



Curso Programação Backend com Python

Material complementar

© Cleuton Sampaio 2021

Tipos de dados e variáveis	3
Numéricos	3
NaN	7
Lógicos	7
None	8
Caracteres	9
Tipos declarados pelo usuário (Objetos)	12
Verificar o tipo	15
Variáveis multivaloradas	15
Funções	27
Retorno	28
Parâmetros	28
Módulos	31
Pasta como módulo	34
Conteúdo de um módulo	35
Escopo de variáveis	38
Variável global Variável local	38
Alteração de variáveis globais	39
Escopo local superior	40

Tipos de dados e variáveis

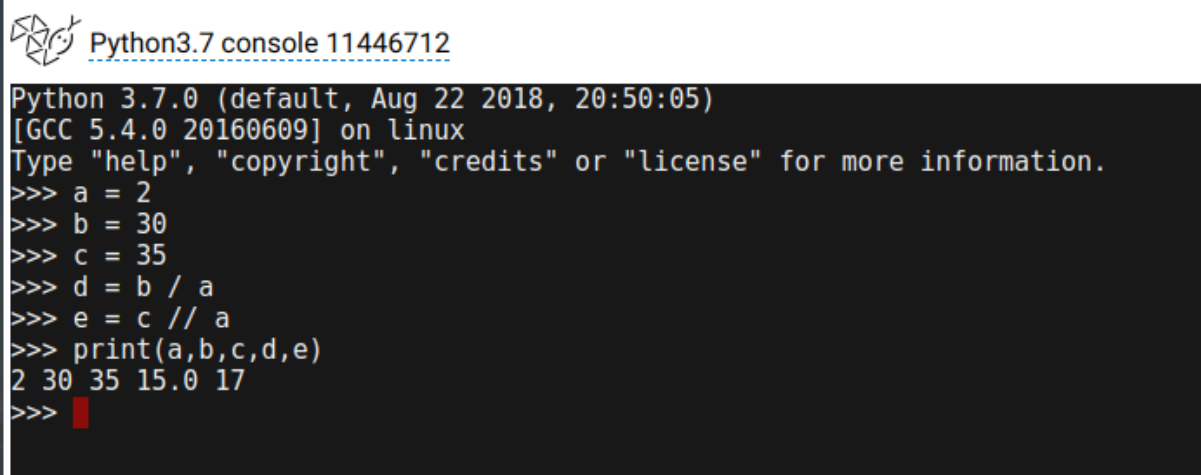
Numéricos

As variáveis em Python devem ser declaradas atribuindo-se um valor a um nome. Este valor

determina o tipo de dados da variável.

Abra um terminal (ou prompt de comandos) e vamos iniciar uma console Python, ou o Python interativo. Para isto, basta digitar: “python” (ou “python3”, se o seu computador tiver o Python 2.x instalado).

Este é o modo interativo do Python e você pode fazer quase tudo nele.



```
Python3.7 console 11446712
Python 3.7.0 (default, Aug 22 2018, 20:50:05)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 2
>>> b = 30
>>> c = 35
>>> d = b / a
>>> e = c // a
>>> print(a,b,c,d,e)
2 30 35 15.0 17
>>>
```

Essas são variáveis numéricas inteiras (conjunto \mathbb{Z}), pois o valor atribuído a elas é inteiro. Note algumas coisas interessantes:

1. Ao dividir b por a, o resultado é real (há um ponto decimal), independentemente das parcelas serem inteiras;
2. O operador “//” realiza uma divisão inteira, pegando o maior número inteiro que se aproxima do resultado da divisão de c por a, retornando um valor inteiro;
3. Quando separamos os argumentos por vírgulas no print(), eles podem ser de tipos diferentes e podem ser números. Os argumentos são convertidos em strings e separados por espaços automaticamente;

Sei que ainda não é a hora, mas vale a pena ver algumas das características do `print()` que podem ser úteis:

```
print(a + ' ' + b)
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

As variáveis `a` e `b` são numéricas e inteiras. Ao imprimir um string (você está concatenando as variáveis para formar um string) não é feita a conversão automática, e o Python não tem como concatenar as variáveis. Para fazer isso você deve converter os valores de `a` e `b` em string:

```
print(str(a) + ' ' + str(b))
```

2 30

Por esta razão é melhor utilizar a vírgula, em vez de criar um string para imprimir.

Outra coisa interessante é: Como fazer para imprimir vários números na mesma linha? Neste caso, entra o parâmetro “end” do `print()`:

```
for x in range(10):
```

```
...
```

```
    print(x, end=';')
```

```
...
```

```
0;1;2;3;4;5;6;7;8;9;
```

Neste caso, cada número será separado do outro por ponto e vírgula, sem pular linha.

Números reais

```
x = 7.89
```

```
z = 0.1928
```

```
print(x, z)
```

```
7.89 0.1928
```

As operações envolvendo números reais sempre resultarão em um número real:

```
print(a,x)
```

```
2 7.89
```

```
print(a + x)
```

```
9.89
```

```
print(int(a + x))
```

```
9
```

A função “int()” trunca o resultado, transformando-o em um valor inteiro. A função “float()” converte o resultado em um valor real:

```
a = int(2)
```

```
x = float(5)
```

```
print(a,x)
```

```
2 5.0
```

Não existe um limite para números inteiros, mas há um limite para o maior índice de uma lista, e pode ser obtido com a variável “sys.maxsize”:

```
import sys
```

```
print(sys.maxsize)
```

```
9223372036854775807
```

Este valor representa um número inteiro de 64 bits (não existe distinção entre int e long no Python 3). E podemos saber o maior número real:

```
print(sys.float_info.max)
```

```
1.7976931348623157e+308
```

Python também tem um literal para infinito positivo e negativo:

```
x = float('inf')
```

```
x
```

```
inf
```

```
z = float('-inf')  
z  
-inf
```

Octal, hexadecimal e binário

Em Python, um literal octal começa com “0o”:

```
x = 015  
File "<stdin>", line 1  
x = 015  
^  
SyntaxError: invalid token  
  
x = 15  
  
x = 0o15  
  
x  
13
```

Hexadecimais começam com “0x”:

```
z = 0xc10  
z  
3088
```

E, finalmente, binários começam com “0b”:

```
t = 0b01101  
t  
13
```

NaN

Assim como “inf”, Python tem um literal “NaN”, para representar um valor não numérico. O correto é que “NaN” é um valor que não pode ser representado como um número, geralmente, cálculos envolvendo infinito podem resultar em NaN. Mas divisão por zero resulta em exceção:

```
Python 3.8.5 (default, May 27 2021, 13:30:53)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> x = float("inf")
>>> z = float("inf")
>>> t = x - z
>>> t
nan
>>> 5/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Lógicos

Os valores True e False (assim mesmo, sem aspas e com a inicial maiúscula) definem respectivamente os valores Verdadeiro e Falso:

```
>>> b = True
>>> v = False
>>> print(b or v)
True
```

Podemos utilizar os operadores lógicos de conjunção e disjunção com variáveis ou valores lógicos. Podemos utilizar valores lógicos em condições:

```
>>> b = True
>>> print('B verdadeiro' if b else 'B falso')
B verdadeiro
```

Eu utilizei uma expressão lógica dentro do print(). É o equivalente ao “operador ternário”, existente em algumas linguagens.

Agora, veja só que interessante:

```
>>> c = 15
>>> print('C verdadeiro' if c else 'C falso')
C verdadeiro
>>> d = -4
>>> print('D verdadeiro' if d else 'D falso')
D verdadeiro
>>> e = 0
>>> print('E verdadeiro' if e else 'E falso')
E falso
>>> f = None
>>> print('F verdadeiro' if f else 'F falso')
F falso
```

Qualquer valor diferente de zero e de None é considerado como verdadeiro!

None

Preste atenção ao valor especial None. Alguns alegam que é igual ao null, existente em outras linguagens, mas não é exatamente assim. None significa que a variável não contém valor algum, o que é diferente de variável inexistente:

```
>>> print(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g' is not defined
>>> g = None
```



```
>>> print(g)
```

```
None
```

Caracteres

Podemos criar variáveis string utilizando sequências unicode delimitadas por aspas simples ou duplas:

```
>>> nome = "João"
```

```
>>> sobrenome = 'da Silva'
```

```
>>> print(nome, sobrenome)
```

```
João da Silva
```

```
>>> print(len(nome))
```

```
4
```

A função “len()” retorna o tamanho em caracteres de um string. Porém, os strings em Python são armazenados em Unicode e podemos utilizar UTF-8 para representar texto.

Strings podem ser concatenados com o operador ‘+’:

```
>>> nome_completo = nome + ' ' + sobrenome
```

```
>>> nome_completo
```

```
'João da Silva'
```

Existem várias operações que podemos realizar com strings, por exemplo:

fatiamento (slicing):

```
>>> sobrenome
```

```
'da Silva'
```

```
>>> sobrenome[0]
```

```
'd'
```

```
>>> sobrenome[1:]
```

```
'a Silva'
```

```
>>> sobrenome[-1]
'a'
>>> sobrenome[2:-2]
' Sil'
```

O fatiamento ocorre com dois índices: inicial e final, sendo que o final geralmente é exclusivo. O índice zero representa o primeiro caractere e o índice -1 representa o último (o -2 é o penúltimo). A fatia [1:] é do segundo caractere até o final, e a fatia [2:-2] representa do terceiro caractere até o penúltimo (sem incluí-lo).

Divisão (split):

```
>>> texto = 'minha terra tem palmeiras onde canta o sabiá'
>>> palavras = texto.split()
>>> palavras
['minha', 'terra', 'tem', 'palmeiras', 'onde', 'canta', 'o', 'sabiá']
```

O método “split()” separa os tokens de um texto, delimitados por um caractere, em elementos de uma lista. Se não informarmos nada, o caractere é o espaço. Outro exemplo:

```
>>> alunos = 'João,Maria,Pedro,Augusto,Eduarda,Clara'
>>> lista = alunos.split(',')
>>> lista
['João', 'Maria', 'Pedro', 'Augusto', 'Eduarda', 'Clara']
>>> for nome in lista:
...     print(nome)
...
João
Maria
Pedro
Augusto
Eduarda
Clara
```

Procura (find):

Podemos procurar um substring dentro de um string:

```
'João da Silva'
>>> nome_completo.find('da')
5
>>> nome_completo.find('da',1,10)
5
```

O método “find()” procura um substring dentro do string, podendo verificar apenas em um intervalo (posição inicial, posição final). Ele retorna a posição do substring dentro do texto ou então o valor -1, caso não encontre.

