



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
Verificar o tipo	12
Variáveis multivaloradas	12
Funções	24
Retorno	25
Parâmetros	25
Módulos	28
Pasta como módulo	31
Conteúdo de um módulo	32
Escopo de variáveis	35
Variável global Variável local	35
Alteração de variáveis globais	36
Escopo local superior	37
Classes e Objetos (OOP)	39
Objetos	39
Classes	43
Cópias	45
Shallow e Deep copy	46
Variáveis de instância	47
Métodos de instância	49
Métodos estáticos e de classe	50
Controle de acesso	50
Herança e polimorfismo	53
Polimorfismo	55
Classes abstratas e interfaces	57
Dunders	58

Tipos de dados e variáveis

Numéricos

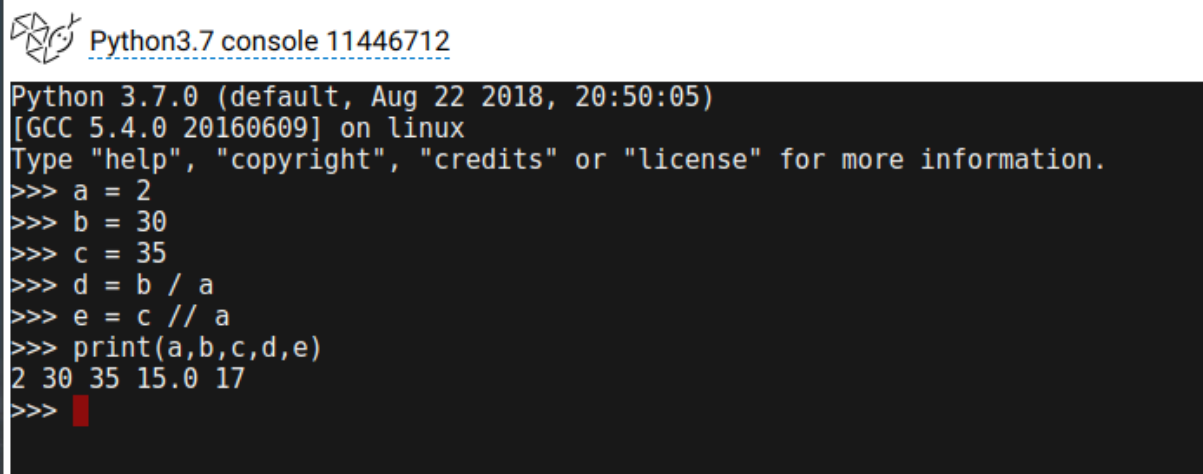
As variáveis em Python devem ser declaradas atribuindo-se um valor a um nome.

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
```

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
```

Classes e Objetos (OOP)

Orientação a objetos é uma forma de modularizar seu programa, muito popular hoje em dia. Se você não conhece OOP (Object-oriented programming) em nenhuma outra linguagem, é melhor estudar o assunto antes de continuar:

<https://www.devmedia.com.br/principais-conceitos-da-programacao-orientada-a-objetos/32285>

Se você já possui experiência com OOP em outras linguagens, como Java ou C#, pode ter algumas surpresas. Em Python, a declaração e instanciamento de classes é bastante diferente.

Objetos

Geralmente falando, em Python, todos os tipos de dados são classes, portanto, suas variáveis são objetos:

```
>>> a = 1
>>> type(a)
<class 'int'>
>>> b = True
>>> type(b)
<class 'bool'>
>>> c = 'Teste'
>>> type(c)
<class 'str'>
>>> print(a.bit_length())
1
>>> print(a.to_bytes(2, byteorder='big'))
b'\x00\x01'
```

Como pode ver, os tipos das variáveis "a", "b" e "c" são classes. E as variáveis são objetos, possuindo métodos, como pode ver nos dois últimos comandos.

