a = A()
a.x += 1     # atribui 4 à a.x, deixando 3 em A.x
```

6.4 O comando `pass`

```
pass_stmt  ::=  "pass"
```

`pass` é uma operação nula — quando é executado, nada acontece. É útil para tornar válido um espaço vazio, quando um comando é requerido sintaticamente, mas nenhum código precisa ser executado, por exemplo:

```
def f(arg): pass    # uma função que não faz nada(ainda)

class C: pass       # uma classe sem métodos (ainda)
```

6.5 O comando `del`

```
del_stmt  ::=  "del"target_list
```

A deleção é recursivamente definida de forma bem similar às atribuições. Ao invés de passá-la em todos os detalhes, aqui estão algumas dicas.

Deletar uma lista alvo recursivamente deleta cada elemento, da esquerda pra direita.

Deletar um nome remove a sua ligação do espaço tanto local quanto global, caso o nome ocorra num comando `global` no mesmo bloco de código. Se o nome não está ligado a nada, uma exceção `NameError` será levantada.

É ilegal deletar um nome do espaço local se ele ocorre como uma variável livre em um bloco aninhado.

Deleções de referências de atributos, subscrições e *slicings* são passadas para o objeto primário envolvido na operação; deleção de um *slicing* é em geral equivalente a atribuição um *slice* vazio do tipo correto (determinado pelo tipo do objeto).

6.6 O comando `print`

```
print_stmt  ::=  "print"( [expression ("," expression)* [","]]
                 | >>"expression [("," expression)+ [","]] )
```

`print` interpreta cada expressão passada e escreve o objeto resultante na saída de dados padrão (veja adiante). Se um objeto não é uma string, ele é primeiro convertido, usando as regras de conversão. A string (resultante ou

original) é então escrita. Um espaço é escrito antes de cada objeto, a menos que para o sistema de saída de dados o cursor esteja posicionado no começo da linha. Este é o caso (1) quando nenhum caracter foi ainda escrito na saída de dados padrão, (2) quando o último caracter escrito na saída foi '`\n`', ou (3) quando a última operação de escrita na saída de dados não foi com o comando `print`. (Por esta razão, em alguns outros casos, pode ser interessante escrever uma string vazia na saída de dados.) **Note:** Objetos que se comportam como arquivos mas não são os objetos arquivo embutidos na linguagem, frequentemente não emulam corretamente este aspecto dos arquivos, então é melhor não confiar nesse comportamente.

Um caracter '`\n`' é escrito no fim, a menos que o comando `print` termine com uma vírgula. Esta é a única ação se o comando contem apenas a keyword `print`.

A saída de dados padrão é definida como o objeto arquivo nomeado `stdout` no módulo `sys`. Se o objeto não existir, ou se ele não tiver um método `write()`, então uma exceção `RuntimeError` é levantada.

`print` tem ainda uma forma estendida, definida pela segunda parte da sintaxe descrita acima. Esta forma é algumas vezes referida como "`print` chevron.". Nesta forma, a primeira expressão depois de `>>` deve resolver em em um objeto arquivo ou que se comporte como arquivo, tendo um método `write()`, como descrito acima. Com esta forma estendida, as expressões subsequentes são impressas neste objeto arquivo. Se a primeira expressão é resolvida como `None`, então `sys.stdout` é usado como o arquivo de saída.

6.7 O comando `return`

```
return_stmt  ::=  "return"[expression_list]
```

`return` pode ocorrer apenas aninhado em uma definição de função, nunca em uma definição de classe.

Se a lista de expressões está presente, ela é interpretada, caso contrário `None` é o valor.

`return` deixa a chamada de função atual com a lista de expressões (ou `None`) como valor de retorno.

Quando `return` sai do controle de uma instrução `try` com uma cláusula `finally`, essa cláusula é executada antes de deixar a função.

Em um gerador, não é permitido incluir uma lista de expressões (`expression_list`) no comando `return`. Nesse contexto, um `return` indica que o gerador finalizou e irá levantar a exceção `StopIteration`.

6.8 O comando `yield`