Substituição (replace):

Podemos substituir um substring por outro (independentemente de tamanho):

```
>>> nome_completo.replace(' da ',' Da ')
'João Da Silva'
>>> nome_completo
'João da Silva'
```

Note que a substituição não ocorre “inPlace”, ou seja, o string original permanece inalterado. Podemos especificar a quantidade de vezes que a substituição deve ocorrer (o default é tudo):

```
>>> texto = 'este * texto * contém * strings'
>>> print(texto.replace('*', '',2))
este texto contém * strings
```

Tipos declarados pelo usuário (Objetos)

Ainda veremos Classes e Objetos em uma aula mais à frente, porém, precisamos falar um pouco sobre variáveis de tipos definidos pelo usuário, ou classes, e a criação de objetos.

Quase todos os tipos de dados em python são Classes, e suas variáveis são objetos. Todos os tipos de dados possuem métodos e propriedades. Para acessar basta colocar um ponto (".") após o nome da variável ou do tipo. Vejamos isso na console interativa ou crie um arquivo python e execute:

```
a = 5
print(type(a))
print(a.to_bytes(8,byteorder='big'))
```

```
b = 7.5
print(type(b))
print(b.hex())
```

```
c = True
print(type(c))
print(c.real)
```

```
d = "minha terra tem palmeiras..."
print(type(d))
print(d.capitalize())
```

O resultado seria este:

```
python3 objetos.py
<class 'int'>
```

```
b'\x00\x00\x00\x00\x00\x00\x00\x05'  
  
<class 'float'>  
  
0x1.e000000000000p+2  
  
<class 'bool'>  
  
1  
  
<class 'str'>  
  
Minha terra tem palmeiras...
```

Como pode ver, cada variável é instância de uma classe específica, como “int”, “float” ou “str”, portanto são **objetos**. E, como objetos, possuem métodos e propriedades, como podemos ver no código e no resultado.

Não entraremos em detalhes das propriedades e métodos dos tipos intrínsecos (built-in) do python, mas você pode saber quais são estes métodos e propriedades, e o que eles fazem, aqui:

<https://docs.python.org/3/library/stdtypes.html#>

Muitos módulos que importamos possuem classes, as quais podemos instanciar. Veja este exemplo do projeto maze (que você já baixou, com certeza):

```
...  
  
from model.stack import Stack  
  
...  
  
pilha = Stack()  
  
...  
  
pilha.push(corrente)
```

Importamos uma classe chamada “Stack”, do módulo “model.stack” e instanciamos uma variável chamada **pilha** com ela. Para criarmos instâncias de classes em python não utilizamos algo como o método new do java:

Java	Python
<code>Stack pilha = new Stack();</code>	<code>pilha = Stack()</code>

Em python, usamos várias classes para criarmos variáveis, por exemplo: Vetores dinâmicos (List), Vetores associativos (Dictionary) e Conjuntos (Set), e todos possuem métodos e propriedades.

Verificar o tipo

Podemos utilizar a função “type()” para saber o tipo de dados de uma variável:

```
>>> print(type(nome_completo))
<class 'str'>
>>> print(type(a))
<class 'int'>
>>> print(type(x))
<class 'float'>
>>> b = True
>>> print(type(b))
<class 'bool'>
```

Variáveis multivaloradas

Em Python temos alguns tipos de variáveis multivaloradas:

- array: conjunto de elementos homogêneos, encapsulando um vetor C;
- list: conjunto de elementos heterogêneos;
- dictionary: lista de conjuntos chave-valor indexados;
- tuple: conjunto ordenado e imutável de elementos;
- set: conjunto de elementos únicos.

Vejamos alguns exemplos:

Array:

```
>>> import array as arr
>>> x = arr.array('f', [6.5, 7.0, 6.2, 8.5, 9.7])
>>> for nota in x:
...     print('nota: {}'.format(nota))
...
nota: 6.5
nota: 7.0
```

```
nota: 6.199999809265137
nota: 8.5
nota: 9.699999809265137
>>> print('média: {}'.format(sum(x) / len(x)))
média: 7.579999923706055
```

Note o uso do método “format()” no print(). Há outras opções que veremos mais adiante, mas cada conjunto de chaves corresponde a um valor passado no format().

Os elementos de um array devem ser do mesmo tipo, que é passado na sua declaração (primeiro argumento). Eis alguns tipos:

`'i' / 'l'`

Inteiro / Longo

`'f'`

Float (real)

`'d'`

Double (real)

`'B'`

Unsigned char (byte)

Usei o comando for para pegar cada elemento do array (diferentemente da função range()).

E se quiséssemos saber a ordem da nota? Poderíamos usar a opção de dois valores retornada pela função enumerate():

```
>>> for i,nota in enumerate(x):
...     print('Prova: {}, nota: {}'.format(i,nota))
...
Prova: 0, nota: 6.5
Prova: 1, nota: 7.0
Prova: 2, nota: 6.199999809265137
Prova: 3, nota: 8.5
```


Prova: 4, nota: 9.699999809265137

Em Python, uma função pode retornar mais de um valor. O primeiro valor retornado pela função `enumerate()` é a variável de controle, ou índice, e a segunda é o próprio valor.

Arrays nos permitem fatiá-los:

```
>>> x
array('f', [6.5, 7.0, 6.199999809265137, 8.5,
9.699999809265137])
>>> print(x[1:-2])
array('f', [7.0, 6.199999809265137])
```

Neste exemplo, pegamos do segundo elemento até o penúltimo (exclusive). Funciona como o fatiamento dos strings, que já vimos.

