Table Of Contents

Previous topic

2. Usando o interpretador Python

Next topic

4. Mais ferramentas de controle de fluxo

This Page

Report a Bug
Show Source

Quick search

Go

Enter search terms or a module, class or function name.

3. Uma introdução informal a Python¶

Nos exemplos seguintes, pode-se distinguir a entrada da saída pela presença ou ausência dos prompts (`>>>` e `...`): para repetir o exemplo, você deve digitar tudo após o prompt, quando o mesmo aparece; linhas que não começarem com um prompt são na verdade as saídas geradas pelo interpretador. Observe que quando aparece uma linha contendo apenas o prompt secundário `...` você deve digitar uma linha em branco; é assim que se encerra um comando de múltiplas linhas.

Muitos dos exemplos neste manual, até mesmo aqueles digitados interativamente, incluem comentários. Comentários em Python são iniciados pelo caractere `#`, e se estendem até o final da linha física. Um comentário pode aparecer no início da linha, depois de um espaço em branco ou código, mas nunca dentro de uma string literal. O caractere `#` em uma string literal não passa de um caractere `#`. Uma vez que os comentários são usados apenas para explicar o código e não são interpretados pelo Python, eles podem ser omitidos ao digitar os exemplos.

Alguns exemplos:

```
# este é o primeiro comentário
SPAM = 1                 # e este é o segundo comentário
                         # ... e agora um terceiro!
STRING = "# Isto não é um comentário."
```

3.1. Usando Python como uma calculadora

Vamos experimentar alguns comandos simples em Python. Inicie o interpretador e aguarde o prompt primário, `>>>`. (Não deve demorar muito.)

3.1.1. Números

O interpretador funciona como uma calculadora bem simples: você pode digitar uma expressão e o resultado será apresentado. A sintaxe de expressões é a usual: operadores `+`, `-`, `*` e `/` funcionam da mesma forma que em outras linguagens tradicionais (por exemplo, Pascal ou C); parênteses podem ser usados para agrupar expressões. Por exemplo:

```
>>> 2+2
4
>>> # Isto é um comentário
... 2+2
4
>>> 2+2  # em um comentário na mesma linha do código
4
>>> (50-5*6)/4
5
>>> # A divisão entre inteiros arredonda para baixo:
... 7/3
2
>>> 7/-3
-3
```

O sinal de igual (`'='`) é usado para atribuir um valor a uma variável. Depois de uma atribuição, nenhum resultado é exibido antes do próximo prompt:

```
>>> largura = 20
>>> algura = 5*9
>>> largura * altura
900
```

Um valor pode ser atribuído a diversas variáveis simultaneamente:

```
>>> x = y = z = 0  # Zerar x, y, z
>>> x
0
>>> y
0
>>> z
0
```

Variáveis precisam ser “definidas” (atribuídas um valor) antes que possam ser usadas, se não acontece um erro:

```
>>> # tentar acessar variável não definida
... n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Há suporte completo para ponto flutuante (*float*); operadores com operandos de diferentes tipos convertem o inteiro para ponto flutuante:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Números complexos também são suportados; números imaginários são escritos com o sufixo `j` ou `J`. Números complexos com parte real não nula são escritos como `(real+imagJ)`, ou podem ser criados pela chamada de função `complex(real, imag)`.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Números complexos são sempre representados por dois floats, a parte real e a parte imaginária. Para extrair as partes de um número complexo *z*, utilize `z.real` e `z.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

As funções de conversão para float e inteiro (`float()`, `int()` e `long()`) não funcionam para números complexos — não existe apenas uma maneira de converter um número complexo para um número real. Use `abs(z)` para obter sua magnitude (como um float) ou `z.real` para obter sua parte real.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
```

No modo interativo, o valor da última expressão exibida é atribuída a variável `_`. Assim, ao utilizar Python como uma calculadora, fica mais fácil prosseguir com os cálculos, por exemplo:

```
>>> taxa = 12.5 / 100
>>> preco = 100.50
>>> preco * taxa
12.5625
>>> preco + _
113.0625
>>> round(_, 2)
113.06
```

Essa variável especial deve ser tratada como *somente para leitura* pelo usuário. Nunca lhe atribua explicitamente um valor — do contrário, estaria criando uma outra variável (homônima) independente, que mascararia a variável especial com seu comportamento mágico.

3.1.2. Strings

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples ou duplas:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

O interpretador exibe o resultado de operações com strings da mesma forma como elas são digitadas na entrada: dentro de aspas, e com aspas, caracteres acentuados e outros caracteres especiais representados por sequências de escape com barras invertidas (como `'\t'`, `'\xc3\xa9'` etc.), para mostrar o valor preciso. A string é delimitada entre aspas simples, exceto quando ela contém uma aspa simples e nenhuma aspa dupla. O comando `print` produz uma saída mais legível para tais strings com caracteres especiais.