```
yield_stmt  ::=  "yield"expression_list
```

O comando `yield` é usado apenas quando definindo um gerador, e é usado apenas no corpo da função que define o gerador. Usar o comando `yield` em uma definição de função é o suficiente para que aquela definição crie um gerador ao invés de uma função normal.

Quando uma função que define um gerador é chamado, ela retorna um iterador, ou o nome mais comumente usado, um gerador. O corpo da função é executado chamando-se o método `next()` do gerador repetidamente, até que uma exceção seja levantada.

Quando um comando `yield` é executado, o estado do gerador é congelado e o valor da `expression_list` é retornado para o invocador do método `next()`. Por "congelado" nós queremos dizer que o estado local é armazenado, incluindo as ligações atuais a variáveis locais, o ponteiro de instrução e a pilha de resolução interna: é guardada toda informação necessária para que na próxima vez que o método `next()` for chamado, a função possa proceder exatamente como se o comando `yield` fosse apenas outra chamada externa normal.

O comando `yield` não é permitido na cláusula `try` de um comando `try` ... `finally`. O problema é que não há qualquer garantia de que o gerador irá continuar a execução, portanto, nenhuma garantia de que o bloco `finally` será executado.

Note: Em Python 2.2, o comando `yield` só é permitido quando geradores foram habilitados. Eles estão habilitados por padrão em Python 2.3. O seguinte comando pode ser usado para habilitá-los em Python 2.2:

```
from __future__ import generators
```

See Also:

PEP 0255, "*Simple Generators*"
The proposal for adding generators and the `yield` statement to Python.

6.9 O comando `raise`

```
raise_stmt  ::=  "raise"[expression ["," expression ["," expression]]]
```

Se nenhuma expressão estiver presente, `raise` levanta novamente a última exceção ativa no escopo atual. Se nenhuma exceção esteve ativa, uma exceção `Queue.Empty` é levantada indicando esse erro.

Em outros casos, `raise` resolve as expressões para obter três objetos, usando `None` como o valor das expressões omitidas. Os primeiros dois objetos são usados para determinar o *tipo* e *valor* da exceção.

Se o primeiro objeto é uma instância, o tipo da exceção é a sua classe, a própria instância é o valor e o segundo objeto deve ser `None`.

Se o primeiro objeto é uma classe, ela se torna o tipo da exceção. O segundo objeto é usado para determinar o valor da exceção: se ele for uma instância da classe, ela se torna o valor. Se for uma tupla, ela é usada como a lista de argumentos para a construtora da classe; se for `None`, uma lista de argumentos vazia é usada, e qualquer outro objeto é tratado como um único argumento para a construtora. A instância então criada pela chamada à construtura é usada como valor da exceção.

Se um terceiro objeto estiver presente e não for `None`, ele deve ser um objeto traceback (veja a seção 3.2), e é usado ao invés do local atual como o lugar em que a exceção ocorreu. Se o terceiro objeto estiver presente e não for o objeto traceback ou `None`, uma exceção `TypeError` é levantada. A forma com três expressões de raise é útil para levantar novamente uma exceção de forma transparente em uma cláusula except, mas `raise` com nenhuma expressão é preferida caso a exceção a ser levantada foi a última exceção ativa no escopo atual.

Informação adicional sobre exceções pode ser encontrada em section 4.2, e informação sobre manipulação de exceções se encontra em section 7.4.

6.10 O comando `break`

```
break_stmt  ::=  "break"
```

`break` pode ocorrer aninhado em um loop `for` ou `while`, mas nunca em uma definição de função ou classe dentro desse loop.

Ele termina o loop mais próximo, pulando a cláusula opcional `else`, caso o loop tenha uma.

Se um loop `for` é terminado por `break`, as variáveis alvo do loop mantêm seu valor atual.

Quando `break` remove o controle de um comando `try` com uma cláusula `finally`, essa cláusula é executada antes de deixar o loop.

6.11 O comando `continue`

```
continue_stmt  ::=  "continue"
```

`continue` pode ocorrer aninhado em um loop `for` ou `while`, mas não em uma definição de função ou classe dentro desse loop. [1] Ele continua com o próximo ciclo do loop mais próximo.

[1] Ele pode ocorrer dentro de uma cláusula `except` ou `else`. A restrição sobre o `try` é por pura preguiça durante a implementação do comando e com o tempo será removida.

6.12 O comando `import`

