Introducao_ao_Python

January 24, 2022

1 Introdução ao Python

Nesta aula veremos alguns **comandos básicos de Python 3** que serão utilizados ao longo do nosso curso.

1.1 IDEs

Para programar em Python, você pode usar um IDE (Integrated Development Environment ou Ambiente Integrado de Desenvolvimento). Trata-se de um software que vai te ajudar a programar, com dicas de nomes de funções, indicador de erros (*debugger*), dentre outras ferramentas. Alguns exemplos de IDEs para programar em Python: * Spyder * Eclipse * PyCharm * IDLE * Sublime

Você pode também usar um simples editor de texto e rodar o seu código pela janela de comando.

Estou usando o **Jupyter Notebook** porque ele permite colocar texto, equações e códigos computacionais no mesmo ambiente, sendo interessante na hora de ensinar algum conteúdo relacionado a programação.

Importante: utilize o **Python 3**, pois existem algumas diferenças com relação ao **Python 2**. Para saber a versão do Python que você está utilizando, execute os seguintes comandos.

[1]:
```
import sys

print(sys.version)
```

```
3.8.5 (default, Sep  4 2020, 07:30:14)
[GCC 7.3.0]
```

Note que os códigos em Python aqui no Jupyter vão ficar nessa caixa cinza, como no caso acima, e os resultados ficam logo abaixo. A partir deste momento, sempre que eu me referir a **Python**, na verdade eu estou me referindo a **Python 3**.

Vamos começar com a função *print()* :

[2]:
```
print("Hello World")
# Isso é um comentário
print('Hello World')
```

```
Hello World
Hello World
```

Essa função escreve na tela o que está entre parênteses. É importante para que você acompanhe os resultados do seu código. Note também que você pode **fazer comentários** no seu código, para explicar os comandos. Para fazer um comentário basta colocar o símbolo do jogo da velha # e depois o seu comentário. O Python não executa comentários.

1.2 Tipos de Variáveis

Temos alguns tipos de variáveis (*data types*) em Python, utilizadas para armazenar diferentes informações. Exemplos: *integer, float, complex, string, list, tuple, dictionary, boolean.*

Começando pelo ***integer***:

[3]:
```
a = 10
print(id(a))
b = 11
print(id(b))
```

```
94239680450144
94239680450176
```

No código acima, o computador reservou uma dada quantidade de memória (32 *bits*, ou 4 *bytes*) e chamou essa parte da memória de *a*. A essa parte da memória chamada de *a* foi associado o valor *10*. Note que esse código não vai ter nenhum resultado visível, porque não foi utilizada a função *print*.

Note agora:

[4]:
```
a = 10
b = 20
print(id(a))
print(id(b))
print(a,b)
a = b
print(a,b)
```

```
94239680450144
94239680450464
10 20
20 20
```

No código acima, *b* é uma outra variável do tipo integer. Para saber o tipo da variável, basta utilizar a função *type*:

[5]:
```
print(type(a))
print(type(b))
```

```
<class 'int'>
<class 'int'>
```

O tipo da variável *a* é *int*.

Obsevação: note que os códigos nas caixas cinzas são sequenciais, e não códigos independentes.

Podemos realizar operações com essas variáveis:

[6]:
```
print(4+3, 2*2, 9-6, 4**3, 3/2)
```

```
7 4 3 64 1.5
```

[7]:
```
x = 3
y = 2
print(x*y)
```

```
6
```

[8]:
```
z = x + y
print(z)
```

```
5
```

[9]:
```
Z = x - y
print(Z,z)
```

```
1 5
```

[10]:
```
z_dois = x**y
print(z_dois)
```

```
9
```

[11]:
```
x = 1
x = 4
x = x + 3
x += 3
print(x)
```

```
10
```

No código acima, primeiro foi dado o valor *1* ao espaço da memória chamado de *x*. Depois, esse mesmo espaço recebeu o valor *4*, ou seja, o valor *1* foi apagado e no seu lugar foi colocado o valor *4*. Na linha debaixo, o novo valor que será alocado no espaço da memória chamado de *x* será o valor que está atualmente em *x* somado a *3*.

O igual aqui em programação não tem o mesmo significado do igual na matemática. Aqui ele funciona como atribuição (assignment).

Depois do ***integer***, nós temos as variáveis ***float*** (números de ponto flutuante), que servem para aproximar/representar números reais como uma dada precisão.

