# Python Básico

André Nepomuceno

Universidade Federal Fluminense

2 de julho de 2021

# O que você vai aprender ?


## Minicursos de Python

Aulas online e gratuitas, com emissão de certificado

**Minicurso 1: Python Básico**

- Instalação
- Módulos
- Variáveis e Expressões
- Funções
- Operações com Strings
- Estruturas de Condição
- Iterações
- Numpy
- Matplotlib Básico

Carga horária: 20h, com 4h por semana, durante 5 semanas.
Início das aulas: 01/06/21

**Minicurso 2: Python Científico**

- Matplotlib Avançado
- Solução Numérica de Equações
- Integração Numérica
- Solução Numérica de Equações Diferenciais
- Ajuste de Curvas
- Transformada de Fourier
- Álgebra Linear com Python
- Computação Simbólica (SymPy)

Carga horária: 20h, com 4h por semana, durante 5 semanas.
Início das aulas: A definir

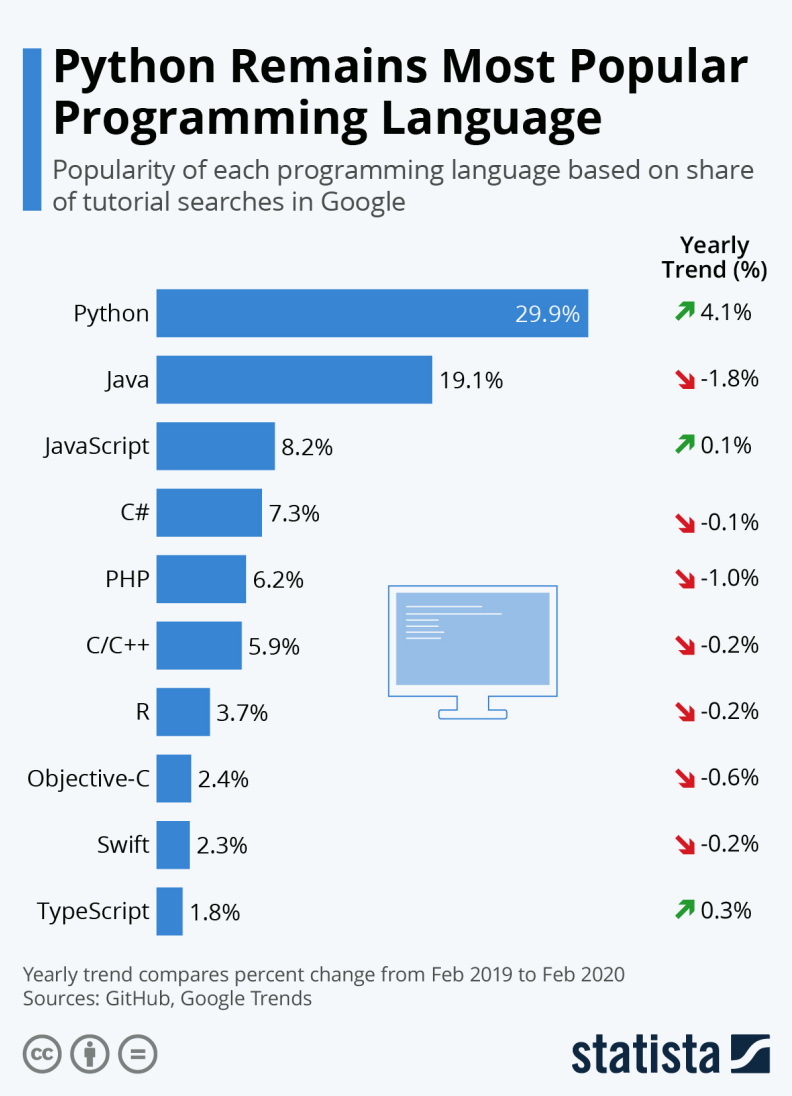
Inscrição para o Minicurso Python Básico:
http://www.professores.uff.br/andrenepomuceno/minicurso-python/
Contato: andrenepomuceno@id.uff.br

# Babel da Programação

Existem aproximadamente 600 linguagens de programação.

- FORTRAN (1957)
- COBOL (1959)
- BASIC (1964)
- Pascal (1970)
- C (1972)
- C++ (1980)
- MATLAB (1984)
- LabVIEW (1986)
- **Python** (1990)
- JavaScript (1995)

# Roubando a Cena




Python é usado em diferentes áreas

- Ciência de Dados
- Inteligência Artificial
- Desenvolvimento Web
- Desenvolvimento de jogos
- Medicina e Farmacologia (AstraZeneca)
- Bioinformática
- Neurociência
- Física e Astronomia
- Business

# Vantagens x Desvantagens

**Vantagens**

- Sintaxe clara e simples
- Gratuito e código aberto
- Disponível para diversas plataformas
- Relativamente fácil de aprender
- Diversos módulos e bibliotecas disponíveis

**Desvantagens**

- A execução do programa pode ser lento, quando comparado a outras linguagens
- Rápido desenvolvimento pode levar a incompatibilidade entre as versões (Python 2 x Python 3)

# Instalação I - Windows

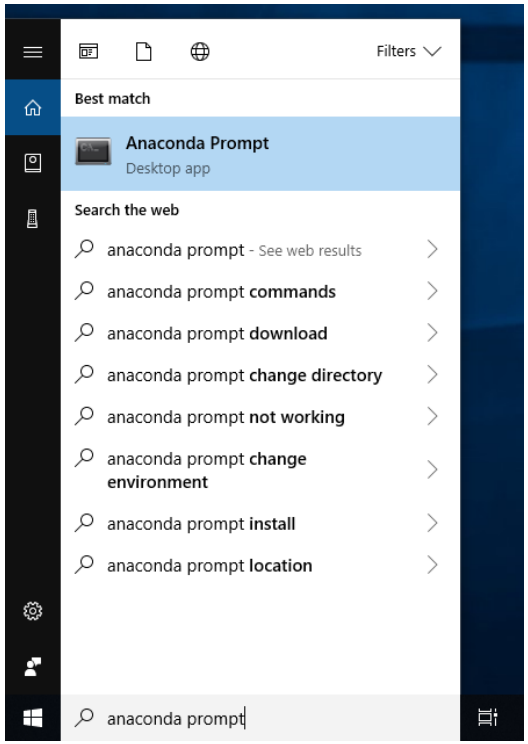
Baixe a versão mais recente do miniconda compatível com o seu sistema operacional (Windows, Linux ou MacOS).
Site: `https://docs.conda.io/en/latest/miniconda.html`