Podemos adicionar ou inserir elementos:

```
>>> x
array('f', [6.5, 7.0, 6.199999809265137, 8.5,
9.699999809265137])
>>> x.append(6.3)
>>> x
array('f', [6.5, 7.0, 6.199999809265137, 8.5,
9.699999809265137, 6.300000190734863])
array('f', [6.5, 7.0, 6.199999809265137, 8.5,
9.699999809265137, 6.300000190734863])
>>> x.insert(1,2.5)
>>> x
array('f', [6.5, 2.5, 7.0, 6.199999809265137, 8.5,
9.699999809265137, 6.300000190734863])
```

O `append()` adiciona elementos ao final do array e o `insert()` insere elementos antes da posição informada no primeiro argumento. Temos também o `extend()` que adiciona um array ao final de outro:

```
>>> x.extend([5.5,6.1,7.2])
>>> x
array('f', [6.5, 7.0, 6.199999809265137, 8.5,
9.699999809265137, 6.300000190734863, 5.5, 6.099999904632568,
7.199999809265137])
```

E podemos remover elementos:

```
>>> x
array('f', [6.5, 2.5, 7.0, 6.199999809265137, 8.5,
9.699999809265137, 6.300000190734863])
>>> del x[1]
>>> x
array('f', [6.5, 7.0, 6.199999809265137, 8.5,
9.699999809265137, 6.300000190734863])
```

List:

Array quase nunca é utilizado em Python, pois a maioria dos programadores prefere a lista (list). A principal diferença é que uma lista é composta por elementos heterogêneos:

```
>>> lista = ['banana',5,True,'Volvo',5.6]
>>> for elemento in lista:
...     print('Elemento: {}'.format(elemento))
...
Elemento: banana
Elemento: 5
Elemento: True
Elemento: Volvo
Elemento: 5.6
>>> for i,elemento in enumerate(lista):
...     print('Posição: {} valor: {}'.format(i,elemento))
```

```
...  
Posição: 0 valor: banana  
Posição: 1 valor: 5  
Posição: 2 valor: True  
Posição: 3 valor: Volvo  
Posição: 4 valor: 5.6
```

Basicamente existem as mesmas operações dos arrays, por exemplo, podemos anexar elementos ao final da lista:

```
>>> lista.append('Abacaxi')  
>>> lista  
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi']
```

Se quisermos anexar uma lista ao final, temos que tomar cuidado. Veja esse exemplo:

```
>>> lista  
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi']  
>>> lista.append(['Relógio',12.7])  
>>> lista  
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi', ['Relógio',  
12.7]]
```

O que aconteceu? Mandamos anexar uma lista ao último elemento da lista original, e ele fez isto! Uma lista é um objeto e o Python entendeu que deveríamos anexar este objeto à lista. Se quisermos estender uma lista, acrescentando vários elementos, temos que usar a função `extend()`:

```
>>> lista  
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi', ['Relógio',  
12.7]]  
>>> del lista[6]  
>>> lista  
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi']  
>>> lista.extend(['Relógio',12.7])
```

```
>>> lista
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi', 'Relógio', 12.7]
```

Removi o último elemento com o comando `del` e depois anexe com a função `extend()`, e o Python entendeu corretamente.

E podemos inserir elementos e listas utilizando o fatiamento. Veja só que loucura:

```
>>> lista
['banana', 5, True, 'Volvo', 5.6, 'Abacaxi', 'Relógio', 12.7]
>>> lista.insert(2, 'Novo')
>>> lista
['banana', 5, 'Novo', True, 'Volvo', 5.6, 'Abacaxi', 'Relógio', 12.7]
>>> lista[:2]=['uva',7]
>>> lista
['uva', 7, 'Novo', True, 'Volvo', 5.6, 'Abacaxi', 'Relógio', 12.7]
```

Eu inseri um elemento com valor “Novo” antes do elemento 2 (terceiro elemento), e depois, substituí os dois primeiros elementos (‘banana’ e 5) por: ‘uva’ e 7. Usei o fatiamento para isto. Do primeiro até o terceiro elemento exclusive.

Existem muito mais métodos em listas, por exemplo:

- `clear()`: Remove todos os elementos da lista;
- `copy()`: Retorna uma cópia da lista;
- `count()` : Retorna a quantidade de elementos na lista com o valor informado;
- `index()`: Retorna o índice na lista do elemento com o valor informado;
- `remove()`: Remove todos os elementos da lista com o valor informado;
- `reverse()`: Reverte a ordem dos elementos da lista;
- `sort()`: Classifica a lista.

Listas multidimensionais

Podemos representar vetores multidimensionais, ou matrizes, utilizando listas de listas. Por exemplo, suponha uma tabela de vendas por produto por ano por filial:

Ano	2016		2017		2018	
Filial	A	B	A	B	A	B
Centro	100	120	110	140	125	130
Sul	80	90	98	100	83	72

Podemos dizer que esta tabela tem 3 dimensões: Filial, Ano e Produto:

```
>>>
vendas=[[100,120],[110,140],[125,130]],[[80,90],[98,100],[83,
72]]]
>>> for filial in vendas:
...     for ano in filial:
...         for produto in ano:
...             print(produto)
...
100
120
110
140
125
130
80
90
98
100
83
72
>>> print(vendas[1][0][1])
90
```

Dictionary

Um dicionário é uma lista de pares formados por chave e valor:

```
>>> produtos = {'pizza': 50.00, 'calzone': 30.0,
'canoli':25.0}
>>> produtos
{'pizza': 50.0, 'calzone': 30.0, 'canoli': 25.0}
>>> produtos['calzone']
30.0
>>> for p in produtos:
...     print(p)
...
pizza
calzone
canoli
```

A chave de um elemento vem primeiro e depois vem o seu valor. Parece um objeto JSON (JavaScript Object Notation). Se quisermos pegar cada chave e seu correspondente valor, podemos fazer um loop assim:

```
>>> for produto,preco in produtos.items():
...     print('Produto: {}, preço: {}'.format(produto,preco))
...
Produto: pizza, preço: 50.0
Produto: calzone, preço: 30.0
Produto: canoli, preço: 25.0
```

O método `items()` retorna dois valores a cada passagem: A chave do item e o valor do item.