Temos então:

[12]:
```
a = 1.0
print(a)
```

```
1.0
```

[13]:
```
print(type(a))
```

```
<class 'float'>
```

A variável *a* agora é um *float*. Temos as mesmas operações aritméticas básicas com *float*:

[14]:
```
a = 3.0
b = 5.0
print(a+b, a*b, a**b, a/b)
```

```
8.0 15.0 243.0 0.6
```

Note que a última operação acima é a divisão. O **Python 3** transforma automaticamente o resultado de uma divisão em um *float*, mesmo que as variáveis de entrada sejam *int*:

[15]:
```
a = 3
b = 2
c = a/b
print(c)
print(type(a))
print(type(b))
print(type(c))
```

```
1.5
<class 'int'>
<class 'int'>
<class 'float'>
```

Esse último resultado é muito importante, pois algumas linguagens não fazem isso. No **Python 2**, por exemplo, o resultado de *3/2* é *1*. O jeito mais seguro de realizar essas operações (se você quer o resultado correto) é garantir que as variáveis sejam *float*:

[16]:
```
a = 3.0
b = 2.0
c = a/b
print(c)
print(type(a))
print(type(b))
print(type(c))
```

```
1.5
<class 'float'>
<class 'float'>
<class 'float'>
```

Para transformar um *integer* em um *float* use a função *float()*, e o caminho inverso é dado pela função *int*:

[17]:
```
a = 4
b = float(a)
print(b)
print(type(a))
print(type(b))
```

```
4.0
<class 'int'>
<class 'float'>
```

[18]:
```
a = 1.9
b = int(a)
print(b)
```

```
1
```

O Python, assim como diversas outras linguagens de programação, também permite que você armazene e trabalhe com palavras, ou seja, com sequências de caracteres. Essas variáveis são chamadas de ***strings***.

Um *string* é definido entre aspas (simples ou duplas):

[19]:
```
nome = 'Arthur'
print(nome)
print(type(nome))
```

```
Arthur
<class 'str'>
```

[20]:
```
sobrenome = "Dent"
print(sobrenome)
```

```
Dent
```

[21]:
```
nome_completo = nome + sobrenome
print(nome_completo)
```

```
ArthurDent
```

[22]:
```
nome_completo_2 = nome + ' ' + sobrenome
print(nome_completo_2)
```

```
Arthur Dent
```

[23]:
```
print(2*nome)
```

```
ArthurArthur
```

[24]:
```
NOME = nome.upper()
print(NOME)
```

```
ARTHUR
```

[25]:
```
print(NOME.lower())
```

```
arthur
```

Existem diversas outras operações com *strings*, como, por exemplo, separar letras, transformar as letras em minúsculas, procurar uma sequência de caracteres dentro de uma *string*, etc. Essas operações não serão muito úteis aqui em nosso curso, mas podem ser interessantes em outras aplicações.

Podemos ter também várias informações armazenadas em uma única variável. Temos as *lists*, as *tuples* e os *dicts*.

Começando pelas **lists**, elas são definidas entre colchetes, com os componentes separados por vírgulas:

[26]:
```
x = [17, 11, -32]
print(x)
print(type(x))
print(type(x[0]))
```

```
[17, 11, -32]
<class 'list'>
<class 'int'>
```

Podemos acessar os componentes da lista da seguinte maneira:

[27]:
```
print(x[0])
print(x[1])
print(x[2])
```

```
17
11
-32
```

Importante: o índice inicial de uma lista é sempre o zero! Você também pode acessar uma lista usando índices negativos:

[28]:
```
print(x[-1])
print(x[-2])
print(x[-3])
```

```
-32
11
17
```

[29]:
```
x[0] = 39
print(x)
a = x[0]
b = x[1]
c = a + b
print(c)
```

```
[39, 11, -32]
50
```

[30]:
```
print(type(x[1]))
```

```
<class 'int'>
```

Podemos acrescentar termos na lista:

[31]:
```
x.append(193)
print(x)
```

```
[39, 11, -32, 193]
```

Podemos ter diferentes tipos de variáveis dentro de uma lista:

[32]:
```
z = [2020, 2.0, 'UnB', 'MNT']
print(z)
```

```
[2020, 2.0, 'UnB', 'MNT']
```

Para remover um elemento temos as seguintes opções:

[33]:
```
z.remove('UnB')
print(z)
```

```
[2020, 2.0, 'MNT']
```

[34]:
```
del z[1]
print(z)
```

```
[2020, 'MNT']
```

Colocando os termos em ordem crescente:

[35]:
```
print(sorted(x))
```

```
[-32, 11, 39, 193]
```

Dá pra fazer uma lista de listas:

[36]:
```
a = [[2,3,9],[1,1,-4],[0,2,5],[1,2,3,4,5,6]]
print(a)
print(a[1][2])
```

```
[[2, 3, 9], [1, 1, -4], [0, 2, 5], [1, 2, 3, 4, 5, 6]]
-4
```

As ***tuples*** são muito parecidas com as listas. A maior diferença é que as *tuples* não podem ser modificadas. É interessante utilizar uma *tuple* no código quando uma variável não puder ser modificada. Elas são definidas entre parênteses:

[37]:
```
a = (1.0 , 4, '9', '--')
print(a)
print(type(a))
```

```
(1.0, 4, '9', '--')
<class 'tuple'>
```

Para acessar os elementos é usada a posição do elemento entre colchetes também:

[38]:
```
print(a[2])
print(a[3])
```

```
9
--
```

Mas ao tentar modificar uma *tuple* ocorre um erro no código:

[39]:
```
a[1] = 10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-51d2cf888a38> in <module>
----> 1 a[1] = 10

TypeError: 'tuple' object does not support item assignment
```

Por fim, temos os ***dicts***, ou dicionários, que armazenam valores com suas respectivas chaves (*keys*). Um *dict* é definido entre chaves:

[47]:
```
aluno_matricula = {'Joseph': 339, 'Climber': 990}
print(aluno_matricula)
print(type(aluno_matricula))
```

```
{'Joseph': 339, 'Climber': 990}
<class 'dict'>
```

Para acessar um valor você usa a chave:

[48]:
```
print(aluno_matricula['Joseph'])
```

```
339
```

Podemos adicionar um novo valor em um *dict*:

[49]:
```
aluno_matricula['John'] = 2229
print(aluno_matricula)
```

```
{'Joseph': 339, 'Climber': 990, 'John': 2229}
```

Note que a ordem em um *dict* não é tão importante, diferente do que ocorre em *lists* e *tuples*.

1.3 Erros Comuns de Programação

Vou apresentar aqui alguns erros ocorrem muitas vezes quando estamos programando.

Erro 1: tentar somar uma *string* com um *int* ou um *float*:

[50]:
```
a = '10'
b = 8
c = a + b
print(c)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-91a195ddfdd8> in <module>
      1 a = '10'
      2 b = 8
----> 3 c = a + b
      4 print(c)

TypeError: can only concatenate str (not "int") to str
```

Como resolver esse problema?

[51]:
```
a = '10'
print(a)
a_float = float(a)
print(a_float)
b = 8
c = a_float + b
print(c)
```

```
10
10.0
18.0
```

Por outro lado, podemos somar *int* e *float*. O Python transforma o resultado automaticamente em *float*:

[52]:
```
a = 1
b = 2.0
c = a + b
print(c)
```

```
print(type(c))
```

```
3.0
<class 'float'>
```

Erro 2: tentar acessar uma posição de uma *list* (ou *tuple*) que não existe:

[53]:
```
a = [13,12,19,28]
```

[54]:
```
print(a[3])
print(a[4])
```

```
28
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-54-33e5cbf87cee> in <module>
      1 print(a[3])
----> 2 print(a[4])

IndexError: list index out of range
```

Erro 3: dividir por zero:

[55]:
```
a = 0
b = 5.0
c = b/a
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-55-aacead146b67> in <module>
      1 a = 0
      2 b = 5.0
----> 3 c = b/a

ZeroDivisionError: float division by zero
```

1.4 Numpy e Bibliotecas Externas

O Python possui diversas funções intrínsecas, que são do próprio Python. Para saber que funções são essas, podemos usar o comando:

[56]:
```
dir(__builtins__)
```

[56]:
```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
```

```
'BaseException',
'BlockingIOError',
'BrokenPipeError',
'BufferError',
'BytesWarning',
'ChildProcessError',
'ConnectionAbortedError',
'ConnectionError',
'ConnectionRefusedError',
'ConnectionResetError',
'DeprecationWarning',
'EOFError',
'Ellipsis',
'EnvironmentError',
'Exception',
'False',
'FileExistsError',
'FileNotFoundError',
'FloatingPointError',
'FutureWarning',
'GeneratorExit',
'IOError',
'ImportError',
'ImportWarning',
'IndentationError',
'IndexError',
'InterruptedError',
'IsADirectoryError',
'KeyError',
'KeyboardInterrupt',
'LookupError',
'MemoryError',
'ModuleNotFoundError',
'NameError',
'None',
'NotADirectoryError',
'NotImplemented',
'NotImplementedError',
'OSError',
'OverflowError',
'PendingDeprecationWarning',
'PermissionError',
'ProcessLookupError',
'RecursionError',
'ReferenceError',
'ResourceWarning',
'RuntimeError',
```

```
'RuntimeWarning',
'StopAsyncIteration',
'StopIteration',
'SyntaxError',
'SyntaxWarning',
'SystemError',
'SystemExit',
'TabError',
'TimeoutError',
'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'ZeroDivisionError',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'breakpoint',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
'credits',
'delattr',
```

```
'dict',
'dir',
'display',
'divmod',
'enumerate',
'eval',
'exec',
'filter',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
'hasattr',
'hash',
'help',
'hex',
'id',
'input',
'int',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
'locals',
'map',
'max',
'memoryview',
'min',
'next',
'object',
'oct',
'open',
'ord',
'pow',
'print',
'property',
'range',
'repr',
'reversed',
'round',
'set',
'setattr',
'slice',
```

```
 'sorted',
 'staticmethod',
 'str',
 'sum',
 'super',
 'tuple',
 'type',
 'vars',
 'zip']
```