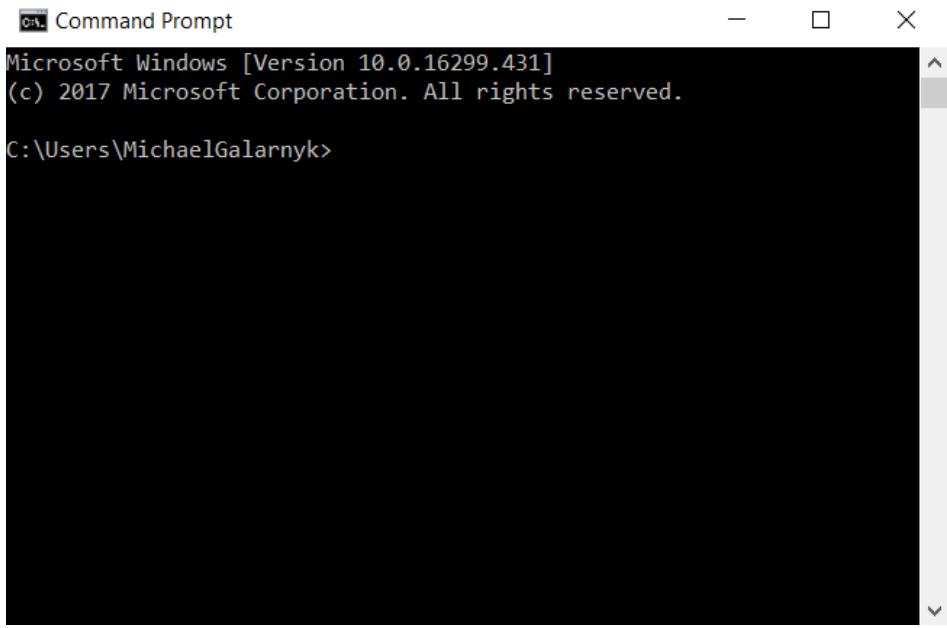
**Instalação no Windows**

- Baixe o instalador
- Execute o arquivo .exe
- Siga as instruções da tela
- Concluída a instalação, abra o "Anaconda Prompt" no menu **Iniciar**
- Para testar a instalação, escreva o comando `conda list` no Anaconda Prompt.

# Instalação I - Windows

## Anaconda Prompt



# Instalação II - Linux

## Instalação no Linux

- Baixe o instalador
- Abra um terminal (Ctrl+Alt+T)
- Execute o comando
  `bash Miniconda3-latest-Linux-x86_64.sh`
- Siga as instruções da tela
- Ao final, feche o terminal e abra um novo
- Teste a instalação: `conda list`

# Instalação III - Android

## QPython3L - Python for Android

- Esse é um dos aplicativos disponíveis no Google Play
- Pode ser usado para programas básicos, mas não para projetos complexos.

# Instalação IV - Módulos

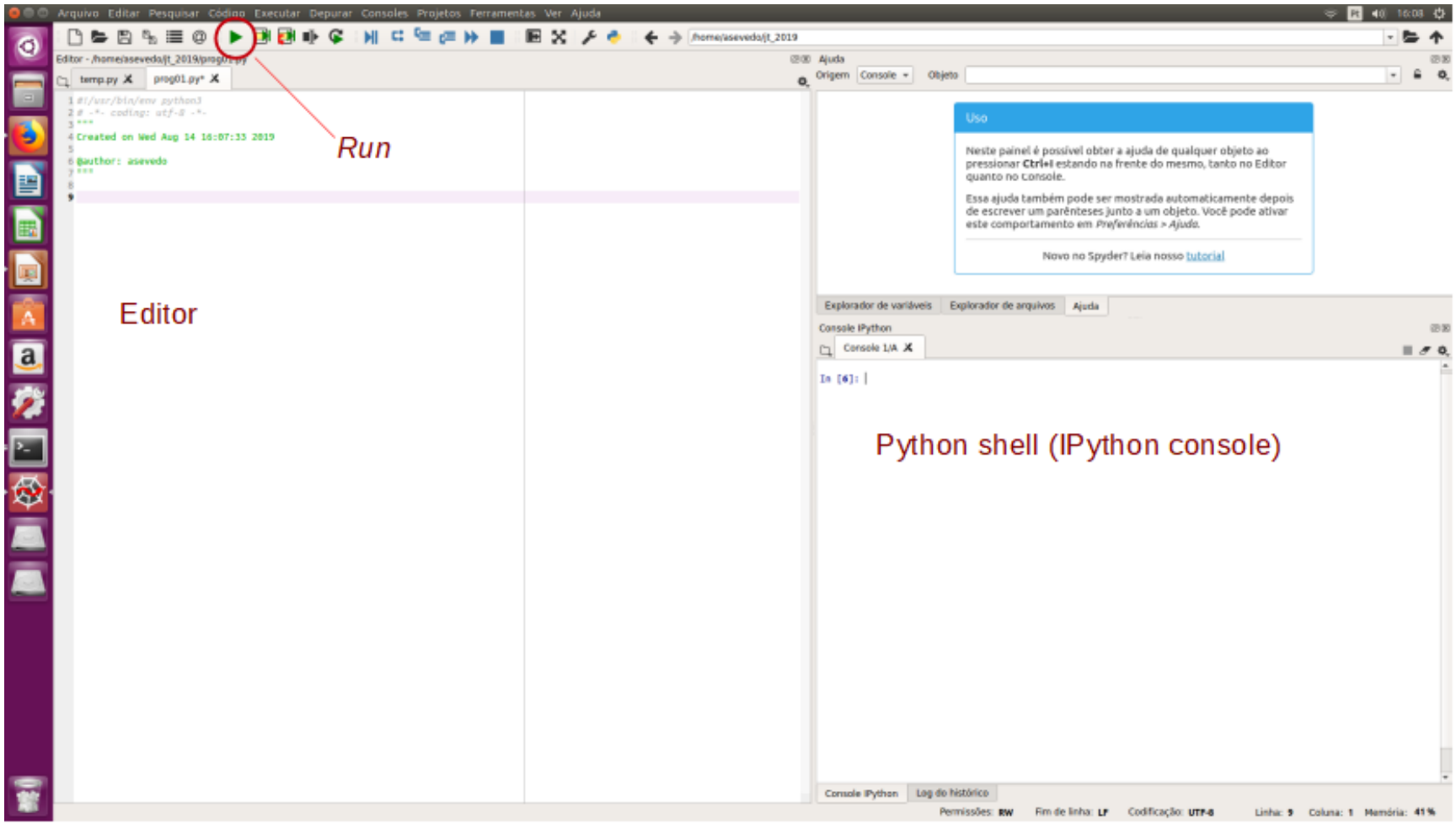
Vamos instalar os pacotes **spyder**, **jupyter**, **numpy**, **matplotlib** e **scipy**

```
conda install spyder
conda install jupyter
conda install numpy matplotlib scipy
```