Strings que contém mais de uma linha podem ser construídas de diversas maneiras. Linhas de continuação podem ser usadas, com uma barra invertida colocada na última posição para indicar que a próxima linha física é a continuação de uma linha lógica:

```
oi = "Eis uma string longa contendo\n\
diversas linhas de texto assim como se faria em C.\n\
     Observe que os espaços em branco no inicio da linha são\
 significativos."

print oi
```

Observe que quebras de linha ainda precisam ser embutidos na string usando `\n` — a quebra de linha física após a última barra de escape é anulada. Este exemplo exibiria o seguinte resultado:

```
Eis uma string longa contendo
diversas linhas de texto assim como se faria em C.
     Observe que os espaços em branco no inicio da linha são significativos.
```

Ou, strings podem ser delimitadas por pares de aspas triplas combinando: `"""` ou `'''`. Neste caso não é necessário escapar o final das linhas físicas com `\`, mas as quebras de linha serão incluídas na string:

```
print """
Uso: treco [OPCOES]
     -h                        Exibir esta mensagem de uso
     -H hostname               Host a conectar
"""
```

produz a seguinte saída:

```
Uso: treco [OPCOES]
     -h                        Exibir esta mensagem de uso
     -H hostname               Host a conectar
```

Se fazemos uma string *raw* (N.d.T: “crua” ou sem processamento de caracteres escape) com o prefixo `r`, as sequências `\n` não são convertidas em quebras de linha. Tanto as barras invertidas quanto a quebra de linha física no código-fonte são incluídos na string como dados. Portanto, o exemplo:

```
oi = r"Eis uma string longa contendo\n\
diversas linhas de texto assim como se faria em C."

print oi
```

Exibe:

```
Eis uma string longa contendo\n\
diversas linhas de texto assim como se faria em C.
```

Strings podem ser concatenadas (coladas) com o operador `+`, e repetidas com `*`:

```
>>> palavra = 'Ajuda' + 'Z'
>>> palavra
'AjudaZ'
>>> '<' + palavra*5 + '>'
'<AjudaZAjudaZAjudaZAjudaZAjudaZ>'
```

Duas strings literais adjacentes são automaticamente concatenadas; a primeira linha do exemplo anterior poderia ter sido escrita como `palavra = 'Ajuda' 'Z'`; isso funciona somente com strings literais, não com expressões que produzem strings:

```
>>> 'str' 'ing'             #  <-  Isto funciona
'string'
>>> 'str'.strip() + 'ing'   #  <-  Isto funciona
'string'
>>> 'str'.strip() 'ing'     #  <-  Isto é inválido
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                      ^
SyntaxError: invalid syntax
```

Strings podem ser indexadas; como em C, o primeiro caractere da string tem índice 0 (zero). Não existe um tipo específico para caracteres; um caractere é simplesmente uma string de tamanho 1. Assim como na linguagem Icon, substrings podem ser especificadas através da notação de *slice* (fatiamento ou intervalo): dois índices separados por dois pontos.

```
>>> palavra[4]
'a'
>>> palavra[0:2]
'Aj'
>>> palavra[2:4]
'ud'
```

Índices de fatias têm defaults úteis; a omissão do primeiro índice equivale a zero, a omissão do segundo índice equivale ao tamanho da string sendo fatiada.:

```
>>> palavra[:2]    # Os dois primeiros caracteres
'Aj'
>>> palavra[2:]    # Tudo menos os dois primeiros caracteres
'udaZ'
```

Diferentemente de C, strings em Python não podem ser alteradas. Tentar atribuir valor a uma posição (índice ou fatia) dentro de uma string resulta em erro:

```
>>> palavra[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> palavra[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Entretanto, criar uma nova string combinando conteúdos é fácil e eficiente:

```
>>> 'x' + palavra[1:]
'xjudaZ'
>>> 'Splat' + palavra[5]
'SplatZ'
```

Eis uma invariante interessante das operações de fatiamento: `s[:i] + s[i:]` é igual a `s`.

```
>>> palavra[:2] + palavra[2:]
'AjudaZ'
>>> palavra[:3] + palavra[3:]
'AjudaZ'
```

Intervalos fora de limites são tratados “graciosamente” (N.d.T: o termo original “gracefully” indica robustez no tratamento de erros): um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia.

```
>>> palavra[1:100]
'judaZ'
>>> palavra[10:]
''
>>> palavra[2:1]
''
```

Índices podem ser números negativos, para iniciar a contagem pela direita. Por exemplo:

```
>>> palavra[-1]   # O último caractere
'Z'
>>> palavra[-2]   # O penúltimo caractere
'a'
>>> palavra[-2:]  # Os dois últimos caracteres
'aZ'
>>> palavra[:-2]  # Tudo menos os dois últimos caracteres
'Ajud'
```

Observe que -0 é o mesmo que 0, logo neste caso não se conta a partir da direita!