Vamos usar mais essas funções (o nome correto é método) que aparecem em minúsculo. Os métodos que aparecem entre dois *underscores* são chamados de **métodos mágicos**, eles são chamados pelo interpretador. Para saber um pouco mais sobre uma dada função, use o comando *help()*:

[57]:
```
#help(__builtins__.abs)
```

Podemos ainda importar várias bibliotecas externas que possuem diversas funções extremamente úteis, de acordo com a aplicação. Vamos utilizar aqui, nessa aula, duas biliotecas externas: a **numpy** para operações matriciais e vetoriais e a **matplotlib** para plotar gráficos.

Para importar uma biblioteca, você deve usar o comando ***import*** no início do código:

[58]:
```
import numpy as np

x = np.cos(1.0)
print(x)
```

```
0.5403023058681398
```

Note que eu importei a bilioteca *numpy* e dei a ela um apelido, *np*. Isso porque vamos precisar digitar o nome da biblioteca repetidamente, então o apelido facilita um pouco a nossa vida. Alguns exemplos com *numpy*:

[59]:
```
a = np.pi
print(a)
```

```
3.141592653589793
```

[60]:
```
b = np.sin(a)
print(b)
```

```
1.2246467991473532e-16
```

(O resultado acima deveria ser zero, mas não é. Por quê?)

[61]:
```
c = np.cos(a)
print(c)
```

```
-1.0
```

Para conhecer todas as funções que existem no *numpy*, basta usar o comando *dir*:

[62]:
```
#dir(np)
```

Uma função muito utilizada dentro do *numpy* é a *linalg*, que possui funções que invertem matrizes, resolvem sistemas lineares, encontram autovalores e autovetores, etc. Para saber um pouco mais:

[63]:
```
#help(np.linalg)
```

Mais alguns exemplos com *numpy*:

[64]:
```
a = np.array([1,7,9])
print(a)
print(type(a))
```

```
[1 7 9]
<class 'numpy.ndarray'>
```

[65]:
```
print(a[1])
```

```
7
```

[66]:
```
matriz_de_zeros = np.zeros((4,4))
print(matriz_de_zeros)
matriz_de_uns = np.ones((7,9))
print(matriz_de_uns)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]]
```

[67]:
```
x = np.arange(0,0.1,0.01)
print(x)
```

```
[0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09]
```

[68]:
```
x = np.arange(0,10,1)
print(x)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

[69]:
```
x = np.linspace(0.0, 10.0, 101)
print(x)
```

```
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.   1.1  1.2  1.3
  1.4  1.5  1.6  1.7  1.8  1.9  2.   2.1  2.2  2.3  2.4  2.5  2.6  2.7
  2.8  2.9  3.   3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9  4.   4.1
  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.   5.1  5.2  5.3  5.4  5.5
  5.6  5.7  5.8  5.9  6.   6.1  6.2  6.3  6.4  6.5  6.6  6.7  6.8  6.9
  7.   7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9  8.   8.1  8.2  8.3
  8.4  8.5  8.6  8.7  8.8  8.9  9.   9.1  9.2  9.3  9.4  9.5  9.6  9.7
  9.8  9.9 10. ]
```

[70]:
```
A = np.array([[1,9,8],[9,2,2],[5,3,11]])
print(A)
```

```
[[ 1  9  8]
 [ 9  2  2]
 [ 5  3 11]]
```

[71]:
```
print(A[1,2])
print(A[2,2])
print(A[0,0])
```

```
2
11
1
```

[72]:
```
B = np.copy(A)

print(B)

B[0,0] = 19.0

print(B)

print(A)

print(id(A))
print(id(B))

#B = np.copy(A)
#print(B)
```

```
[[ 1  9  8]
 [ 9  2  2]
 [ 5  3 11]]
[[19  9  8]
 [ 9  2  2]
```

```
 [ 5  3 11]]
[[ 1  9  8]
 [ 9  2  2]
 [ 5  3 11]]
139907828529792
139907470682352
```

[73]:
```
nova_matriz_de_zeros = np.zeros((5,5),float)
print(nova_matriz_de_zeros)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

1.5 Laços de Repetição (Loops)

Frequentemente precisamos que um mesmo procedimento seja repetido diversas vezes. Para isso usamos os ***loops*** ou ***laços de repetição***. Essa será uma das principais ferramentas aqui no nosso curso, por isso é fundamental entendê-la bem.