# O Editor Spyder

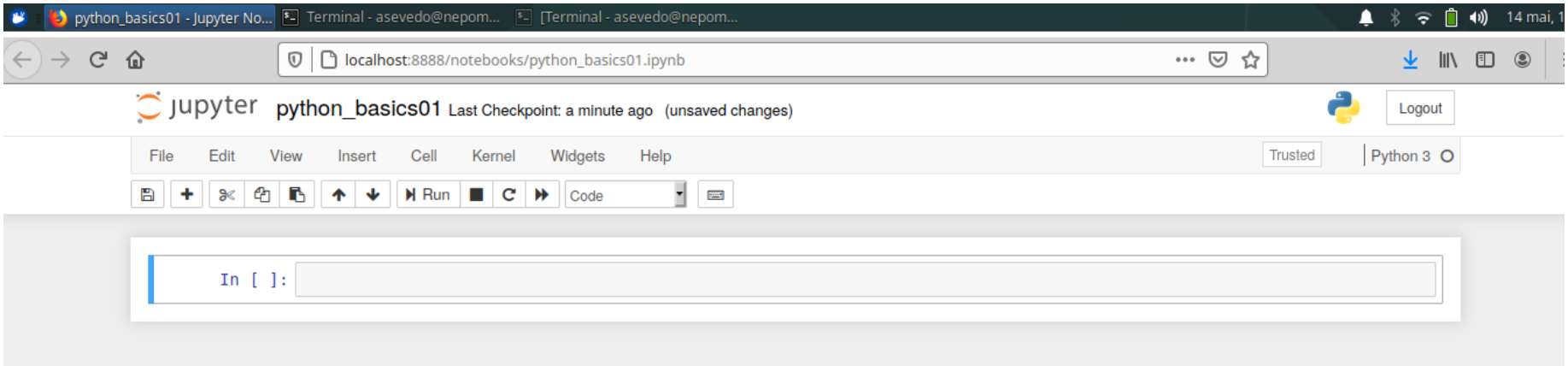
Para abrir o spyder: `spyder`

Os arquivos (scripts) devem ser salvos com extensão ".py"



# Jupyter Notebook

Jupyter Notebook é literalmente um "livro de anotações" do seu trabalho.
Basicamente, é um documento interativo aberto num browser.
Para abrir: `jupyter notebook`
Novo código: **New** → **Python 3**
Versão web: `https://jupyter.org/try`



# Criando um Programa em Python

- Vamos utilizar o editor spyder para criar nosso programa.
- No spyder, escolha um diretório onde os programas serão salvos.
- Vamos criar um programa que imprime a mensagem "Hello World"
- Para imprimir uma mensagem na tela, usamos a função **print**.
- Exemplo: print("Hello World!")
- Salvamos o programa como prog01.py, e clicamos na seta verde do spyder ("Run File") para executá-lo.

# Tipos de Dados

Basicamente, existem quatro tipos de dados: string, inteiro e flutuante e complexo.

**Strings (type:`str`)**
Strings são textos formados por caracteres. Em Python, strings são sempre escrito entre aspas.

**Inteiros (type:`int`)**
Podem ser positivos, negativos ou zero, por exemplo 1, 9, - 54, 38523. A aritmética de inteiros é exata.
Em Python, é possível separar dígitos com "_". Por exemplo, o número 10525839 pode ser escrito como 10_525_839.

# Tipos de Dados

**Flutuante (type:`float`)**
Flutuantes são representações de números reais. Por exemplo: 1.3, -0.456 ou $1.65 \times 10^{-6}$. **Note que usamos ponto (.) como separador decimal**.
Notação científica é escrita como `1.65e-6`.
Exemplo: 13 é inteiro, 13.0 é flutuante, e "13" é um string.

**Complexo (type:`complex`)**
Números complexos são formados por uma parte real e uma imaginária. Em Python, um número complexo é formado por dois flutuantes. Por exemplo, o número $5.1 + 2.3\mathbf{i}$ pode ser escrito de duas formas: `5.1+2.3j` ou `complex(5.1,2.3)`.

# Conversão

**Conversão (Type Casting)**

As vezes é necessário transformar um tipo de dado em outro tipo (transformar um string em um inteiro, por exemplo). As funções que fazem essa conversão são chamadas *type casting*, e são as seguintes:

**int()** - transforma o valor entre parênteses em um inteiro;
**float()** - transforma o valor entre parênteses em um flutuante;
**complex()** - transforma o valor entre parênteses em um complexo;
**str()** - transforma o valor entre parênteses em um string.