```
import_stmt  ::=  "import"module ["as"name] ( ","module ["as"name] )*
                  | "from"module "import"identifier ["as"name]
                   ( ","identifier ["as"name] )*
                  | "from"module "import("identifier ["as"name]
                   ( ","identifier ["as"name] )* [","] ")"
                  | "from"module "import*"
module       ::=  (identifier ".")* identifier
```

O comando import é executado em dois passos: (1) encontre o módulo e inicialize-o se necessário; (2) defina um nome ou nomes no espaço local (do escopo onde o comando `import` ocorre). A primeira forma (sem `from`) repete esses passos para cada identificador na lista. A forma com `from` executa o passo (1) uma vez e depois o passo (2) repetidamente.

Neste contexto, "inicializar" um módulo *built-int* ou uma extensão, significa chamar uma função de inicialização que o módulo deve fornecer para este propósito (na implementação de referência, o nome da função é obtido prefixando a string "init" ao nome do módulo); "inicializar" um módulo com código puramente em Python significa simplesmente executar o código do corpo desse módulo.

O sistema mantem uma tabela de módulos que foram ou estão sendo inicializados, indexada pelo nome do módulo. Essa tabela é acessível como `sys.modules`. Quando um nome de um módulo é encontrado nesta tabela, significa que o passo (1) foi concluído. Se não, uma busca por uma definição do módulo é iniciada. Quando encontrado, é então carregado. Detalhes no processo de buscar e carregar um módulo dependem da implementação e plataforma. Geralmente envolve buscar por um módulo *built-int* com o nome dade e então procurá-lo em uma lista de locais dada por `sys.path`.

Se um módulo *built-in* é encontrado, seu código de inicialização é executado é o passo (1) é finalizado. Se nenhum arquivo coincidente for encontrado, a exceção `ImportError` é levantada. Se um erro de sintaxe ocorre, a exceção `SyntaxError` é levantada. Senão, um módulo vazio com o nome dado é criado e inserido na tabela, e então o código do é executado dentro desse módulo vazio. Exceções durante esta execução encerram o passo (1).

Quando passo (1) termina sem levantar uma exceção, o passo (2) pode começar.

A primeira forma do comando `import` atribui o nome do módulo no espaço local ao objeto correspondente, e então passa para o próximo identificador, se houver. Se o nome do módulo é seguido da keyword `as`, o nome seguinte é usado como o nome local para o módulo.

A forma usando `from` não atribui o módulo ao seu nome; ao invés disso, passa-se por toda a lista de identificadores, procurando cada um no módulo encontrado no passo (1), e então liga-se o nome no espaço local ao objeto encontrado. Como ocorre com a primeira forma de `import`, um nome local alternativa pode ser especificado usando-se "`as` nome". Se um nome não for encontrado, `ImportError` é levantada. Se a lista de identificadores for substituída por um asterisco ('`*`'), então todos os nomes públicos definidos no módulos são ligados no espaço local do comando `import`.

Os *nomes públicos* definidos por um módulo são determinados por uma variável no módulo chamada `__all__`; se definida, ela deve ser uma sequência de strings que são os nomes definidos ou importáveis desse módulo. Os nomes em `__all__` são todos considerados públicos e existentes. Se `__all__` não é definida, então o conjunto de nomes públicos inclui todos os nomes encontrados no módulo que não começam com um '_'. `__all__` deve conter toda a API pública desse módulo. A intenção é evitar exportar acidentalmente itens que não são parte da API (como outros módulos da biblioteca que foram importados e usados apenas dentro do módulo).

A forma usando `from` com '`*`' pode ocorrer apenas no escopo do módulo. Se for usada dentro de uma função, e a função contem ou está contida em um bloco com outras variáveis, o compilador levanta a exceção `SyntaxError`.

Hierarquia dos nomes em um módulo:

quando os nomes em um módulo contém um ou mais pontos, a busca pelo caminho do módulo é feita de forma diferente. A sequência de identificadores até o último ponto é usada para encontrar um "pacote"; o último identificado é então procurado dentro do pacote. Um pacote é geralmente um subdiretório de um diretório em `sys.path` com um arquivo '__init__'.

[XXX Can't be bothered to spell this out right now; see the URL http://www.python.org/doc/essays/packages.html

for more details, also about how the module search works from inside a package.]

A função `__import__()` é fornecida para suportar aplicações que determinam dinamicamente que módulos precisam ser carregados; veja Funções Built-in na *Referência da Biblioteca Python* para mais informaçõesl.

6.12.1 Comandos futuros

Um *comando futuro* orienta o compilador de que um determinado módulo deve ser compilado usando a sintaxe ou a semântica que estará disponível numa versão futura da linguagem. A intenção é facilitar a migração para versões futuras que introduzem mudanças que criam incompatibilidades na linguage. Ele permite o uso de novas funções em determinados módulos, antes que a função se torne o padrão.

```
future_statement  ::=  "from__future__import"feature ["as"name] (","feature ["as"name
                       | "from__future__import("feature ["as"name] (","feature ["as"n
feature           ::=  identifier
name              ::=  identifier
```

Um comando futuro deve aparecer logo no topo do módulo. As únicas linhas permitidas antes dele são:

- a *docstring* do módulo (se houver),
- comentários,
- linhas em branco,
- outros comandos futuros.

As funções reconhecidas em Python 2.3 são '`generators`' (geradores), '`division`' (divisão de inteiros) e '`nested_scopes`' (escopos aninhados). '`generators`' e '`nested_scopes`' são redundantes em Python 2.3 porque estão sempre ativos.

Um comando futuro e reconhecido e tratado durante a compilação: mudanças na semântica do núclero da linguagem são frequentemente implementadas gerando um código diferente. É até possível que uma nova função introduza novas incompatibilidades na linguagem (como uma nova palavra reservada), caso em que o compilador precisa interpretar o código do módulo de forma diferente. Estas decisões não podem ser deixadas de lado até o momento da execução.

Para qualquer versão, o compilador sabe exatamente que funções foram definidas e levanta um erro se um comando futuro contém algo desconhecido.

A semântica usada no momento da execução é a mesma de qualquer comando *import*: há um módulo padrão chamado `__future__`, descrito adiante, e ele será importado da maneira usual no momento em que o comando é executado.

Note que não há nada de especial no comando:

```
import __future__ [como um nome]
```

Esse não é um comando futuro; é um comando import ordinário, sem nenhuma semântica especial ou restrição na sintaxe.

Código compilado pelo comando `exec` ou chamadas às funções `compile()` e `execfile()` que ocorram em um módulo `M` contendo um comando futuro irão, por definição, usar a nova sintaxe ou semântica associada com o comando. Isso pode, a partir de Python 2.2, ser controlado com argumentos opcionais à função `compile()` — veja a documentação dessa função na biblioteca padrão para mais detalhes.

Um comando futuro digitado no *prompt* do interpretador interativo irá ter efeito pelo resto da sessão. Se o interpretador é iniciado com a ação **-i** e se for passado para execução um *script* que contenha um comando futuro, ele estará em efeito na sessão apenas depois que o script for executado.

6.13 O comando `global`

```
global_stmt  ::=  "global"identifier (","identifier)*
```

O comando `global` é uma declaração que vale para todo o bloco de código atual. Significa que todos os identificadores listados são interpretados como globais. É impossível atribuir um valor à uma variável global sem usar o comando `global`, apesar de ser possível se referir à globais sem ter sido declaradas como tal.

Nomes listados em um comando `global` não devem ser usados no mesmo bloco de texto antes do comando.

Nomes listados no comando não devem ser definidos como parâmetros formais ou como variável alvo de um loop `for`, definiçÕes de classes, definições de funções ou no comando `import`.

(A implementação atual não impõe restrições formais às duas últimas restrições citadas, mas os programas não devem abusar dessa liberdade, já que futuras implementações podem alterar esse comportamento.

Nota: o comando `global` é uma diretiva para o *parser*. Ele se aplica apenas ao código processado no mesmo momento que o comando. Por exemplo, quando usado em um comando `exec`, ele não afeta o bloco que código que contém esse `exec`, e o código contido nele não é afetado por um comando `global` contido no bloco que contém o `exec`. O mesmo se aplica às funções eval(), `execfile()` e `compile()`.

6.14 O comando `exec`

```
exec_stmt  ::=  "exec"expression ["in"expression [","expression]]
```

Este comando permite a execução dinâmica de código Python. A primeira expressão deve resolver em uma string, um objeto arquivo aberto ou um objeto código. Se for uma string, ela é processada como se fosse um conjunto de instruções, que são então executadas (a menos que ocorra um erro de sintaxe). Se for um arquivo, ele é processado até EOF e executado. Se for um objeto código, ele é simplesmente executado. Em todos os casos, espera-se que o código a ser executado seja válido, como originário de um arquivo (veja a seção 8.2, “File input”). Esteja ciente de que os comandos `return` e `yield` não podem ser usados fora de definição de funções, mesmo no contexto do código passado para o comando `exec`.

Em todos os casos, se as partes opcionais forem omitidas, o código é executado no escopo atual. Se apenas a primeira expressão depois de `in` é especificada, deve ser um dicionário, usado tanto para as variáveis globais quanto para as locais. Se duas expressões são passadas, são usadas para as variáveis globais e locais, respectivamente. Se passada, *locals* pode ser qualquer mapeamento. Changed in version 2.4: anteriormente, era requerido que *locals* fosse um dicionário.

Como efeito colateral, uma implementação pode inserir chaves adicionais nos dicionários além daquelas correspondentes as variáveis criadas pelo código executado. Por exemplo, a implementação atual pode inserer uma referência ao dicionário do módulo built-in __builtin__ na chave `__builtins__` (!).

Dica:

resolução dinâmica de expressões é possível através da função `eval()`. As funções `globals()` e `locals()` retornam os dicionários globais e locais, respectivamente, o que pode ser útil para uso com o comando `exec`.

CAPÍTULO

SEVEN

Instruções compostas

Instruções compostas comtêm (grupos de) outras instruções; elas afetam ou controlam de alguma forma a execução destas outras instruções. Em geral, instruções compostas se propagam por várias linhas, apesar de que quando usadas de formas simples, ela pode estar contida em uma só linha.

As instruções `if`, `while` e `for` implementam as tradicionais estruturas de controle de fluxo. `try` especifica os manipladores de exceções e/ou código de limpeza para um grupo de instruções. Definições de funções e de classes também são consideradas sintaticamente como instruções compostas.

Instruções compostas consistem de uma ou mais 'cláusulas.' Uma cláusula consiste de um cabeçalho e de um 'corpo'(suite, no original). Os cabeçalhos das diversas cláusulas de uma instrução composta estão sempre no mesmo nível de indentação. Cada cabeçalho começa com uma keyword identificador e termina com um sinal de dois-pontos. O corpo é um grupo de instruções controlado por uma cláusula. Ele pode ter um ou mais instruções separadas por um sinal de ponto-e-vírgula na mesma linha do cabeçalho, logo após o sinal de dois pontos do cabeçalho, ou pode ter uma ou mais instruções indentadas nas linhas seguintes. Apenas a última forma pode conter intruções compostas aninhadas; o seguinte exemplo é ilegal, principalmente por não deixar claro a qual `if` um possível `else` seguinte pertenceria:

```
if test1: if test2: print x
```

Note ainda que o ponto-e-vírgula estabelece uma ligação mais forte que o sinal de dois-pontos neste contexto, então, no exemplo seguinte, são executados ou todas ou nenhuma das instruções `print` presentes:

```
if x < y < z: print x; print y; print z
```

Sumário:

```
compound_stmt  ::=  if_stmt
                    | while_stmt
                    | for_stmt
                    | try_stmt
                    | funcdef
                    | classdef
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 com o símbolo `NEWLINE` possivelmente seguido de um `DEDENT`. Note ainda que cláusulas de continuação opcionais sempre começam com uma keyword que não pode iniciar uma instrução, para que não hajam ambiguidades. (o problema de a qual `if` pertence o `else` é resolvido em Python requerendo que todos os `if` aninhados estejam indentados.

Para deixar mais claro, a formatação das regras gramaticais adiante posiciona cada cláusula em uma linha separada.

7.1 O comando `if`

O comando `if` é usado para execução condicional:

```
if_stmt  ::=  "if"expression ":"suite
              ( "elif"expression ":"suite )*
              ["else:"suite]
```

Ele seleciona exatamente um dos corpos existentes, resolvendo as expressões, até que uma delas seja verdadeira (veja a seção 5.11 para as definições de verdadeiro e falso); então, aquele corpo é executado (e nenhuma outra parte do comando `if` é executada ou resolvida). Se todas as expressões são falsas, o corpo da cláusula `else`, se existente, é executado.

7.2 O comando `while`

O comando `while` é usado para execução contínua, enquanto uma expressão for verdadeira:

```
while_stmt  ::=  "while"expression ":"suite
                 ["else:"suite]
```

O comando repetidamente testa a expressão e, se for verdadeira, executa o primeiro corpo; se a expressão for falsa (o que pode ocorrer na primeira vez que é testada) o corpo da cláusula `else`, se presente, é executado, e o loop termina.

O comando `break` executado no primeiro corpo finaliza o loop sem executar o corpo da cláusula `else`. O comando `continue` executado no primeiro corpo pula o resto dele e volta a testar a validade da expressão.

7.3 O comando `for`

O comando `for` é usado para iterar pelos elementos de uma sequência (como uma string, tupla ou lista) ou qualquer outro objeto iterável:

```
for_stmt  ::=  "for"target_list "in"expression_list ":"suite
               ["else:"suite]
```

A lista de expressões é resolvida uma vez; ela deve retornar um objeto iterável. Um iterador é criado e o corpo é então executado uma vez para cada item, em ordem ascendente. Cada item é então passado para a lista de alvos (target_list) usando as regras normais de definição, e então o corpo é executado. Quando os itens acabam (o que ocorre imediatamente se a sequência estiver vazia), o corpo da cláusula `else`, se presente, é executado, e o loop termina.

O comando `break` executado no primeiro corpo finaliza o loop sem executar o corpo da cláusula `else`. O comando `continue` executado no primeiro corpo pula para o próximo item da sequência, ou para a cláusula `else` se não houver um próximo item.

O corpo pode designar objetos para as variáveis na lista de alvos; isso não afeta o próximo item designado a elas.

A lista de alvos não é apagada quando o loop é finalizado, mas se a sequência estiver vazia, naturalmente as variáveis da lista não terão recebido objeto algum pelo loop. Dica: a função built-in `range()` retorna uma sequência de inteiros adequada para realizar uma sequência de execuções através de um índice qualquer, de forma similar ao `for` encontrado em C ou Pascal.