Temos duas maneiras de realizar um loop em Python: com o **for** ou com o **while**.

Vamos começar com o ***for***:

[74]:
```
for i in range(2,7):
    print(i)
```

```
2
3
4
5
6
```

Note o espaço antes de *print*, isso indica que essa função está dentro do *loop* do *for*. Veja esse outro exemplo:

[75]:
```
for i in range(2,9):
    print('oi', i)
    a = 2*i
print('adeus')
```

```
oi 2
oi 3
oi 4
oi 5
oi 6
oi 7
```

```
oi 8
adeus
```

O *oi* está dentro do *loop*, mas o *adeus* está fora. Por isso *oi* aparece várias vezes mas *adeus* aparece uma única vez. Neste caso, *i* é a variável de repetição e *range(2,9)* significa que o *i* vai variar de *2* a *8*: sempre começa no número da esquerda e vai até o número da direita menos *1*.

(**Dica:** sempre que você estiver utilizando uma nova função ou ferramenta, faça muitos testes antes de implementá-la em seu código. Você precisa saber exatamente como ela funciona, principalmente em situações extremas.)

Veja os exemplos abaixo:

[76]:
```
for i in range(5):    # Função range com um único argumento
    print(i)
```

```
0
1
2
3
4
```

[77]:
```
for i in range(-10,2):    # Função range com dois argumentos
    print(i)
```

```
-10
-9
-8
-7
-6
-5
-4
-3
-2
-1
0
1
```

[78]:
```
for i in range(-10, 20, 3):    # Função range com três argumentos
    print(i)
```

```
-10
-7
-4
-1
2
5
8
11
```

```
14
17
```

No funcionamento da função *range* no exemplo acima: o *i* vai de *-10* a *19*, de *3* em *3*.

Nesses exemplos acima, executamos apenas um *print* dentro do *loop*, ou seja, repetidas vezes. Mas podemos colocar qualquer tipo de operação dentro de um *loop*. Veja mais alguns exemplos:

[79]:
```
for i in range(4):
    a = 2*i
    print(a)
```

```
0
2
4
6
```

[80]:
```
a = 0
for i in range(10):
    a = a + i
    print('Valor de a : ', a)
```

```
Valor de a :  0
Valor de a :  1
Valor de a :  3
Valor de a :  6
Valor de a :  10
Valor de a :  15
Valor de a :  21
Valor de a :  28
Valor de a :  36
Valor de a :  45
```

[81]:
```
a = 0
for i in range(10):
    a = a + i
print('Valor de a : ', a)
```

```
Valor de a :  45
```

Por que usar loops? Os dois códigos abaixo levam ao mesmo resultado.

[82]:
```
a = 0
a = a + 1*1
a = a + 2*2
a = a + 3*3
a = a + 4*4
a = a + 5*5
a = a + 6*6
print(a)
```

```
91
```

[83]:
```
a = 0
for i in range(7):
    a = a + i*i
print(a)
```

```
91
```

Note agora o processo de iteração em uma matriz:

[84]:
```
x = np.linspace(0.0,1.0,11)
print(x)
for i in range(len(x)):
    print(x[i])
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
0.0
0.1
0.2
0.30000000000000004
0.4
0.5
0.6000000000000001
0.7000000000000001
0.8
0.9
1.0
```

Agora em uma matriz com linhas e colunas:

[85]:
```
A = np.array([[90,91,92],[93,94,95],[96,97,98]])
print(A)
```

```
[[90 91 92]
 [93 94 95]
 [96 97 98]]
```

[86]:
```
for i in range(3):
    for j in range(3):
        print(A[i,j])
```

```
90
91
92
93
94
95
96
```

```
97
98
```

No código acima temos um *loop* dentro de um *loop*. No exemplo abaixo também temos essa estrutura (veja a ordem em que o *loop* ocorre):

[87]:
```
for i in range(5):
    for j in range(3):
        print(i,j)
```

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
3 0
3 1
3 2
4 0
4 1
4 2
```

A outra maneira de fazer um *loop* é utilizando o comando ***while***. Esse comando vai repetir as operações dentro do bloco enquanto a condição inicial for verdadeira. Alguns exemplos:

[88]:
```
a = 0
print('Valor de a: ', a)
while a < 5:
    a = a + 1
    print('Valor de a: ', a)
```

```
Valor de a:  0
Valor de a:  1
Valor de a:  2
Valor de a:  3
Valor de a:  4
Valor de a:  5
```

[89]:
```
a = 0
print('Valor de a: ', a)
while a <= 5:
    a = a + 1
    print('Valor de a: ', a)
```