**Exercício**:Digite os comandos abaixo no jupyter e observe o resultado.

```
float(3)
int(7.2)
int(7.9)
complex(3.)
complex(0,3.)
```

# Operações

## Operações Aritméticas do Python

+ adição
- subtração
* multiplicação
/ divisão
// inteiro da divisão (retorna o **menor** inteiro)
% módulo (resto da divisão)
** potência (m elevado a n, m**n)

## Warning I

O operador de divisão (/) sempre retorna um **flutuante**, mesmo que a divisão seja entre inteiros divisíveis.

# Atributos e Métodos

Em Python, tudo é um **objeto**, incluindo números (um objeto é uma combinação de dados e funções). Objetos possuem *atributos* e *métodos*. O método é uma função que atua no objeto. A sintaxe para acessar atributos e métodos é:

- Atributo: <objeto>.<atributo>
- Métodos: <objeto>.<metodo>()

## Example

```
>>> (4 + 5j).real
4.0
>>> (4 + 5j).imag
5.0
>>> (4 + 5j).conjugate()
(4-5j)
```

# Funções Matemáticas

Existem duas funções matemáticas disponíveis "by default" no Python. A função **abs**, que retorna o módulo de um número, e a função **round**, que faz a aproximação de um número flutuante.

## Example

```
>>> abs(-6.4)
6.4
>>> abs(3 + 4j)
5.0
>>> round(5.56)                    #retorna um inteiro
6
>>> round(3.141592653589,2) #2 casas decimais
3.14
```

Como calcular raiz quadrada e funções trigonométricas ?

# Módulos

Em Python, um módulo é um conjunto de funções. Para utilizar uma função de um determinado módulo, você precisará **importar** o módulo para o seu programa, da seguinte forma:
**from** <nome_do_modulo> **import** <funcão1> <função2 >...
Por exemplo, a função **sqrt()**, que calcula a raiz quadrada de um número, pertente ao módulo *math*. Para usá-la num programa, devemos escrever:

```
from math import sqrt
sqrt(2)
```

Em geral, um módulo contém várias funções. E como podemos saber quais funções estão disponíveis num módulo ? Basta importamos o módulo e usar o comando **help**. Tente os seguintes comandos no jupyter:

```
import math
help(math)
```

# Módulo **math**

## Algumas funções do módulo `math`

| | |
|---|---|
| `math.sqrt(x)` | $\sqrt{x}$ |
| `math.exp(x)` | $e^x$ |
| `math.log(x)` | $\ln x$ |
| `math.log(x, b)` | $\log_b x$ |
| `math.log10(x)` | $\log_{10} x$ |
| `math.sin(x)` | $\sin(x)$ |
| `math.cos(x)` | $\cos(x)$ |
| `math.tan(x)` | $\tan(x)$ |
| `math.asin(x)` | $\arcsin(x)$ |
| `math.acos(x)` | $\arccos(x)$ |
| `math.atan(x)` | $\arctan(x)$ |
| `math.sinh(x)` | $\sinh(x)$ |
| `math.cosh(x)` | $\cosh(x)$ |
| `math.tanh(x)` | $\tanh(x)$ |
| `math.asinh(x)` | arsinh(x) |
| `math.acosh(x)` | arcosh(x) |
| `math.atanh(x)` | artanh(x) |
| `math.hypot(x, y)` | The Euclidean norm, $\sqrt{x^2 + y^2}$ |
| `math.factorial(x)` | $x!$ |

O módulo math tem um atributo que retorna o número $\pi$: `math.pi`.

# Módulos

Existe uma outra maneira, mais usada profissionalmente, para importar um módulo. Nessa forma, damos um apelido ao módulo, e a função só é chamada quando precisa ser executada. Veja o exemplo abaixo:

```
import math as mt
mt.sqrt(3.6)
```

**Exercício**

1. Utilize o módulo math para calcular a raiz quadrada, o logaritmo decimal e o logaritmo neperiano de um número.
2. Escreva um programa que utilize o módulo math para transformar um ângulo de 30 graus em radianos e que calcula o seno, cosseno e tangente deste ângulo (math.radians(x)).

# Variáveis

Em termos simples, uma variável é um letra ou palavra que guarda um valor de qualquer tipo de dado (string, inteiro, flutuante ou complexo). Mais precisamente, uma variável reserva um espaço na memória do computador para guardar o valor a ela atribuído.
Para criar uma variável, escolhemos um nome e atribuímos um valor a variável usando o operador (=). Execute o exemplo abaixo:

```
x = 5
print(x)
```

O que acontece aqui ?
1. Primeiro, criamos a variável x e atribuímos a ela o valor 5. Automaticamente, Python interpretará que a variável x é um inteiro.
2. O valor atribuído a x é mostrado pela função print. Note que ela mostra o **valor** de x, e não a letra (nome da variável).

# Variáveis

Quando um novo valor é atribuído a variável, o valor anterior é "esquecido", e a variável passa a ter o valor atual. Exemplo:

```
x  = 5
print(x)
x = 9*2 + 1  #substitui o valor anterior pelo novo
print(x)
x = "agora sou string"
print(x)
```

É possível criar várias variáveis numa única linha:

```
a,b,c = 2.1,-5.6,9.2
print(a,b,c)
```

# Variáveis

Uma vez que uma variável é criada, podemos utilizá-la em qualquer parte do programa para executar cálculos. Exemplo:

```
x = 2
y = 3
print("x = ", x )
print("y = ", y )
print("x + y =", x + y )
print("x*y =", x * y )
```

Também é possível usar a própria variável para atribuir a ela um novo valor. Tente o código abaixo:

```
z = 2
print(z)
z = z + 1
print(z)
```

## Observações

- As variáveis devem ser criadas **antes** de serem usadas no código.
- Python faz diferença entre letras maiúsculas e minúsculas. A variável X é diferente da variável x.
- Uma boa prática de programação é escolher nomes que tenham sentido e relação com a variável. Por exemplo, se uma variável se relaciona ao número de dias de uma semana, um bom nome seria dias_da_semana.
- O nome de uma variábel não pode ser um "palavra reservada" (*reserved keyword*). Veja a lista de palavras reservadas aqui: `https://www.w3schools.com/python/python_ref_keywords.asp`

# Variáveis - Exemplo

Vamos implementar a fórmula de Heron para calcular a área de um triângulo de lados a,b e c:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

onde $s = \frac{1}{2}(a+b+c)$

**Example**

```
import math as mt
a,b,c = 4.5,2.4,3.9
s = (a+b+c)/2.
area = mt.sqrt(s*(s - a)*(s - b)*(s - c))
print(area)
```

# Comparações

O resultado de uma comparação em Python é um objeto *booleano* que pode ter dois valores: **True** ou **False**. Os principais operadores de comparação são listados abaixo.