Para saber a quantidade de itens presentes em um dicionário, usamos a função `len()`. Neste caso, seria 3.

Para saber se um item existe no dicionário, podemos usar o formato “in”:

```
>>> if 'pizza' in produtos:
...     print('A pizza custa: {}'.format(produtos['pizza']))
...
A pizza custa: 50.0
```

Para adicionar e remover elementos é bem simples:

```
>>> produtos['gnocchi'] = 20.0
>>> produtos
{'pizza': 50.0, 'calzone': 30.0, 'canoli': 25.0, 'gnocchi': 20.0}
>>> produtos.pop('calzone')
30.0
>>> produtos
{'pizza': 50.0, 'canoli': 25.0, 'gnocchi': 20.0}
```

Basta atribuir um valor a uma chave inexistente e adicionamos um elemento. Para removê-lo, usamos o método “pop()” que também retorna seu valor.

Existem muito mais métodos para dicionários:

- `clear()`: Remove todos os elementos;
- `copy()`: Retorna uma cópia do dicionário;
- `fromkeys()`: Retorna um dicionário com as chaves especificadas;
- `get()`: Retorna o valor de uma chave;
- `items()`: Retorna uma lista contendo tuplas para cada elemento (chave e valor);
- `keys()`: Retorna uma lista com as chaves;
- `popitem()`: Remove o último par chave-valor inserido;
- `setdefault()`: Retorna o valor da chave especificada. Se a chave não existir, insere a chave e o valor informado no dicionário;

- `update()`: Atualiza o dicionário com os elementos informados;
- `values()`: Retorna uma lista de todos os valores no dicionário.

Tuple:

Uma tupla é um conjunto imutável de elementos. Funciona como uma lista, mas os elementos não podem ser alterados:

```
>>> cliente = ('Fulano de Tal','rua 5 número 7',500)
>>> cliente
('Fulano de Tal', 'rua 5 número 7', 500)
>>> cliente[2]
500
>>> cliente[-1]
500
>>> cliente[-1] = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> if 'Fulano de Tal' in cliente:
...     print('Endereço do Fulano: {}'.format(cliente[1]))
...
Endereço do Fulano: rua 5 número 7
```


Set:

Um set é um conjunto de elementos únicos, como um conjunto matemático:

```
>>> a = {'carro', 'moto', 'barco'}
>>> b = {'ônibus', 'carro', 'bicicleta'}
>>> print(a | b) # União
{'carro', 'ônibus', 'moto', 'bicicleta', 'barco'}
>>> print(a & b) # Interseção
{'carro'}
>>> print(a - b)
{'moto', 'barco'}
>>> print(b - a)
{'ônibus', 'bicicleta'}
```

Temos a união, interseção e diferença, ou seja, operações básicas de conjuntos. Note que na união, os elementos comuns não se repetem.

Um set é para operações de conjuntos, e não para acessarmos e mudarmos elementos individualmente. Portanto, não é possível acessar cada elemento como fazemos com listas. Uma maneira de acessarmos os elementos em um set é através de um loop:

```
>>> for tipo in (a | b):
...     print('O elemento é: {}'.format(tipo))
...
O elemento é: carro
O elemento é: ônibus
O elemento é: moto
O elemento é: bicicleta
O elemento é: barco
```

Outra maneira é com iterator:

```
>>> i = iter(a)
```

```
>>> for t in enumerate(i):
...     print(t)
...
(0, 'carro')
(1, 'barco')
(2, 'moto')
```

O iterador retorna uma tupla com a posição e o valor do elemento do set.

Finalmente, podemos utilizar o método “pop()”, porém ele retira o elemento do set:

```
>>> b
{'carro', 'ônibus', 'bicicleta'}
>>> b.pop()
'carro'
```

Podemos adicionar ou remover elementos do set, mas não podemos alterá-los. E nem podemos ter tipos mutáveis, como: list ou dictionary como elementos do set. Veja só este exemplo:

```
>>> a.add('avião')
>>> a
{'carro', 'barco', 'avião', 'moto'}
>>> a.discard('avião')
>>> a.discard('avião')
>>> a.remove('moto')
>>> a.remove('moto')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'moto'
```

Adicionar é simples! Basta usar o método add(). Para remover, temos dois métodos: discard() e remove(). A diferença é que, caso o elemento não exista no set, o remove retorna um erro e o discard não.

É claro que podemos testar a existência de um elemento com o “in”:

```
>>> b
{'ônibus', 'bicicleta'}
>>> if 'bicicleta' in b:
...     print('Sim')
...
Sim
```

E podemos saber a quantidade de elementos com o len():

```
>>> print(len(b))
2
```

Funções

Como toda linguagem de programação moderna, Python permite modularizar seu código-fonte através da criação de funções. São subrotinas que podem ou não retornar valores e podem ou não receber parâmetros. Declaramos funções com “def”:

```
>>> def PI():
...     return 22/7
...
>>> print(PI())
3.142857142857143
```

Todo o bloco de comandos que faz parte da função deve ser indentado. O padrão é indentar com quatro espaços. Note que ao final da declaração da função, deve haver um “:”.