```
Valor de a:  0
```

```
Valor de a:  1
Valor de a:  2
Valor de a:  3
Valor de a:  4
Valor de a:  5
Valor de a:  6
```

[90]:
```
a = 0
print('Valor de a: ', a)
while a == 5:
    a = a + 1
    print('Valor de a: ', a)
```

```
Valor de a:  0
```

O *loop* continua enquanto a condição inicial for verdadeira (*True*). Quando a condição se torna falsa, o *loop* para. Por exemplo:

[91]:
```
print(10 > 5)
```

```
True
```

[92]:
```
print(10 < 5)
```

```
False
```

[93]:
```
print(5 == 5)
```

```
True
```

[94]:
```
print(10 == 5)
```

```
False
```

Devemos tomar cuidado com os critérios de parada no *loop while*, pois podemos entrar em um *loop* infinito.

1.6 Estruturas Condicionais

Condicionais serão extremamente importantes em nosso curso também. Juntamente com os comandos de *loop*, eles fazem parte das ferramentas de controle de fluxo em um programa. Os comandos relacionados aos condicionais são ***if***, ***elif*** e ***else***. O que tem dentro de um bloco ***if*** só será executado pelo programa se a condição inicial for obedecida (verdadeira, ou *True*). Vamos ver alguns exemplos:

[95]:
```
x = 1.0
if x > 0.0:
    print('x é maior que zero')
```

```
x é maior que zero
```

[96]:
```
x = -1.0
if x > 0.0:
    print('x é maior que zero')
```

No código acima não aparece nada escrito na tela, porque a função *print* está dentro de um bloco *if* cuja condição não é verdadeira. Veja agora os dois códigos abaixo:

[97]:
```
x = 1.0
if x > 0.0:
    print('x é maior que zero')
print('x é maior que zero')
```

```
x é maior que zero
x é maior que zero
```

[98]:
```
x = -1.0
if x > 0.0:
    print('x é maior que zero')
print('x é maior que zero..')
```

```
x é maior que zero..
```

Os comando *elif* e *else* são utilizados da seguinte maneira:

[99]:
```
x = 3.5
if x > 5.0:
    print('x é maior que 5')
elif x <= 5.0 and x >= 3.0:
    print('x está entre 3 e 5, inclusive')
else:
    print('x é menor que 3')
```

```
x está entre 3 e 5, inclusive
```

O *elif* apresenta uma condição auxiliar. Se a condição do *if*, que é a principal, não for obedecida, então o programa vai para a próxima condição *elif*. Podemos ter quantas condições *elif* quisermos. A condição *else* encerra o condicional: se nenhuma das condições acima do *else* for obedecida, então o programa vai executar o que está no bloco do *else*.

[100]:
```
x = 7.0
if x < 5 or x > 8:
    print('x é menor que 5 ou maior que 8')
```

[101]:
```
x = 5.0
if x > 3 and x > 7:
    print('x é maior que 7')
```

[102]:
```
x = 3.0
if x == 3:
```

```
    print('x é igual a 3')
```

```
x é igual a 3
```

[103]:
```
x = 4.0
if x != 3:
    print('x é diferente de 3')
```

```
x é diferente de 3
```

Podemos combinar *loops while* com condicionais *if* da seguinte maneira:

[104]:
```
a = 0
while True:
    print(a)
    a = a + 1
    if a == 5: break
```