## Operadores de Comparação

< menor que
<= menor ou igual
== igual a
>= maior ou igual
> maior que
!= diferente

## Warning II

Não confundir o operador atribuição (=) com o operador de comparação (==)!

# Comparações

## Example

```
>>> 7 == 8
False
>>> 4 > 3.18
True
>>> var_comp = 5 ==2
>>> print(var_comp)
False
>>> "teste" != "teste1"
True
```

## Warning III

Nunca se deve comparar flutuantes com (==) ou (!=).

# Comparações

Flutuantes não devem ser testados com igualdade por causa da precisão finita. Se necessário, podelos usar a função `isclose` do módulo `math`.

## Example

```
>>> a = 0.01
>>> b = 0.1**2
>>> a == b
False
>>> import math as mt
>>> mt.isclose(a,b)
True
```

# Comparações

Em Python, quando um número é convertido para um tipo booleano, 0 é interpretato com **False** e qualquer outro valor é **True**.

## Example

```
>>> a = 0
>>> b = 57
>>> bool(a)
False
>>> bool(b)
True
```

# Comparações - O operador *in*

Python tem um operador especial conhecido como "operador in", escrito simplesmente como **in**. Ele testa se o lado esquerdo do operador está contido num coleção de números ou strings no lado direito. Teste os comandos abaixo.

```
print ("y" in "Python")
print ("x" in "Python")
print ("p" in "Python")
print ("th" in "Python")
print ("to" in "Python")
```

Também é possível testar se uma variável NÃO ESTÁ numa coleção, usado o operador **not in**. Exemplos:

```
print("y" not in "Python")
print("p" not in "Python")
```

# Condições - A palavra-chave *if*

Uma condição é um teste realizado pelo programa para saber se uma determina ação deve ou não ser executada. A maneira mais comum de se escrever condições num programa é usando a palavra **if**. A sintaxe (forma de escrever a expressão) é a seguinte:

**if** <condição >:
    <o_que_sera_executado >

É importante notar que:

1. Logo após a condição, devemos colocar **dois pontos** (:). Python interpretará que todo o comando abaixo deverá ser executado SE a condição for satisfeita.
2. Os comandos que serão executados devem ser colocados abaixo da condição, com **espaços a esquerda** (em geral, utilizamos quatro espaços). Todos as linhas que estiverem nessa margem são consideradas como dentro do "bloco if". Editores como spyder criarão esse espaço automaticamente, assim que a tecla <enter >for pressionada após (:).

# Condições - A palavra-chave *if*

**Example**

```
x = 7
if x < 10:
    print("Essa linha sera executada se  x < 10." )
    print("Mesmo que a linha acima.")
print("Essa linha sera sempre executada.")
```

**Exercício**

Teste o programa acima para x = 8,9,10 e 11.

# Condições - *and* e *or*

Algumas vezes precisamos testar mais de uma condição para uma variável. Isso poder ser feito utilizando os operadores **and** e **or**. O operador **and** indica que o comando só será executado se TODAS as condições forem satisfeitas, enquanto **or** indica que e execução ocorrerá se UMA DAS condições for satisfeita. Veja o exemplo abaixo.

## Example

```
x = 3
y = 5
if x == 3 and y ==5 :
    print("x é 3")
    print("y é 5")
if x < 5 or y > 5 :
    print("x está entre 0 e 5")
if x < 5 and y > 5 :
    print("x está entre 0 e 5")
```

# Condições - if else

Para fazer com que um programa em Python execute diferentes ações dependendo das condições, podemos usar a palavra **else**, formando um bloco "if else". A sintaxe é a seguinte:

if <condição >:
    <execução>
else:
    <execução>

Note que as palavras **if** e **else** devem estar alinhadas.

## Example

```
x = 4
if x > 2 :
    print ( "x é maior que 2" )
else:
    print ( "x é menor que 2" )
```

# Operações com Strings

Um *string* é uma sequência ordenada e imutável de caracters. Uma variável contento um string pode ser definida usando (“ ”) ou (' '):

```
p = "text1"
p = 'text1'
```

Podemos utilizar os operadores "+"e "*"para concatenar string

## Example

```
>>> "ab" + "cd"
'abcd'
>>> "56" + "1"
'561'
>>> "-o-"*4
'-o--o--o--o-'
>>> "a"*4 + "B"
'aaaaB'
```

# Operações com Strings

Para "tabular"ou pular uma linha, usamos os comandos `\t` e `\n`, respectivamente.

## Example

```
>>> a = "Coluna 1 \tColuna 2"
>>> print(a)
Coluna 1  Coluna 2
>>> areas = "Fisica\nQuimica\nBiologia"
>>> print(areas)
Fisica
Quimica
Biologia
```

# Operações com Strings - *Slicing*

Cada caracter de um string é associado a um *índice*, onde o primeiro caracter é 0, o segundo caracter é 1, o terceiro é 2, a assim por diante. Dados um string *s*, o comando `s[i:j]` retorna um substring entre os índices `i` e `j`, **incluindo** `i` mas **excluindo** `j`.