Warning: Há um problema sutil quando a sequência está sendo modificada pelo loop (o que pode ocorrer com sequências mutáveis, como listas). Um contador interno é usado para manter um controle de qual é o próximo item, e é incrementado a cada iteração. Quando este contador chega ao valor do comprimento da sequência, o loop termina. Isso significa que se o corpo deleta o item atual(ou um item anterior) da sequência, o loop vai pular o próximo item (já que ele toma o índice do item atual, que já foi manipulado). De forma semelhante, se o corpo insere um item na sequência antes do item atual, ele será manipulado novamente na próxima passagem do loop. Isso pode levar a bugs sérios que podem ser evitados fazendo uma cópia temporária da sequência. Por exemplo:

```
for x in a[:]:
    if x < 0: a.remove(x)
```

7.4 O comando `try`

O comando `try` especifica manipuladores de exceções e/ou código de limpeza para um grupo de instruções:

```
try_stmt      ::=  try_exc_stmt | try_fin_stmt
try_exc_stmt  ::=  "try:"suite
                   ("except"[expression ["","target]] ":"suite)+
                   ["else:"suite]
try_fin_stmt  ::=  "try:"suite "finally:"suite
```

A duas formas de usar o comando `try`: `try`...`except` e `try`...`finally`. Estas formas não podem ser misturadas (mas podem ser aninhadas uma na outra).

A forma `try`...`except` especifica um ou mais manipuladores de exceções (as cláusulas `except`). Quando nenhuma exeção ocorre na cláusula `try`, nenhum manipulador é executado. Quando uma exceção ocorre, uma procura por um manipulador inicia. Essa procura inspeciona cada uma das cláusulas except até que é encontrada uma que coincida com a exceção. Uma cláusula excetp sem uma exceção, se existente, deve ser a última; ela coincide com qualquer exceção. Para uma cláusula except com uma expressão, essa expressão é resolvida, e se o objeto resultante for compatível com a exceção, ela é marcada como coincidente. Um objeto é coincidente com a exceção se ele é a classe ou uma classe base da exceção, ou, no caso (desaconselhável) de strings como exceções, é a própria string que foi levantada (note que a identidade do objeto tem de coincidir, isto é, tem de ser o mesmo objeto, não apenas uma string com o mesmo valor).

Se nenhuma cláusula coincide com a exceção, a busca continua no código em volta e na pilha de invocação.

Se a resolução de uma expressão no cabeçalho de uma cláusula except levanta uma exceção, a busca original é cancelada, e uma nova busca é iniciada para a nova exceção no código em volta da expressão e na pilha de chamada (o código é tratado como se todo o comando `try` levantou a exceção).

Quando uma cláusula except coincidente é encontrada, a exceção é atribuído ao alvo especificado naquela cláusula, se presente, e então a cláusula é executada. Todas as cláusulas except têm de conter um bloco de código executável. Quando chega ao fim desse bloco, a execução continua normalmente após todo o comando try. (isso significa que se existirem dois manipuladores aninhados para a mesma exceção, e a exceção ocorre no try do manipulador interno, o manipulador externo não irá manipular a exceção.)

Antes que o corpo de uma cláusula except seja executado, detalhes sobre a exceção são designados à três variáveis no módulo `sys` module: `sys.exc_type` recebe o objeto identificador da exceção; `sys.exc_value` recebe os parâmetros; e `sys.exc_traceback` recebe um objeto traceback (veja a seção 3.2), identificando o ponto no programa em que a exceção ocorreu. Estes detalhes também estão disponíveis através da função `sys.exc_info()` que retorna uma tupla (*exc_type*, *exc_value*, *exc_traceback*). O uso dessa função é preferível as variáveis correspondentes, já que estas são inseguras em um programa que use threads. A partir de Python 1.5, as variáveis têm seus valores originais restaurados (antes da chamada) quando retornando de uma funçãoque manipulou a exceção.

A cláusula opcional `else` é executada se e quando o controle passa do fim da cláusula `try`. .[1] Exceções na cláusula `else` não são manipuladas pelas cláusulas `except` precedentes.

A forma `try`...`finally` especifica um manipulador de 'limpeza'. A cláusula `try` é executada. Quando nenhuma exceção ocorre, a cláusula `finally` é executada. Quando uma exceção ocorre na cláusula `try`, a exceção é temporariamente salva, a cláusula `finally` é executada a então a exceção salva é re-levantada. Se a cláusula `finally` levanta outra exceção ou executa uma instrução `continue` ou `break`, a exceção salva é perdida. Uma instrução `continue` na cláusula `finally` é ilegal. (A razão é um problema com a implementação atual – essa restrição pode ser removida no futuro). A informação sobre a exceção não está disponível para o programa durante a execução da cláusula `finally`.