Tenha ou não parâmetros (ou argumentos), é preciso informar os parêntesis na declaração e na invocação da função.

Retorno

Uma função pode ou não retornar um valor e podemos ou não ignorá-lo:

```
>>> def calcular(valor,taxa,prazo):
...     return valor + valor*(taxa/100)*prazo
...
>>> montante = calcular(1000.00,2.5,10)
>>> print(montante)
1250.0
```

Uma função pode retornar **mais de um valor**:

```
>>> def calcular(valor,taxa,prazo):
...     juros = valor * (taxa/100) * prazo
...     montante = valor + juros
...     return juros,montante
...
>>> j,m = calcular(1000.00,2.5,10)
>>> print('Juros: {}, Montante: {}'.format(j,m))
Juros: 250.0, Montante: 1250.0
```

Parâmetros

Os parâmetros podem ser referenciados de maneira posicional ou através de seu nome:

```
>>> j,m = calcular(1000.00,prazo=10,taxa=2.5)
>>> print('Juros: {}, Montante: {}'.format(j,m))
Juros: 250.0, Montante: 1250.0
```

Neste exemplo, invocamos a mesma função “calcular” passando o valor como parâmetro posicional (ele realmente fica na primeira posição, na ordem da função), mas invertemos a ordem dos parâmetros “taxa” e “prazo”, para isto, informamos seus nomes. Você pode informar parâmetros de maneira posicional ou não.

Também podemos criar parâmetros opcionais, informando um valor “default”:

```
>>> def calcular(valor,taxa,prazo=1):
...     juros = valor * (taxa/100) * prazo
...     montante = valor + juros
...     return juros,montante
...
>>> j,m = calcular(1000.00,2.5)
>>> print('Juros: {}, Montante: {}'.format(j,m))
Juros: 25.0, Montante: 1025.0
```

Ao atribuir um valor na declaração do parâmetro, você o torna opcional. Se um argumento é opcional, os seguintes a ele também devem ser. Por exemplo, vamos supor que a taxa tenha um valor default, mas o prazo não:

```
>>> def calcular(valor,taxa=2.5,prazo):
...     juros = valor * (taxa/100) * prazo
...     montante = valor + juros
...     return juros,montante
...
File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

Ele está reclamando que há um argumento (ou parâmetro) sem valor default, depois de um argumento que possui valor default. Argumentos com valor default são opcionais. Porém, você pode ter todos os parâmetros como opcionais:

```
>>> def calcular(valor=10,taxa=2.5,prazo=1):
...     juros = valor * (taxa/100) * prazo
...     montante = valor + juros
```

```
... return juros,montante
...
>>> print(calcular())
(0.25, 10.25)
```

Vamos supor que você queira informar apenas a taxa, deixando os outros dois argumentos com os valores default. Para isto, tem que usar a notação com nomes de parâmetros:

```
>>> print(calcular(taxa=1.2))
(0.12, 10.12)
```

Módulos

Um módulo Python é uma unidade de código, ou um arquivo-fonte, que expõe declarações para serem utilizadas por outros scripts. Entenda como uma “biblioteca” de código Python.

Vamos imaginar que você crie um conjunto de funções e queira utilizá-las em outros programas. Por exemplo, vamos imaginar que você tenha um jogo e precise criar um tabuleiro virtual bidimensional. Você quer mover objetos (retângulos) e saber se colidiram, mas tem que tomar cuidado com as bordas.

Baixe o arquivo de exemplos deste capítulo (exemplos.zip)

O código “geom.py” faz isto. Nele, há duas funções:

```
def colisao(objeto1,objeto2):
    if objeto1['x1']> objeto2['x2']or objeto2['x1']>
objeto1['x2']:
        return False
    if objeto1['y1'] < objeto2['y2'] or objeto2['y1'] <
objeto1['y2']:
        return False
    return True

def mover(objeto,direcao):
    novo=GABARITO_OBJETO.copy()
    novo['x1']=objeto['x1']+DIRECAO[direcao][0]
    novo['y1']=objeto['y1']+DIRECAO[direcao][1]
    novo['x2']=objeto['x2']+DIRECAO[direcao][0]
    novo['y2']=objeto['y2']+DIRECAO[direcao][1]
    if novo['x1']<0 or novo['x2']==LARGURA:
        return False
    if novo['y1']==ALTURA or novo['y2']<0:
        return False
```

```
objeto['x1']=novo['x1']
objeto['y1']=novo['y1']
objeto['x2']=novo['x2']
objeto['y2']=novo['y2']
return True
```

E temos um código imediato que serve para testar as funções. Agora, vamos tentar usar essa biblioteca dentro do nosso script. Crie um script assim:

```
import geom
#
# Game loop
#
# Criar objeto:
player=geom.GABARITO_OBJETO.copy()

# Mover objeto:
retorno = geom.mover(player,'cima')
if not retorno:
    print('erro')
```

Perfeito, não? Importamos todas as declarações de “geom” em “jogo”, e podemos usar o prefixo “geom” como namespace. Agora, tente executar o jogo:

```
python jogo.py
True
False
Moveu objeto1 para a direita: {'x1': 1, 'y1': 10, 'x2': 11,
'y2': 0}
Moveu objeto3 para cima: {'x1': 15, 'y1': 16, 'x2': 20, 'y2':
11}
Moveu objeto3 para esquerda: {'x1': 14, 'y1': 16, 'x2': 19,
'y2': 11}
Moveu objeto3 para baixo: {'x1': 14, 'y1': 15, 'x2': 19,
'y2': 10}
Não pode mover para a direita: {'x1': 989, 'y1': 999, 'x2':
999, 'y2': 989}
```



```
Não pode mover para cima: {'x1': 989, 'y1': 999, 'x2': 999, 'y2': 989}
```

```
Não pode mover para a esquerda: {'x1': 0, 'y1': 10, 'x2': 10, 'y2': 0}
```

```
Não pode mover para baixo: {'x1': 0, 'y1': 10, 'x2': 10, 'y2': 0}
```

Ué?! Por que ele mostrou essas linhas? Quando importamos um módulo, o seu código imediato é executado. Serve como um “startup”, caso ele seja invocado diretamente pelo Python. Para evitarmos isto, usamos uma variável dunder especial chamada: “__name__”. Note que dentro de “geom.py” eu inseri o comando:

```
print(__name__)
```

Quando executamos diretamente o script “geom.py”, ele mostra: “__main__”, que é o módulo principal da aplicação, criado diretamente pelo interpretador Python.

Porém, quando executamos “jogo.py”, ele mostra “geom”, que é o nome do módulo “geom.py”, importado a partir do “jogo.py”.

Moral da história: O módulo que foi executado diretamente pelo Python passa a ser o “__main__”, o que podemos testar na variável “__name__”. Então, podemos colocar um “if” dentro de “geom.py”, de modo a só executarmos o código imediato se ele for o módulo “__main__”:

```
if __name__ == '__main__':  
    print(__name__)  
    objeto1=GABARITO_OBJETO.copy()  
    objeto2=GABARITO_OBJETO.copy()  
    objeto3=GABARITO_OBJETO.copy()  
...
```

O arquivo “geom_lib.py” já está assim. E modificamos também o arquivo “jogo.py”, criando o “jogo_novo.py”:

```
import geom_lib as geom
#
# Game loop
#
# Criar objeto:
player=geom.GABARITO_OBJETO.copy()

# Mover objeto:
retorno = geom.mover(player, 'cima')
if not retorno:
    print('erro')
```

Usei uma forma diferente do import que permite renomear o namespace. Assim, não tive que alterar o resto do código: “import biblioteca as nome”.

Pasta como módulo

Seu módulo não precisa ficar na mesma pasta. Na verdade, você pode criar pastas diferentes, dependendo do tipo de módulo que vai importar. Por exemplo, veja a pasta “utils”, que contém uma versão do “geom_lib.py” e veja o script “jogo_pasta.py” que a importa:

```
import utils.geom_lib as geom
#
# Game loop
#
# Criar objeto:
player=geom.GABARITO_OBJETO.copy()

# Mover objeto:
retorno = geom.mover(player, 'cima')
if not retorno:
    print('erro')
```

Agora, temos um módulo “utils” e um submódulo “geom_lib”. Para não mudar o código do jogo, eu usei a sintaxe “import as...”.

Note que dentro da pasta “utils” há um script “__init__.py”, que serve para prevenir conflito entre os nomes de pastas e os pacotes já existentes. Mas também serve para executar código de inicialização de um pacote. Geralmente, “__init__.py” fica vazio, mas eu coloquei um print() só para mostrar.

Conteúdo de um módulo

A função dir() exibe todos os elementos de um módulo importado, que estão disponíveis para uso. Por exemplo, dentro do Python interativo importei o módulo e rodei o comando:

```
>>> import utils.geom_lib
Rodou init
>>> dir(utils.geom_lib)
['ALTURA', 'DIRECAO', 'GABARITO_OBJETO', 'LARGURA',
 '__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__',
 'colisao', 'mover', 'objetos']
```

Note que importei sem a opção “import as...”, portanto, tenho que me referir ao módulo com seu nome completo: utils.geom_lib. Alguns elementos eu mesmo criei, outros, o Python criou.

Há uma terceira forma de importar elementos de um módulo, que nos permite filtrar o que queremos importar: “from <módulo> import elemento [,outro elemento, etc]”.

Veja um exemplo:

```
>>> from utils.geom_lib import mover
Rodou init
>>> dir(mover)
['__annotations__', '__call__', '__class__', '__closure__',
 '__code__', '__defaults__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
```

```
'__get__', '__getattr__', '__globals__', '__gt__',  
'__hash__', '__init__', '__init_subclass__', '__kwdefaults__',  
'__le__', '__lt__', '__module__', '__name__', '__ne__',  
'__new__', '__qualname__', '__reduce__', '__reduce_ex__',  
'__repr__', '__setattr__', '__sizeof__', '__str__',  
'__subclasshook__']
```

```
>>> dir(utils.geom_lib)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'utils' is not defined
```

```
>>> from utils.geom_lib import mover,colisao
```

```
>>> dir(utils.geom_lib)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'utils' is not defined
```

Neste exemplo começamos importando só a função mover(). Note que nada mais de utils.geom_lib está disponível. Depois, importamos as funções: mover() e colisao().

Podemos importar todos os elementos de um módulo de duas maneiras:

- import <módulo>
- from <módulo> import *

A primeira maneira importa tudo e coloca os elementos dentro do namespace do módulo (a não ser que você use a opção “as <nome>”). A segunda, importa também todos os elementos, mas coloca no namespace do módulo principal. Veja só:

```
>>> from utils.geom_lib import *
```

```
Rodou init
```

```
>>> dir(utils.geom_lib)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
NameError: name 'utils' is not defined
```

```
>>> dir()
['ALTURA', 'DIRECAO', 'GABARITO_OBJETO', 'LARGURA',
 '__annotations__', '__builtins__', '__doc__', '__loader__',
 '__name__', '__package__', '__spec__', 'colisao', 'mover',
 'objetos']
```

O namespace `utils.geom_lib` nem existe no módulo principal, e todos os seus elementos estão diretamente ligados ao módulo principal (`dir()` mostra todos os elementos do módulo principal). Podemos utilizá-los diretamente. Agora, da outra maneira é diferente:

```
>>> import utils.geom_lib
Rodou init
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__',
 '__name__', '__package__', '__spec__', 'utils']
>>> dir(utils)
['_all_', '__builtins__', '__cached__', '__doc__',
 '_file_', '__loader__', '__name__', '__package__',
 '_path_', '__spec__', 'geom_lib']
>>> dir(utils.geom_lib)
['ALTURA', 'DIRECAO', 'GABARITO_OBJETO', 'LARGURA',
 '__builtins__', '__cached__', '__doc__', '__file_',
 '__loader__', '__name__', '__package__', '__spec__',
 'colisao', 'mover', 'objetos']
```

Agora, os elementos não estão no módulo principal, exceto o módulo `utils`. E, ao listarmos `utils`, encontramos `geom_lib`. Todos os seus elementos podem ser acessados a partir do namespace `utils.geom_lib`.

Se você examinar o código de `utils.geom_lib`, notará uma função cujo nome inicia por sublinha:

```
def _privfunc():
    pass
```

Qualquer elemento cujo nome inicie por sublinha é considerado privado e não é importado. Note também o comando “pass”, que não executa operação alguma. É só para guardar espaço.

Podemos controlar o que será importado utilizando a variável “__all__” declarada dentro do “__init__.py” de um módulo:

```
__all__=['mover','colisao']
```

Se quisermos permitir a importação de elementos privados, basta acrescentar à lista da variável “__all__”.

Escopo de variáveis

Em Python temos o escopo global (do módulo) e o local (de função):

```
>>> glob='Variável global'
>>>
>>> def funcao():
...     local='Variável local'
...     print(glob,local)
...
>>> funcao()
```

Variável global Variável local

Mas, ao contrário de outras linguagens, como Java, não existe o escopo de bloco de comandos. Veja este exemplo:

```
>>> glob='Variável global'
>>>
>>> def funcao():
...     local='Variável local'
...     while True:
```

```
...     var_bloco='Var bloco'
...     break
...     print(glob,local,var_bloco)
...
>>> funcao()
```

Não existe escopo de bloco de comandos

Eu criei uma variável chamada `var_bloco` dentro do `while`, mas ela continuou a existir depois de sair do bloco de código.

Alteração de variáveis globais

Python tem suas idiossincrasias. Uma delas é a questão das variáveis globais. Me diga o que faz este código:

```
>>> achou=False
>>> def procurar(texto):
...     for c in texto:
...         if c=='*' or c=='&':
...             achou=True
...
>>> procurar('Este * um t&xto legal')
>>> print(achou)
False
```

Ele deveria encontrar os caracteres dentro do texto, não? O que aconteceu?

Na verdade, criamos uma variável global “achou” (eu sei, essa é uma decisão ruim, mas é para exemplificar). E a tornamos verdadeira quando a função encontra algum dos caracteres. Só que as variáveis globais só ficam disponíveis para leitura. Ao tentar alterar uma variável global dentro de uma função, o Python cria uma variável local com o mesmo nome, evitando alterar a variável global. É um mecanismo de defesa. Se quisermos alterar a variável global dentro da função, demos que usar a declaração global:

```
>>> achou=False
>>> def procurar(texto):
...     global achou
...     for c in texto:
...         if c=='*' or c=='&':
...             achou=True
...
>>> procurar('Este * um t&xto legal')
>>> print(achou)
True
```

De qualquer forma, essa é uma prática ruim (alterar variáveis globais dentro de funções). Nossas funções devem ser “puras”, ou seja, seu resultado deveria ser determinado apenas pelos parâmetros que recebe e não deveriam alterar nada que esteja fora de seu escopo:

<https://pt.stackoverflow.com/questions/255557/o-que-%C3%A9-uma-fun%C3%A7%C3%A3o-pura>

Escopo local superior

O Python 3 incluiu a declaração **nonlocal** para permitir alteração de variáveis locais de escopo superior. Vou demonstrar isso com um exemplo do Stackoverflow (<https://stackoverflow.com/questions/1261875/python-nonlocal-statement>):

```
x = 0
def outer():
    x = 1
    # Esta é uma função declarada dentro de outra função:
    def inner():
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)
```



```
outer()
print("global:", x)

# inner: 2
# outer: 1
# global: 0
```

Nem o “x” global nem o “x” do escopo da função “outer()” foram alterados. Para permitir que a função “inner()” altere o valor de “x” no escopo da função “outer()”, precisamos incluir a declaração nonlocal:

```
x = 0
def outer():
    x = 1
    def inner():
        nonlocal x
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)

# inner: 2
# outer: 2
# global: 0
```