## Example

```
>>> s = 'Python3'
>>> s[:3]    #mesmo que s[0:3]
'Pyt'
>>> s[2:5]
'tho'
>> s[4:]
'on3'
>> s[::2] #mesmo que s[0:7:2] (passo de 2)
'Pto3'
```

# Operações com Strings - Métodos

Existem vários métodos para manipular strings. Veja uma lista extensiva aqui: `www.w3schools.com/python/python_ref_string.asp`

## Example

```
>>> a = 'Marie Curie'
>>>  a.upper()
'MARIE CURIE'
>>> a.lower()
'marie curie'
>>> a.swapcase()
'mARIE cURIE'
>>> a.replace('Curie', 'Sklodowska')
'Marie Sklodowska'
>>> a
'Marie Curie'   #a não é alterado!
```

# Operações com Strings - `format`

O método `format` é útil para formatar números.

## Example

```
>>> import math as mt
>>> print("O Valor aproximado de {} \\
    é {:.3f}".format('pi',mt.pi))
O Valor aproximado de pi é 3.142
>>> print("A raiz quadrada de {} é \\
    aprox. {:.4f}".format(8,mt.sqrt(8)))
A raiz quadrada de 8 é aprox. 2.8284
>>> x = 25.369**2
>>> print("x = {:.2e}".format(x))
x = 6.44e+02
```

{**:.3f**} significa formatar o argumento como um flutuante com três casas decimais.

# Iterações - while loops

Iterações são necessárias quando queremos que o programa repita uma operação várias vezes. Basicamente, existem duas maneiras de executar repetições: **while** loop e **for** loop.
No caso do comando **while**, a sintaxe é:

while <condição>:
    <comando>

Note que a sintaxe é semelhante ao **if**. O código acima fará com que <comando> seja executado enquanto <condição> for **True** (verdadeira). Se <condição> nunca for **False**, o programa entrará num loop infinito.

# Iterações - while loops

## Example

```
num = 1
while num <= 5:
    print( num )
    num += 1        #equivalente a num = num+1
print( "Fim!" )
```

Nesse programa, a condição "num <= 5" é testada, e cada vez que ela for True, o bloco dentro do **while** é executada, ou seja, o valor da variável num é impresso na tela e aumentado em uma unidade. Quando num for igual a 6, a condição será False, o loop para, e a linha print("Fim!") é executada.

# Iterações - for loops

Outra maneira bastante comum de implementar loops é através do comando **for**. A sintaxe é:

for <variaveis> in <coleção>:
    <comandos>

Uma coleção pode ser um string, uma variável ou números. Exemplos:

**Example**

```
for letter in "banana":
    print(letter)
print("Fim!")
#outra forma
fruta = "banana"
for letter in fruta:
    print(letter)
print("Fim!")
```

# Iterações - for loops

Para executar iterações sobre uma coleção de números, Python oferece a função **range()**. Essa função pode ser usada de três formas:

- range(x) – Será criada uma sequência de TODOS os números inteiros de 0 até x -1. Por exemplo, range(4) gerará a sequência (0,1,2,3).
- range(x,y) - Será criada uma sequência de números inteiros de x até y-1.
- range(x,y,z) – Nesse caso, x e y são os limites da sequência, e z é o passo. Por exemplo, range(1,11,2) criará (1,3,5,7,9). Note que o último número NÃO é incluído.

# Iterações - for loops

Teste os exemplos abaixo no jupyter.

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

```
for x in range(1,11,2):
    print(x)
```

```
for x in range(15,2,-2):
    print(x)
```

# Iterações - `enumerate` e `zip`

## Example

```
>>> nomes = ['Einstein','Darwin','Nash']
>>> for i, nm in enumerate(nomes):
        print(i,":",nm)
0 : Einstein
1 : Darwin
2 : Nash
```

## Example

```
>>> a = ['Einstein','Darwin','Nash']
>>> b = ['Fisico','Naturalista','Matematico']
>>> for par in zip(a,b):
        print(par)
('Einstein', 'Fisico')
('Darwin', 'Naturalista')
('Nash', 'Matematico')
```

# Controlando loops

É possível abortar um loop antes que ele execute todas as interações utilizando o comando **break**. Exemplo:

```
for nota in (7,8.5,9,7.8,5.6, 9.5,8):
    if nota < 7:
        print("Estudante não aprovado!")
        break
```

O loop será executado até que a condição nota $< 7$ seja encontrada (condição verdadeira). Nesse ponto, o programa imprimirá a mensagem acima e sairá do loop.

# Controlando loops

Para fazer com que um loop verifique todos os elementos de uma sequência, mas só processe parte deles, devemos usar o comando **continue**. Por exemplo, suponha que você queira saber quais números, entre 1 e 100, **NÃO** são divisíveis por 2 ou 3. O código para esse problema seria:

```
for i in range(101):
    if i%2 ==0:
        continue
    if i%3 ==0:
        continue
    print(i)
print("Fim!")
```

Os comandos **break** e **continue** só podem ser usados dentro de loops

# Loops dentro de loops

Loops podem ser criados dentro de loops. Nessas situações, é importante atentar para o **correto espaçamento** de cada comando, que definirá a qual loop ele pertente. Estude o exemplo abaixo.

```
for i in range(3):
    print("Entrando no primeiro loop para i =" ,i)
    for j in range(4):
        print("Entrando no segundo loop para j = ", j)
        print("i, j = ", i, j)
    print("Saindo do primeiro loop para i =" ,i)
```

# Loops - Exemplo

A série de Fibonacci é uma sequência de números onde o primeiro número é 1, o segundo também é 1, e os demais são a soma dos dois anteriores, ou seja: 1,1,2,3,5,8,13,21.... Vamos escrever um programa que calcule e imprima os *n* primeiros termos da série.