[1] Atualmente, o controle "passa do fim", exceto no caso de uma exceção ou na execução dos comandos `return`, `continue` ou `break`.

Quando uma instrução `return`, `break` ou `continue` é executada no corpo do comando `try`...`finally`, a cláusula `finally` também é executada 'durante a saída.' Uma instrução `continue` na cláusula `finally` é ilegal. (A razão é um problema com a implementação atual – essa restrição pode ser removida no futuro).

Mais informações sobre exceções podem ser encontradas na seção 4.2, e informações sobre o uso do comando `raise` para gerar exceções podem ser encontradas na seção 6.9.

7.5 Definição de funções

Uma definição de função cria um objeto function, de acordo com a definição do usuário (veja a seção 3.2):

```
funcdef         ::=  [decorators] "def"funcname "("[parameter_list] "):"suite
decorators      ::=  decorator+
decorator       ::=  "@"dotted_name ["("[argument_list [","]] ")"] NEWLINE
dotted_name     ::=  identifier ("."identifier)*
parameter_list  ::=  (defparameter ",")*
                     (  "*"identifier [, "**"identifier]
                      | "**"identifier
                      | defparameter [","] )
defparameter    ::=  parameter ["="expression]
sublist         ::=  parameter ("," parameter)* [","]
parameter       ::=  identifier | "("sublist ")"
funcname        ::=  identifier
```

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

A definição não executa o corpo da função; este é executado apenas quando a função é chamada.

A function definition may be wrapped by one or more decorator expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code:

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

is equivalent to:

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

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

Valores padrão de parâmetros são resolvidos quando a definição da função é executada. Isso significa que a expressão só é resolvida uma vez, quando a definição da função é resolvida, e que o mesmo valor "pré-computado" é usado para todas as chamadas. É importante entender isso, principalmente quando o valor padrão é um objeto mutável, como uma lista ou dicionário: se a função modifica o objeto (por exemplo, adicionando um item a uma lista) o valor padrão está definitivamente modificado. Geralmente, não é essa a intenção. Uma forma de contornar isso é usar `None` como padrão, e testar por ele 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 da chamada de funções é descrita em mais detalhes na seção **??**. Uma chamada de função sempre designa valores a todos os parâmetros mencionados na lista de parâmetros, seja de argumentos posicionais, de palavras-chave ou de valores padrão. Se a forma "`*identifier`" está presente, essa variável é inicializada com uma tupla contendo quaisquer argumentos posicionais em excesso. Se a forma "`**identifier`" é usada, a varíável é inicializada com um dicionário contendo quaisquer argumentos com palavras-chave usados em excesso. Os valores padrão são uma tupla e um dicionário vazios, respectivamente.

É também possível criar funções anônimas (funções que não são ligadas a um nome), para uso imediato em expressões. Elas usam o a instrução lambda, descrita na seção 5.12. Note que lambda é meramente um atalho para uma definição simplificada de função; uma função definida com o comando "`def`" pode ser repassada ou ter o nome redefinido, assim como uma função definida por lambda. O comando "`def`" é mais poderoso pois permite a execução de múltiplos comandos dentro do corpo da função.

Nota: Funções são objetos de primeira classe. Um comando "`def`" executado dentro de uma definição de função, cria uma função local que pode ser retornada ou passada adiante. Variáveis livres usadas na função aninhada podem acessar variáveis locais da função contendo o comando. Veja a seção 4.1 para detalhes.

7.6 Definição de classes

Uma definição de classe cria um objeto class (veja a seção 3.2):

```
classdef     ::=  "class"classname [inheritance] ":"suite
inheritance  ::=  "("[expression_list] ")"
classname    ::=  identifier
```

Uma definição de classe é uma instrução executável. Ela primeiro resolve a lista de hierarquia, se presente. Cada item na lista deve resolver em um objeto class ou type que possa ser subclasse. O corpo da classe é então executado num novo quador de execução (ver seção 4.1), usando um novo espaço local e o espaço global original. (normalmente, o corpo contém apenas definições de funções.) Quando o corpo da classe termina a execução, seu quadro de execução é descartado, mas o espaço local é salvo. Um objeto class é então criado, usando a lista de hierarquia para as classes base e o espaço local salvo como o dicionário de atributos. O nome da classe é ligado a esse objeto class no espaço local original.

Nota: variáveis definidas na definição de classe são variáveis da classe; elas são compartilhadas por todas as instâncias. Para definir variáveis da instância, elas devem ter um valor designado no método `__init__()` ou em qualquer outro método. Ambas as variáveis da classe ou da instância são acessíveis através da notação "`self.name`", e uma variável da instância oculta uma variável da classe de mesmo nome quando acessada dessa forma. Variáveis de classe com valores imutáveis podem ser usadas como valores padrão para as variáveis das instâncias. For new-style classes, descriptors can be used to create instance variables with different implementation details.