O que acontece quando atribuímos um valor a um objeto? Se for um objeto imutável, o python verificará se já existe aquele objeto em memória e, neste caso, apenas atribui o mesmo endereço à variável:

```
>>> nome="fulano"
>>> id(nome)
140098638964640
>>> nome2="fulano"
>>> id(nome2)
140098638964640
```

A função "id()" retorna o endereço de um objeto na memória. Note que o endereço das duas variáveis é o mesmo, pois ambas apontam para o mesmo objeto imutável. Se criarmos outro objeto string, este endereço muda:

```
>>> nome3="beltrano"
>>> id(nome3)
140098669530800
```

Bom, se o objeto é o mesmo e se duas variáveis apontam para o mesmo objeto, é de se esperar que, ao alterarmos o objeto, ambas reflitam o mesmo valor, certo?

```
>>> nome="fulano"
>>> id(nome)
140098638964640
>>> nome2="fulano"
>>> id(nome2)
140098638964640

>>> nome="cicrano"
>>> nome2
'fulano'
```


Algo errado? Era de se esperar que, ao alterarmos o objeto apontado por nome, a variável nome2 fosse afetada. Na verdade, strings são objetos imutáveis, logo, se alterarmos o conteúdo de uma variável string, estamos, na verdade, criando outro objeto e apontando para este. Se havia outra variável apontando para o objeto original, não será afetada.

Isto também acontece com inteiros:

```
>>> numero = 5
>>> outro_numero = numero
>>> id(numero)
94711138497440
>>> id(outro_numero)
94711138497440
>>> numero = 6
>>> outro_numero
5
>>> id(numero)
94711138497472
>>> id(outro_numero)
94711138497440
```

Com outros tipos de variáveis, como números reais, este comportamento pode ser diferente:

```
>>> numero_real = 50.05
>>> id(numero_real)
140098670363296
>>> outro_ainda = 50.05
>>> id(outro_ainda)
140098670362792
>>> a = 5
```

```
>>> b = 5
>>> id(a)
94711138497440
>>> id(b)
94711138497440
```

Note que o comportamento com objetos que são números reais é diferente do comportamento com objetos que são inteiros.

Este comportamento de objetos imutáveis é diferente para cada implementação de Python (CPython, PyPy ou Jython).

Com objetos mutáveis, podemos criar cópias independentes. Por exemplo, vamos supor uma list:

```
>>> numeros = [1,2,3,4,5]
>>> outros_numeros = [1,2,3,4,5]
>>> id(numeros)
140098669581512
>>> id(outros_numeros)
140098638878472
```

Note que, apesar do valor atribuído ser o mesmo, dois objetos diferentes foram criados na memória. E o que acontece se atribuirmos outra variável a um objeto mutável?

```
>>> numeros = [1,2,3,4,5]
>>> id(numeros)
140098669581512
>>> lista1 = numeros
>>> id(lista1)
140098669581512

>>> numeros[-1]=10
```

```
>>> lista1  
[1, 2, 3, 4, 10]
```

Criamos uma variável `lista1`, atribuindo a ela o objeto `numeros`. O que o Python fez? Usou o mesmo objeto! Tanto é verdade, que, ao alterarmos o último elemento da lista `numeros`, o último elemento da lista `lista1` também foi alterado! Quando o objeto é mutável, o Python não faz uma cópia, apenas adiciona um "apelido" para o mesmo objeto!

Lembre-se sempre disto!

Classes

Podemos criar nossas próprias classes (ou tipos de dados), e instanciarmos objetos a partir delas. Por exemplo:

```
>>> class MinhaClasse:  
...     pass  
...  
>>> var1 = MinhaClasse()  
>>> type(var1)  
<class '__main__.MinhaClasse'>
```

A sintaxe é simples:

```
class <nome da classe>:  
    <variáveis da classe>  
    <métodos>
```

Você sabe a diferença entre classe e instância, certo? Sabe o que são variáveis de classe e variáveis de instância, certo? Vou mostrar uma coisa:

```
>>> class Carro:  
...     eixos=2  
... 
```

```
>>> fusca = Carro()
>>> fusca.marca = "Volkswagen"
>>> fusca.eixos
2
>>> fusca.marca
'Volkswagen'
>>> focus = Carro()
>>> focus.marca = "Ford"
>>> focus.eixos
2
>>> focus.marca
'Ford'
>>> Carro.eixos
2
```

Toda variável declarada dentro de uma classe é considerada como variável de classe (ou "static" em Java). No exemplo, a variável `eixos` é da classe `Carro`, portanto, só existe uma única cópia dela em memória.