## Example

```
n = 20
a,b = 1,1
print(1,":",a)
print(2,":",b)
for i in range(3,n+1):
    a,b = b, a+b    #atenção aqui
    print(i,":",b)
```

**Exercício** Modifique o programa acima para imprimir também a razão entre cada termo e o seu antecessor (razão áurea). Use n = 40.
https://www.youtube.com/watch?v=1Jj-sJ78O6M&t=101s

# Definindo Funções

Filosofia: defina uma vez, use várias vezes.
Além das funções definidas nos módulos do Python, podemos criar nossas próprias funções. Funções são úteis quando precisamos executar o mesmo comando em vários lugares do código. Para definir uma função, a sintaxe é

**def** <nome_da_função> (parametros):
    <comandos>

## Exemplo 1

```
def googbye():
    print("Adeus Mundo!")
goodbye()
```

# Definindo Funções

## Exemplo 2 - Função com um parâmetro

```
def hello( nome ):
    print("Olá", nome)

hello("Adriana")
hello("Oscar")
```

## Exemplo 3 – Função com vários parâmetros

```
def multiplica(x, y):
    result = x*y
    print(result)

multiplica(2,4)
multiplica(5,27)
```

# Definindo Funções

O principal uso de uma função é retornar um valor para o programa principal. Para isso, devemos usar o comando **return**. Com esse comando, um valor calculado pela função pode ser guardado em uma variável e utilizado em outras partes do programa. O comando também termina a execução da função.

## Exemplo 4 – usando return

```
from math import sqrt

def pitagoras(a,b):
    return sqrt(a**2 + b**2)

c = pitagoras(3,4)
print(c)
```

Note a diferença entre **return** e **print()**.

# Definindo Funções

## Exemplo 5 – Retornos múltiplos

```
def circulo(raio):
    pi = 3.1415926536
    comprimento = 2*pi*raio
    area = pi*raio**2
    return comprimento, area

x, y = circulo(3.2)
print("Comprimento do círculo: ", x)
print("Área do círculo: ", y)
```

Note que nesse exemplo a função retorna **dois** valores.

# Definindo Funções - Argumento Nomeado

Nos exemplos anteriores, os argumentos foram passados na mesma ordem que são definidos na função (argumento posicional). No entanto, podemos passar os argumentos em qualquer ordem se os argumentos forem nomeados (*keyword arguments*).

## Exemplo 6 – Argumento Nomeado

```
def razao(x,y,z):
    return x/y + z

r = razao(y=2,z=0,x=1)
print(r)
```

# Definindo Funções - Argumento Nomeado

Também é possível usar ambos argumentos simultaneamente, mas o argumento posicional deve ser o **primeiro** listado.

## Exemplo 7 – Argumento Posicional e Nomeado

```
>>> razao(1,z=0,y=2)
0.5
>>> razao(z=0,2,x=1)
 File "<ipython-input-42-781af3970b9c>", line 1
razao(z=0,2,x=1)
          ^
SyntaxError: positional argument follows keyword
argument
```

# Definindo Funções - Argumentos *default*

Podemos definir um função com um argumento opcional. Se esse argumento não for passado para a função, ela usará o *default*.

## Exemplo 8 – Argumento *default*

```
>>> def comprimento(valor, unid='m'):
        return 'L = {:.2f} {}'.format(valor,unid)
>>> comprimento(25.439,'cm')
'L = 25.44 cm'
>>> comprimento(52.726)
'L = 52.73 m'
```

# Definindo Funções - Escopo

Quando uma variável é definida e usada dentro de uma função, ela é chamada variável **local**. Variáveis definidas fora do escopo de funções são chamadas **globais**, e podem ser acessadas em qualquer parte do programa.

## Exemplo 9 – Variável local

```
>>> def func():
        a = 5          #variavel local
        print(a)
>>> func()
5
>>> print(a)
NameError: name 'a' is not defined
```

Uma função pode acessar uma variável global, se a variável for definida **antes** da função ser chamada.

# Definindo Funções - Criando um Módulo

Para criar um módulo, escreva uma ou mais funções num arquivo e salve-o com extensão ".py". Para utilizar sua função, basta **importar** o módulo. O arquivo da função deve estar no mesmo diretório do seu programa principal.

## Exemplo 10 – Meu Módulo

```
#arquivo meu_modulo.py
import math as mt
def area_retangulo(a,b):
    return a*b

def area_hexagono(lado):
    return 3*lado**2*mt.sqrt(3)/2.
```

# Definindo Funções - Criando um Módulo

Agora vamos chamar o módulo criado

## Exemplo 10 – continuação

```
>>> from meu_modulo import area_retangulo, \\
    area_hexagono        #mesma linha
>>> area_retangulo(2.5,3)
7.5
>>> area_hexagono(1.25)
4.059494080239556
```

# Definindo Funções - Função Anônima

É possível criar uma função sem nome (anônima) usando a expressão **lambda**. A função é atribuída a uma variável, que pode ser usada como uma função. A sintaxe é:

```
lambda <parametros>: <comando>
```

## Exemplo 11 – Função Anônima

```
>>> f = lambda x: x**2
>>> f(2)
4
```

O código acima é exatamente o mesmo que:

```
def f(x):
    return x**2
f(2)
```

# Listas

Em Python, uma **lista** é um conjunto ordenado de objetos que podem ser de vários tipos (inteiro, flutuante, complexo, boleano, string, etc.). Por exemplo, para criar uma lista, fazemos:

```
L = [1, 2.5, "Olá",True]
```

Cada entrada da lista é chamada de **elemento**, cada elemento tem uma **posição** na lista, e cada posição tem um inteiro associada a ela. Assim, o número (elemento) 1 está na posição zero da lista, o número 2.5 na posição um, e assim por diante. O **índice** que indica uma posição na lista sempre começa em zero.
Uma lista é um objeto **mutável**, e portanto podemos acrescentar ou retirar um elemento da lista.
Podemos também criar uma lista vazia: `L0 = []`.

# Listas

Um elemento da lista pode ser acessado pelo seu índice. O operador **in** pode ser usado para verificar se um dado elemento pertence a lista.

## Example

```
>>> L = [1,2.5,5.69,"x"]
>>> L[0]
1
>>> L[3]
'x'
>>> L[-1]
'x'
>>> 2 in L
False
>>> 'x' in L
True
```

# Listas - objetos mutáveis

Como lista são mutáveis, é possível modidficar itens da lista.