CAPÍTULO

EIGHT

Componentes de alto-nível

O interpretador Python pode receber sua entrada de uma série de fontes: de um script passado como entrada de dados padrão ou como argumento, digitado interativamente, de um módulo, etc. Este capítulo cobre a sintaxe usada nestes casos.

8.1 Programas Python completos

Quando a uma especificação de uma linguagem não precisar prescrever como o interpretador deve ser invocado, é útil ter uma noção de um programa completo na linguagem. Um programa completo em Python é executado em um ambiente minimamente inicializado: todos os módulos da bilioteca padrão ou *built-in* estão disponíveis, mas nenhum foi inicializado, exceto pelos módulos `sys` (vários serviços do sistema), `__builtin__` (funções, exceções e `None`) e `__main__`. O ótimo é usado para prover o ambiente local e global para a execução do programa.

A sintaxe de um programa completo (quando a entrada de dados é um arquivo) descrita na próxima seção.

O interpretador também pode ser invocado em modo interativo; neste caso, ele não lê executa um programa completo, e sim um comando de cada vez. O ambiente inicial é idêntico ao de um programa; cada comando é executado no ambiente do módulo `__main__`.

Em UNIX, um programa completo pode ser passado para o interpretador em três formas: com a opção **-c** *string* na linha de comando, como um arquivo passado como primeiro argumento, ou pela entrada de dados padrão. Se o arquivo ou entrada de dados é um terminal, o interpretador entra em modo interativo; caso contrário, ele executa o arquivo como um programa completo.

8.2 Arquivo como entrada de dados

Todo código lido de um arquivo em modo não interativo tem a mesma forma:

```
file_input  ::=  (NEWLINE | statement)*
```

Essa sintaxe é usada nas seguintes situações:

- quando estiver *parseando* um programa completo (de um arquivo ou de uma string);
- quando estiver *parseando* um módulo;
- quando estiver *parseando* uma string passada para o comando `exec`;

8.3 Entrada de dados interativa

A entrada de dados em modo interativo é lida usando a seguinte gramática:

```
interactive_input  ::=  [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Note que uma instrução composta deve ser seguida de uma linha em branco quando em modo interativo; isso é necessário para que o *parser* possa detectar o fim da entrada.

8.4 Expressões como entrada de dados

Há duas formas de usar expressões como entrada de dados. Ambas ignoram os espaços no começo da linha.

A string usada como argumento para a função `eval()` deve ter a seguinte forma.

```
eval_input  ::=  expression_list NEWLINE*
```

A linha lida pela função `input()` deve ter a seguinte forma:

```
input_input  ::=  expression_list NEWLINE
```

Nota: para ler linhas 'cruas' sem interpretá-las, você pode usar a função `raw_input()` ou o método `readline()` em arquivos.

APÊNDICE

A

Histórico e Licenças

A.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes
2.3.3	2.3.2	2003	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes
2.4	2.3	2004	PSF	yes
2.4.1	2.4	2005	PSF	yes

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible

licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

A.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.4.2

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.4.2 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.4.2 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2004 Python Software Foundation; All Rights Reserved" are retained in Python 2.4.2 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.4.2 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.4.2.

4. PSF is making Python 2.4.2 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.4.2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.4.2 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.4.2, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.4.2, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 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.

CNRI LICENSE AGREEMENT FOR 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.

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.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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.

A.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

A.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.keio.ac.jp/ matumoto/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
```

A.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/about/index.html.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ‘‘AS IS’’ AND
GAI_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 GAI_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 GAI_ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN GAI_ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

A.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

A.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.
```

A.3.5 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
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.
```

A.3.6 Cookie management

The `Cookie` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

A.3.7 Profiling

The `profile` and `pstats` modules contain the following notice:

```
Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software
and its associated documentation for any purpose (subject to the
restriction in the following sentence) 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 InfoSeek not be used in
advertising or publicity pertaining to distribution of the software
without specific, written prior permission.  This permission is
explicitly restricted to the copying and modification of the software
to remain in Python, compiled Python, or other languages (such as C)
wherein the modified or derived code is exclusively imported into a
Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION 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.
```

A.3.8 Execution tracing

The `trace` module contains the following notice:

```
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.
```

A.3.9 UUencode and UUdecode functions

The uu module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

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
```

A.3.10 XML Remote Procedure Calls

The `xmlrpclib` module contains the following notice:

```
    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.
```

ÍNDICE REMISSIVO

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S