```
>>> palavra[-0]
'A'
```

Intervalos fora dos limites da string são truncados, mas não tente isso com índices simples (que não sejam fatias):

```
>>> palavra[-100:]
'AjudaZ'
>>> palavra[-100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

Uma maneira de lembrar como slices funcionam é pensar que os índices indicam posições *entre* caracteres, onde a borda esquerda do primeiro caractere é 0. Assim, a borda direita do último caractere de uma string de comprimento *n* tem índice *n*, por exemplo:

```
 0   1   2   3   4   5   6
 +---+---+---+---+---+---+
 | A | j | u | d | a | z |
 +---+---+---+---+---+---+
-6  -5  -4  -3  -2  -1
```

A primeira fileira de números indica a posição dos índices 0...6 na string; a segunda fileira indica a posição dos respectivos índices negativos. Uma fatia de *i* a *j* consiste em todos os caracteres entre as bordas *i* e *j*, respectivamente.

Para índices positivos, o comprimento da fatia é a diferença entre os índices, se ambos estão dentro dos limites da string. Por exemplo, comprimento de `palavra[1:3]` é 2.

A função built-in (embutida) `len()` devolve o comprimento de uma string:

```
>>> s = 'anticonstitucionalissimamente'
>>> len(s)
29
```

See also:

Sequence Types — str, unicode, list, tuple, bytearray, buffer, xrange
Strings, e as strings Unicode descritas na próxima seção, são exemplos de ***sequências*** e implementam as operações comuns associadas com esses objetos.

String Methods
Tanto strings comuns quanto Unicode oferecem um grande número de métodos para busca e transformações básicas.

String Formatting
Informações sobre formatação de strings com `str.format()` são descritas nesta seção.

String Formatting Operations
As operações de formatação de strings antigas, que acontecem quando strings simples e Unicode aparecem à direita do operador `%` são descritas dom mais detalhes nesta seção.

3.1.3. Strings Unicode

A partir de Python 2.0 um novo tipo para armazenar textos foi introduzido: o tipo `unicode`. Ele pode ser usado para armazenar e manipular dados no padrão Unicode (veja http://www.unicode.org/) e se integra bem aos objetos string pré-existentes, realizando conversões automáticas quando necessário.

Unicode tem a vantagem de associar um único número ordinal a cada caractere, para todas as formas de escrita usadas em textos modernos ou antigos. Anteriormente, havia somente 256 números ordinais possíveis para identificar caracteres. Cada texto era tipicamente limitado a uma “code page” (uma tabela de códigos) que associava ordinais aos caracteres. Isso levou a muita confusão especialmente no âmbito da internacionalização de software (comumente escrito como `i18n` porque `internationalization` é `'i'` + 18 letras + `'n'`). Unicode resolve esses problemas ao definir uma única tabela de códigos para todos os conjuntos de caracteres.

Criar strings Unicode em Python é tão simples quanto criar strings normais:

```
>>> u'Hello World !'
u'Hello World !'
```

O `u` antes das aspas indica a criação de uma string Unicode. Se você desejar incluir caracteres especiais na string, você pode fazê-lo através da codificação *Unicode-Escape* de Python. O exemplo a seguir mostra como:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

O código de escape `\u0020` insere um caractere Unicode com valor ordinal 0x0020 (o espaço em branco) naquela posição.

Os outros caracteres são interpretados usando seus valores ordinais como valores ordinais em Unicode. Se você possui strings literais na codificação padrão Latin-1 que é usada na maioria dos países ocidentais, achará conveniente que os 256 caracteres inferiores do Unicode coincidem com os 256 caracteres do Latin-1.

Para os experts, existe ainda um modo *raw* da mesma forma que existe para strings normais. Basta prefixar a string com ‘ur’ para usar a codificação *Raw-Unicode-Escape*. A conversão `\uXXXX` descrita acima será aplicada somente se houver um número ímpar de barras invertidas antes do escape ‘u’.

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\\u0020World !'
```

O modo raw (cru) é muito útil para evitar o excesso de barras invertidas, por exemplo, em expressões regulares.

Além dessas codificações padrão, Python oferece todo um conjunto de maneiras de se criar strings Unicode a partir de alguma codificação conhecida.

A função embutida `unicode()` dá acesso a todos os codecs Unicode registrados (COders e DECoders). Alguns dos codecs mais conhecidos são: *Latin-1*, *ASCII*, *UTF-8*, e *UTF-16*. Os dois últimos são codificações de tamanho variável para armazenar cada caractere Unicode em um ou mais bytes. (N.d.T: no Brasil, é muito útil o codec *cp1252*, variante estendida do *Latin-1* usada na maioria das versões do MS Windows distribuídas no país, contendo caracteres comuns em textos, como aspas assimétricas `“x”` e `‘y’`, travessão `—`, bullet `•` etc.).

A codificação default é ASCII, que trata normalmente caracteres no intervalo de 0 a 127 mas rejeita qualquer outro com um erro. Quando uma string Unicode é exibida, escrita em arquivo ou convertida por `str()`, esta codificação padrão é utilizada.:

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal not in range(128)
```