As variáveis atribuídas aos objetos individuais (`fusca`, `focus`) são variáveis de instância, e cada objeto tem seu próprio valor para elas. Na verdade, objetos diferentes da mesma classe, podem possuir propriedades com nomes diferentes.

Eu atribuí a nova propriedade `marca` aos dois objetos que criei, com valores diferentes. Cada objeto possui sua `marca`, mas a classe `Carro` não possui `marca`:

```
>>> fusca.marca
'Volkswagen'
>>> focus.marca
'Ford'
>>> Carro.marca
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Carro' has no attribute 'marca'
```

Como já deve ter visto, instanciamos classes atribuindo uma variável ao nome da classe:

```
>>> fusca = Carro()
>>> id(fusca)
140098638965896
>>> focus = Carro()
>>> id(focus)
140098638965672
>>> id(Carro)
94711172186904
>>> volks = fusca
>>> id(volks)
140098638965896
```

Não há necessidade de utilizar um operador new, como em Java. Se estivermos instanciando um objeto mutável, uma nova instância será criada na memória. Mas, se estivermos atribuindo um objeto já existente, então estamos apenas criando um novo "apelido" para ele, que é o caso das variáveis fusca e volks. Note que a própria classe Carro tem seu endereço de memória separado.

Posso estar sendo chato e redundante, mas é importantíssimo que você entenda como funciona atribuição de objetos em Python e quais as diferenças para outras linguagens de programação, como Java, por exemplo.

Cópias

E se quisermos criar uma cópia de um objeto em memória?

```
>>> fusca = Carro()
>>> id(fusca)
140098638965896
>>> volks = fusca
```

```
>>> id(volks)
140098638965896
```

Neste exemplo, a variável `volks` aponta para o mesmo objeto da variável `fusca`. E se eu quisesse criar uma cópia, ou seja um novo objeto, mas com os mesmos valores do `fusca`? Uma opção é importar o módulo `copy` e utilizar o seu método `copy`:

```
>>> fusca.marca = "Volkswagen"
>>> volks = copy.copy(fusca)
>>> id(fusca)
140098638965896
>>> id(volks)
140098638965952
>>> fusca.marca
'Volkswagen'
>>> volks.marca
'Volkswagen'
>>> fusca.marca = "New beetle"
>>> volks.marca
'Volkswagen'
```

Note que eu fiz uma cópia do objeto, tanto é que posso alterar a propriedade `marca` do `fusca`, que o objeto `volks` permanecerá sem alterações.

Shallow e Deep copy

Python tem suas idiossincrasias! Há dois tipos de cópias de objetos: **Shallow** e **Deep**. O método `copy.copy()` executa uma **shallow** copy. Isso é melhor demonstrado com objetos que possuem outros objetos, como listas:

```
>>> lista1 = [[1,2,3],[4,5,6]]
>>> lista2 = copy.copy(lista1)
>>> id(lista1)
140098639081800
```

```
>>> id(lista2)
140098639081992
>>> lista2[0][0]=33
>>> lista2
[[33, 2, 3], [4, 5, 6]]
>>> lista1
[[33, 2, 3], [4, 5, 6]]
```

Eu usei o método `copy.copy()` e criei dois objetos diferentes! Os ids são diferentes! Como é possível eu alterar a `lista2` e a `lista1` ser afetada?

O método `copy.copy()` executa uma shallow copy (cópia rasa), procurando reutilizar os objetos internos de uma lista. Se quisermos realmente criar uma cópia profunda, temos que usar o método `copy.deepcopy()`:

```
>>> lista1 = [[1,2,3],[4,5,6]]
>>> lista2 = copy.deepcopy(lista1)
>>> lista2[0][0]=33
>>> lista2
[[33, 2, 3], [4, 5, 6]]
>>> lista1
[[1, 2, 3], [4, 5, 6]]
```

Variáveis de instância

Se quisermos criar uma classe com variáveis de instância, precisamos de um método construtor. Isto é feito com o método `__init__` (script "carro.py"):

```
class Carro:
    eixos = 2
    def __init__(self,marca=None):
        self.marca = marca
```

E podemos importar e instanciar este tipo:

```
>>> from carro import Carro
>>> volks = Carro("Volkswagen")
>>> print('Carro: {} eixos {}'.format(volks.marca,
volks.eixos))
Carro: Volkswagen eixos 2
```

O método `__init__` é invocado sempre que instanciamos uma classe. Note que ele possui um primeiro parâmetro posicional, tradicionalmente nomeado `self`, que é uma referência ao próprio objeto que está sendo criado/acessado. Usando esta referência, podemos atribuir variáveis a esta instância. Assim, criamos variáveis de instância em classes Python.