```
0
1
2
3
4
```

[105]:
```
for i in range(10):
    if i == 2: break
    print(i)
```

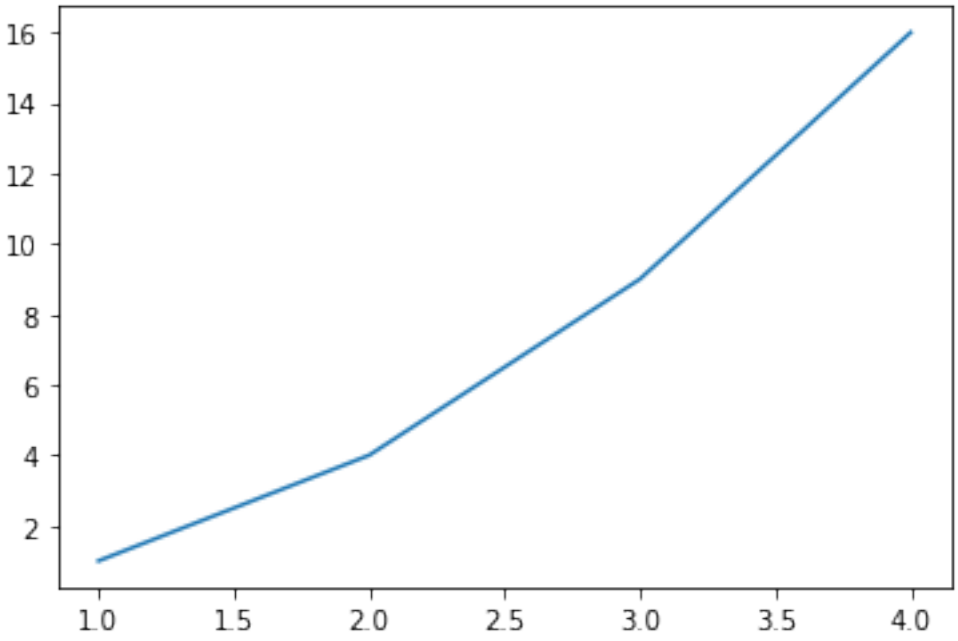
```
0
1
```

1.7 Gráficos

Para fazer gráficos vamos utilizar a biblioteca matplotlib.

[106]:
```
import matplotlib.pyplot as plt
```

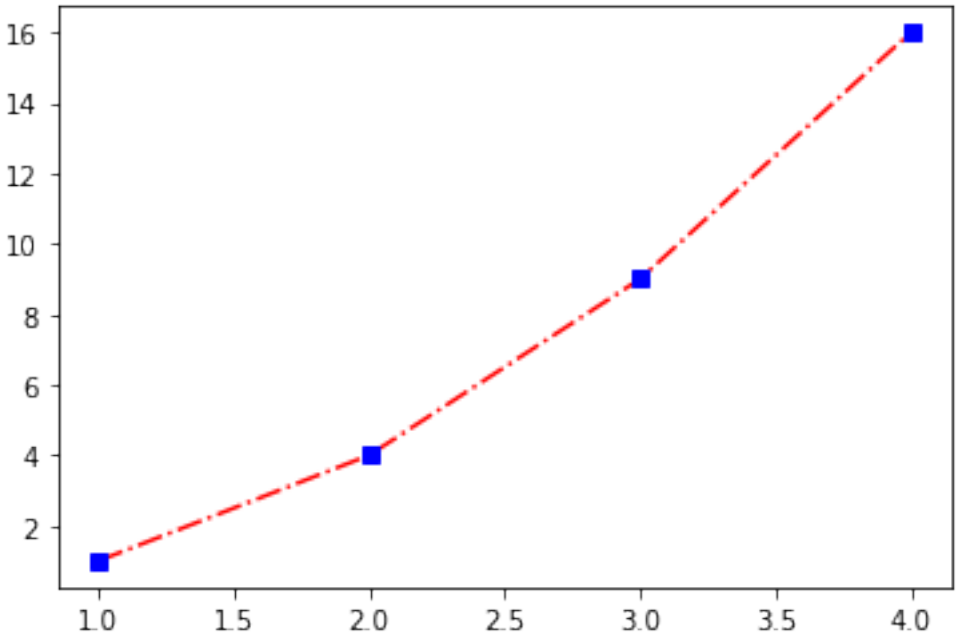
[107]:
```
x = np.array([1,2,3,4])
y = np.array([1,4,9,16])
plt.plot(x,y)
plt.show()
```



Podemos também exportar o gráfico, para que você possa utilizá-lo no seu trabalho.

[108]:
```
x = np.array([1,2,3,4])
y = np.array([1,4,9,16])
plt.plot(x,y,'r-.')
plt.plot(x,y,'bs')
#plt.savefig('figura.png', format='png', dpi=1200, bbox_inches='tight')
#plt.savefig('figura.pdf', format='pdf', dpi=1200, bbox_inches='tight')
```

[108]:
```
[<matplotlib.lines.Line2D at 0x7f3eacf06a90>]
```



No exemplo acima, note as opções para plotar o resultado. *k-* indica uma linha preta, enquanto *bo* indica círculos azuis. Para mais opções: Google. Vamos plotar gráficos usando o *numpy*:

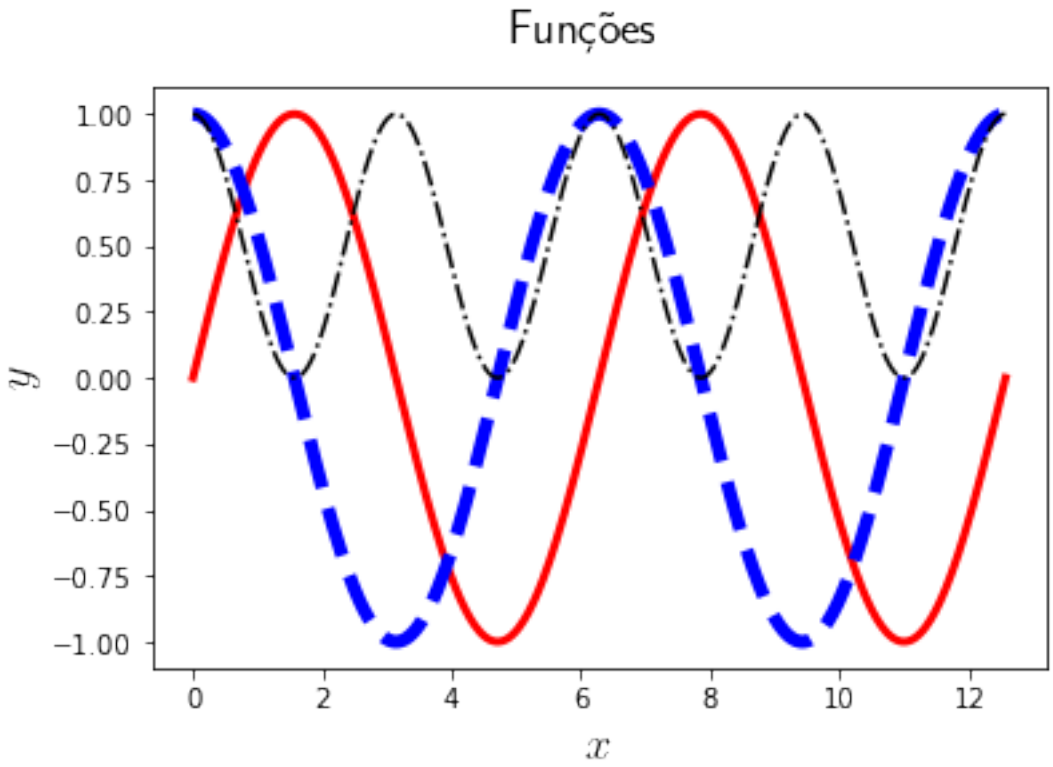
[109]:
```
x = np.linspace(0.0, 4.0*np.pi, 500)
y = np.sin(x)
z = np.cos(x)
z2 = np.cos(x)**2.0
z3 = np.cos(x**2.0)