Para converter uma string Unicode em uma string de 8-bits usando uma codificação específica, basta invocar o método `encode()` de objetos Unicode passando como parâmetro o nome da codificação destino. É preferível escrever nomes de codificação em letras minúsculas.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

Se você tem um texto em uma codificação específica, e deseja produzir uma string Unicode a partir dele, pode usar a função `unicode()`, passando o nome da codificação de origem como segundo argumento.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4. Listas

Python inclui diversas estruturas de dados *compostas*, usadas para agrupar outros valores. A mais versátil é *list* (lista), que pode ser escrita como uma lista de valores (itens) separados por vírgula, entre colchetes. Os valores contidos na lista não precisam ser todos do mesmo tipo.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Da mesma forma que índices de string, índices de lista começam em 0, listas também podem ser concatenadas, fatiadas e multiplicadas:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Todas as operações de fatiamento devolvem uma nova lista contendo os elementos solicitados. Isto significa que o fatiamento a seguir retorna uma cópia rasa (*shallow copy*) da lista:

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Diferentemente de strings, que são *imutáveis*, é possível alterar elementos individuais de uma lista:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Atribuição à fatias também é possível, e isso pode até alterar o tamanho da lista ou remover todos os itens dela:

```
>>> # Substituir alguns itens:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remover alguns:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Inserir alguns:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Inserir uma cópia da própria lista no início
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Limpar a lista: substituir todos os items por uma lista vazia
>>> a[:] = []
>>> a
[]
```

A função embutida `len()` também se aplica a listas:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

É possível aninhar listas (criar listas contendo outras listas), por exemplo:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # Veja a seção 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Observe que no último exemplo, `p[1]` e `q` na verdade se referem ao mesmo objeto! Mais tarde retornaremos a *semântica dos objetos*.

3.2. Primeiros passos rumo à programação

Naturalmente, podemos utilizar Python para tarefas mais complicadas do que somar 2+2. Por exemplo, podemos escrever o início da *sequência de Fibonacci* assim:

```
>>> # Sequência de Fibonacci:
... # a soma de dois elementos define o próximo
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Este exemplo introduz diversas características ainda não mencionadas.

- A primeira linha contém uma atribuição múltipla: as variáveis `a` e `b` recebem simultaneamente os novos valores 0 e 1. Na última linha há outro exemplo de atribuição múltipla demonstrando que expressões do lado direito são sempre avaliadas primeiro, antes da atribuição. As expressões do lado direito são avaliadas da esquerda para a direita.
- O laço `while` executa enquanto a condição (aqui: `b < 10`) permanecer verdadeira. Em Python, como em C, qualquer valor não-zero é considerado verdadeiro, zero é considerado falso. A condição pode ser ainda uma lista ou string, na verdade qualquer sequência; qualquer coisa com comprimento maior que zero tem valor verdadeiro e sequências vazias são falsas. O teste utilizado no exemplo é uma comparação simples. Os operadores padrão para comparação são os mesmos de C: `<` (menor que), `>` (maior que), `==` (igual), `<=` (menor ou igual), `>=` (maior ou igual) e `!=` (diferente).
- O *corpo* do laço é *indentado*: indentação em Python é a maneira de agrupar comandos em blocos. No console interativo padrão você terá que digitar tab ou espaços para indentar cada linha. Na prática você vai preparar scripts Python mais complicados em um editor de texto; a maioria dos editores de texto tem facilidades de indentação automática. Quando um comando composto é digitado interativamente, deve ser finalizado por uma linha em branco (já que o parser não tem como adivinhar qual é a última linha do comando). Observe que toda linha de um mesmo bloco de comandos deve ter a mesma indentação
- O comando `print` escreve o valor da expressão ou expressões fornecidas. É diferente de apenas escrever a expressão no interpretador (como fizemos nos exemplos da calculadora) pela forma como lida com múltiplas expressões e strings. Strings são exibidas sem aspas, e um espaço é inserido entre os itens para formatar o resultado assim:

  ```
  >>> i = 256*256
  >>> print 'O valor de i é', i
  O valor de i é 65536
  ```

  Uma vírgula ao final evita a quebra de linha:

  ```
  >>> a, b = 0, 1
  >>> while b < 1000:
  ...     print b,
  ...     a, b = b, a+b
  ...
  1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
  ```

  Note que o interpretador insere uma quebra de linha antes de imprimir o próximo prompt se a última linha não foi completada.