O parâmetro `self` é declarado no método, mas não é passado quando instanciamos o objeto! O Python passa automaticamente! O que acontece se não especificarmos o parâmetro `self`, ao declaramos o método? Bom, se ele contiver um parâmetro, receberá automaticamente a referência ao objeto. Caso contrário, ele será considerado um método da classe Carro:

```
class Carro:
    eixos = 2
    def __init__(self,marca=None):
        self.marca = marca
    def metodo():
        print('Chamou')
```

```
>>> from carro import Carro
>>> volks = Carro()
>>> volks = Carro('volkswagen')
>>> volks.metodo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: metodo() takes 0 positional arguments but 1 was
given
>>> Carro.metodo()
Chamou
```


Note que lambança! Se tentarmos invocar o método "metodo()" a partir do objeto, ele reclama que está faltando o parâmetro self. Mas podemos invoca-lo a partir da classe.

Métodos de instância

Um método de instância pertence à instância da classe e pode acessar as variáveis de instância usando o parâmetro self:

```
class Carro:
    eixos = 2
    def __init__(self, marca=None):
        self.marca = marca
    def dict(self):
        return {'marca': self.marca}
```

É um método que possui o primeiro argumento, que pode ou não ser chamado self, mas que sempre receberá o objeto a partir do qual o método foi invocado. Ele pode acessar as variáveis de instância. Eis o resultado:

```
>>> from carro_1 import Carro
>>> var1 = Carro("Ford")
>>> type(var1.dict())
<class 'dict'>
>>> print(var1.dict())
{'marca': 'Ford'}
```

O método dict retorna um dictionary com a propriedade marca da instância.

Métodos estáticos e de classe

O Python faz distinção entre métodos estáticos e métodos de classe. Nós os declaramos utilizando um decorator (uma declaração iniciada por "@"):

```
class Carro:
    eixos = 2
    def __init__(self, marca=None):
        self.marca = marca
    @staticmethod
    def comentario():
        return "Esta é a classe carro."
    @classmethod
    def tipoVeiculo(cls):
        return "Este veículo tem {} eixos".format(cls.eixos)
```

O decorator `@staticmethod` declara um método estático, independente, que não tem acesso a nada da classe ou da instância. O decorator `@classmethod` recebe um parâmetro para a classe e pode ter acesso a qualquer variável ou método desta classe.

Controle de acesso

Python não tem um controle de acesso muito forte, como Java (default, private, protected e public). A princípio, todas as variáveis e métodos são públicos, ou seja: Podem ser acessados a partir de qualquer código.

Há uma convenção em Python para tratar variáveis e métodos como privados: Devem começar com um único caractere sublinha ("_"):

```
class Carro:
    eixos = 2
    def __init__(self, marca=None, chassi=None):
```

```
        self._chassi = chassi
        self.marca = marca
    def _mostrar(self):
        return self._chassi
```

Vamos ver como utilizar esta classe:

```
>>> from carro_3 import Carro
>>> volks = Carro("Volkswagen", "DABB01")
>>> volks.marca
'Volkswagen'
>>> volks._chassi
'DABB01'
>>> volks._mostrar()
'DABB01'
```

A variável `_chassi` e o método `_mostrar()` não fazem parte da API pública da classe, logo, não deveriam ser acessada por código externo. Nós os acessamos sem nenhuma limitação. Como eu disse, é apenas uma convenção, ou seja, você não deveria utilizar membros cujo nome inicia por sublinha, pois não fazem parte da API pública da classe.