fig = plt.figure ()
ax = fig.add_subplot()

fig.suptitle('Funções',  fontsize=18, fontweight='bold', usetex=True)
ax.set_ylabel(r'$y$',fontsize=18, usetex=True)
ax.set_xlabel(r'$x$',fontsize=18, usetex=True)

plt.plot(x,y,'-r', lw = 3)
plt.plot(x,z, '--b', lw = 5)
plt.plot(x,z2, '-.k')
#plt.plot(x,z3)

plt.show()
```



Note as várias opções utilizadas para plotar o gráfico acima. Existem várias maneiras de personalizar o seu gráfico e deixá-lo mais profissional. Vamos ver algumas dessas opções ao longo do curso.

1.8 Criando Funções

Uma função é um bloco de código que só vai rodar quando for chamada no código principal ou em outra função. Você pode passar variáveis de entrada para a função (*input*) e pegar variáveis como resultado da função (*output*). A função deve ser sempre definida antes de ser chamada no código. No **Python** uma função é criada utilizando-se o comando ***def***. Exemplo:

[110]:
```
def hello():
    print('Hello World')

#dir()
```

[111]:
```
def triplo(x):
    y = 3.0*x
    print(f'O triplo de {x} é {y}')
    return y
```

Nada acontece ao executar apenas o código acima, pois a função não foi chamada. Abaixo vou chamar a função:

[112]:
```
print(triplo(4.0))
```

```
O triplo de 4.0 é 12.0
12.0
```

Mais alguns exemplos:

[113]:
```
def soma(a,b):
    return a + b
x = 30
y = 19
print(soma(x,y))
```

```
49
```

[114]:
```
def nome_completo(nome, sobrenome):
    print(nome + ' ' + sobrenome)

nome_completo('José', 'Arcadio')
```

```
José Arcadio
```

[115]:
```
def componentes(a):
    for i in a:
        print(i)

x = np.array([1,5,9,-3,-17])
componentes(x)
```

```
1
5
9
-3
-17
```

[116]:
```
a = 5
def soma2(a,dd):
    print('------', a)
    return a + dd
print(soma2(11,17))
print(a)
```

```
------ 11
28
5
```

Podemos ter funções recursivas (tente entender a função abaixo):

[117]:
```
def fatorial(n):
    if n < 0:
        print('O número deve ser positivo!')
    elif n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n*fatorial(n-1)

fatorial(6)
```

[117]: 720

Você não precisa obrigatoriamente utilizar funções em seu código, porém elas deixam o código muito mais organizado e eficiente.

1.9 Conclusão

O que nós vamos precisar aqui no curso, de verdade, é de

1. ***loops*** (*for* e/ou *while*),
2. ***matrizes*** do *numpy*,
3. ***plotar gráficos*** com o *matplotlib* e
4. ***funções***, para podermos utilizar o *numba*.

Com essas 4 ferramentas será possível fazer todos os trabalhos do curso.

Mas é claro que você pode ir muito além.