Quando pensamos em variáveis de classe privadas, podemos ter problemas, pois é preciso separar as variáveis de cada subclasse. Uma maneira de esconder variáveis de classe privadas é prefixando-as com dois caracteres sublinha ("`__`");

```
class Carro:
    eixos = 2
    __quantidade = 0
    def __init__(self, marca=None):
        Carro._Carro__quantidade+=1
        self.marca = marca
    @classmethod
    def contagem(cls):
```

```
        return "existem  
{0}veiculos".format(cls.__Carro__quantidade)
```

Se prefixarmos um membro com dois caracteres sublinha, então a variável é renomeada como:

```
_<nome da classe>__<nome da variável>
```

Veja só como utilizamos:

```
>>> from carro_4 import Carro  
>>> fusca = Carro("Volkswagen")  
>>> Carro.contagem()  
'existem 1 veiculos'  
>>> fusca.__quantidade  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'Carro' object has no attribute '__quantidade'  
>>> Carro.__quantidade  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: type object 'Carro' has no attribute  
'__quantidade'  
>>> Carro.__Carro__quantidade  
1
```

Continuamos a poder acessar a variável, se a prefixarmos com `_Carro`. O objetivo não é esconder a variável, mas evitar conflitos de nomes com subclasses.

Herança e polimorfismo

Como toda linguagem orientada a objetos, Python oferece esses dois mecanismos. Vamos ver o script "veiculo.py":

```
class Veiculo:
    rodas = 4
    __contagem = 0
    def __init__(self, marca=None):
        self.marca = marca
    @classmethod
    def quemSou(cls):
        return(cls)
    def mostrar(self):
        return {'marca': self.marca}

class Carro(Veiculo):
    pass
```

A classe Veiculo tem variáveis de classe públicas, privadas, métodos de classe etc. A classe Carro herda de Veiculo. São herdados: Construtor, variáveis públicas de classe, variáveis privadas de instância e os métodos.

```
>>> from veiculo import Veiculo, Carro
>>> v = Veiculo("Volkswagen")
>>> c = Carro("fusca")
>>> c.quemSou()
<class 'veiculo.Carro'>
>>> c._Veiculo__contagem
0
>>> c._Carro__contagem
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Carro' object has no attribute
'_Carro__contagem'
```

Veja que a classe Carro não declarou um construtor, porém, ele existe, pois foi herdado. A princípio, tudo o que você declarar na classe derivada, não será herdado da classe original. Como Carro não tem um `__init__` ele herda o da classe Veiculo (junto com quaisquer variáveis que instancie com self, dentro do `__init__`).

Carro não tem uma variável privada de classe chamada `__contagem`. Mas ele herda a de Veiculo. Vamos modificar um pouco para mostrar como ficaria:

```
class Veiculo:
    rodas = 4
    __contagem = 100
    def __init__(self, marca=None):
        self.marca = marca
    @classmethod
    def quemSou(cls):
        return(cls)
    def mostrar(self):
        return {'marca': self.marca}

class Carro(Veiculo):
    __contagem = 50
```

Veja só agora:

```
>>> from veiculo import Veiculo, Carro
>>> c = Carro("fusca")
>>> v = Veiculo("Volkswagen")
>>> c._Carro__contagem
50
>>> v._Veiculo__contagem
100
```

Confuso, não? Mas o que vai te confundir mesmo é a Herança múltipla. Vou mostrar apenas um pequeno exemplo:

```
class Veiculo():
    rodas = 4
    def __init__(self, marca=None):
        self.marca = marca

class Automotor():
    cilindradas = 1000

class Carro(Veiculo, Automotor):
    def mostrar(self):
        return {'marca':self.marca, 'cilindradas':self.cilindradas}
```

Como pode ver, a classe Carro deriva de duas classes bases: Veiculo e Automotor, portanto herda membros das duas:

```
>>> from veiculo2 import *
>>> c = Carro()
>>> c.mostrar()
{'marca': None, 'cilindradas': 1000}
```

Usei a sintaxe * para importar tudo do módulo veiculo2, e a classe carro herdou as propriedades marca e cilindradas.

Polimorfismo

É a capacidade de diferenciar a invocação de métodos de acordo com a classe específica do objeto. Vamos supor este código:

```
class Veiculo():
    def ligar(self):
        raise NotImplementedError("A subclasse deve implementar este método")
```

```
class Carro(Veiculo):
    def ligar(self):
        return('ligando o carro')

class Moto(Veiculo):
    def ligar(self):
        return('ligando a moto')
```

Temos uma estrutura de classes aqui. Note que a classe base, Veiculo, levanta um erro, caso invoquemos o método ligar, pois ela não sabe que tipo de veículo ela é, portanto, não sabe como liga-lo. Ao executar, temos este comportamento:

```
>>> from poly import *
>>> v = Veiculo()
>>> c = Carro()
>>> m = Moto()
>>> v.ligar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File
"/home/cleuton/Documents/projetos/python/curso/licao6/poly.py"
, line 3, in ligar
    raise NotImplementedError("A subclasse deve implementar
este método")
NotImplementedError: A subclasse deve implementar este método
>>> m.ligar()
'ligando a moto'
>>> c.ligar()
'ligando o carro'
```

Se tentarmos invocar o método ligar a partir de um objeto Veiculo, o erro acontecerá. E cada objeto invoca o seu próprio método ligar, independentemente da classe base.

Classes abstratas e interfaces

Python não possui um mecanismo para isto, mas oferece um módulo, chamado **ABC**, que permite definir Abstract Base Classes ou classes básicas abstratas.

Uma classe abstrata ou mesmo uma interface, servem para criarmos um modelo para classes derivadas, que devem oferecer os métodos especificados. Por exemplo, uma classe derivada de veículo deve, obrigatoriamente, oferecer um método ligar. Em Java ou C# podemos fazer isto através de classes abstratas ou interfaces. Mas Python não tem esse recurso.

Podemos criar uma classe abstrata, com métodos abstratos, utilizando o módulo ABC:

```
from abc import ABC, abstractmethod

class Veiculo(ABC):
    @abstractmethod
    def ligar(self):
        pass

class Carro(Veiculo):
    pass
```

A classe Veiculo herda de ABC (abstract base class) e possui um método marcado com o decorator `@abstractmethod`, portanto, não pode ser instanciada. Ela serve de base para criarmos outras classes, como a classe Carro.

Só que a classe Carro não declarou o método abstrato ligar, portanto, também não pode ser instanciada:

```
>>> from veiculo3 import *
>>> c = Carro()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: Can't instantiate abstract class Carro with
abstract methods ligar

>>> v = Veiculo()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Veiculo with
abstract methods ligar
```

Dunders

Eu já falei sobre dunder (nomes que ficam entre dois caracteres sublinha, como `__init__`) e o Python tem vários métodos dunder para você declarar em sua classe.

Abra o programa exemplo do curso, `maze/model/labirinto.py`. Na classe `Labirinto` há dois métodos dunder:

```
def __init__(self, linhas=10, colunas=10):
    ...
def __str__(self):
    ...
```

O `__init__` já conhecemos: Ele faz o papel de Construtor de instâncias, e o `__str__` retorna uma representação informal em string do objeto, funcionando aproximadamente como o método `Object.toString()`, do Java.

O programa `maze/maze.py` simplesmente passa uma instância de `Labirinto` para o `print` e ela é impressa direitinho, porque o método `__str__` é invocado, de modo a retornar uma representação string para o `print` mostrar:

```
from model.labirinto import Labirinto
from solver import Solver

labirinto = Labirinto(5, 50)
print(labirinto)
```

```
solver = Solver()
solver.solve(labirinto)
print(labirinto)
```

Existem vários métodos dunder (ou mágicos) que você pode declarar em sua classe:

`__eq__`, `__lt__`, `__gt__` : Comparam instâncias da classe. O primeiro verifica igualdade e o segundo, se a instância é menor que outra. Por exemplo:

```
class Carro:
    def __init__(self, marca=None, comprimento=0.0):
        self.marca = marca
        self.comprimento = comprimento
    def __eq__(self, other):
        if isinstance(other, Carro):
            return self.marca==other.marca
        raise TypeError("Não é um Carro válido")

    def __lt__(self, other):
        if isinstance(other, Carro):
            return self.comprimento<other.comprimento
        raise TypeError("Não é um Carro válido")
```

Exemplo:

```
>>> from dunder import Carro
>>> a = Carro("Volkswagen", 3.50)
>>> b = Carro("Ford", 4.10)
>>> a==b
False
>>> c = a
>>> a==c
True
>>> a<b
True
```

```
>>> b<a  
False
```

Aqui você vai achar vários tipos de dunder methods:

<https://dbader.org/blog/python-dunder-methods>