## Example

```
>>> L = [1,'dois',3.14,0]
>>> L[2] = 2.6
>>> L
[1,'dois',2.6,0]
```

Atenção ao exemplo abaixo

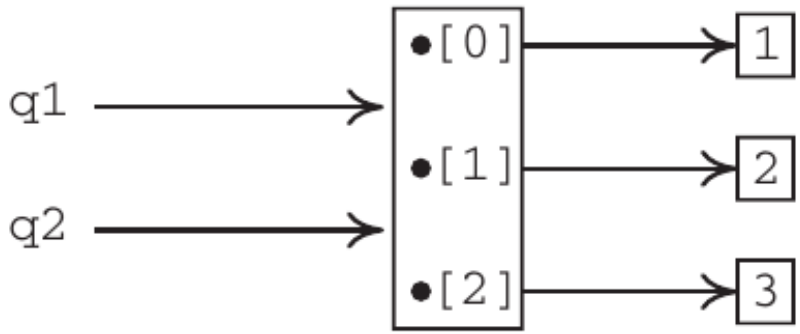
## Example

```
>>> q1 = [1,2,3]
>>> q2 = q1
>>> q1[2] = 'x'
>>> q1
[1,2,'x']
>>> q2
[1,2,'x']
```

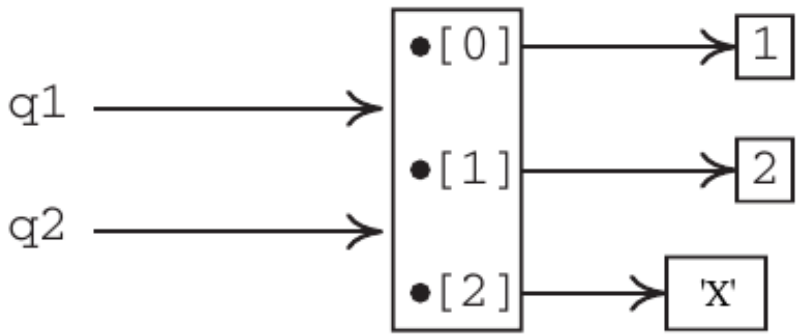
# Listas - objetos mutáveis

## Warning IV

No exemplo acima, as variáveis q1 e q2 se referem a **mesma lista**, e apontam para o mesmo local da memória. Modificar q1 automaticamente modifica q2.




q1 = q2




q1[2] = 'X'

Para criar um cópia de uma lista, podemos usar o método `copy()`.
Exemplo: `q3 = q1.copy()`.

# Listas - Métodos

Existem vários métodos que podem ser usados com listas. Exemplos de alguns métodos:

- append() - adiciona um elemento ao final da lista.
- insert() - semelhante ao append(), mas podemos escolher a posição onde o novo elemento será alocado. Exemplo: L.insert(1,4.56)
- remove() - remove um elemento específico que está na lista. Exemplo: L.remove(4.56)
- pop() - remove um elemento da lista, dado sua posição. Exemplo: L.pop(1) vai remover o elemento que está no posição "1"da lista, ou seja, o segundo elemento. L.pop() remove o último elemento da lista.
- index()- retorna o índice da primeira ocorrência de um elemento da lista (posição do elemento). Exemplo: L.index(2.5)
- sort() - ordena os elementos de uma lista em ordem crescente.
- reverse() - inverte a ordem dos elementos da lista.

# Listas - Exemplos

## Example

```
>>> import math as mt
>>> L = []
>>> for i in range(5):
        L.append( round(mt.sqrt(i**2.5),2) )
>>> L
[0.0, 1.0, 2.38, 3.95, 5.66]
>>> L.insert(1,5.3)
>>> L
[0.0, 5.3, 1.0, 2.38, 3.95, 5.66]
>>> L.sort()
>>> L
[0.0, 1.0, 2.38, 3.95, 5.3, 5.66]
```

# Listas - Exemplos

O método de string `split()` retorna uma lista com elementos do string original.

**Example**

```
>>> s = 'A,B,C,D,E'
>>> L = s.split(',')
>>> L
['A', 'B', 'C', 'D', 'E']
```

**Exercícios**

1. Dado as listas L1 = [1,2,3] e L2 = [5.4,2.1,0], verifique o resultado do comando L1 + L2.
2. Verifique e interprete o resultado dos seguintes comandos: `any(L2)` e `all(L2)`, com L2 dado acima.

# Listas - Exemplos

Podemos criar um lista de funções usando a expressão **lambda**

## Example

```
>>> flist = [lambda x: 1,
             lambda x: x,
             lambda x: x**2,
             lambda x: x**3]
>>> flist[2](3)  #flist[2] é x**2
9
```

**Compreensão de Lista**

## Example

```
>>> L = [x**2 for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> L = [x**2 for x in range(10) if x%2==0]; L
[0, 4, 16, 36, 64]
```

# Tuplas

Uma tupla pode ser definida como uma lista **imutável**. Uma tupla é construída com ítens encerrados por parênteses:

```
t = (1.5,'k',4)
```

Os elementos da tupla podem ser acessados da mesma forma que as listas. No entanto, um elemento da uma dupla não pode ser alterado ou removido:

```
>>> t[0]
1.5
>>> t[0] = 3.1
TypeError: 'tuple' object does not support item
assignment
```

Para criar um tupla com um único elemento, fazemos `t = (2,)`.

# Dicionários

Em Python, um dicionário é uma conjunto de objetos que são indexados por *chaves*. Dicionários formam uma coleção de de pares (chave, valor), e são objetos **mutáveis**. Um dicionário pode definido da seguinte forma: `{chave:valor}`. A chave der ser um objeto **imutável**.

## Example

```
>>> numeros = {'primeiro':1, 'segundo':2,
               'terceiro':3}
>>> numeros
{'primeiro': 1, 'segundo': 2, 'terceiro': 3}
>>> numeros['primeiro']
1
>>> numeros['segundo'] = 3
{'primeiro': 1, 'segundo': 3, 'terceiro': 3}
```

Note que cada chave é **única**.

# Dicionários

Uma outra forma de criar um dicionário é passar o par (chave,valor) para o construtor **dict**.

## Example

```
>>> num = dict([('primeiro',1),('segundo',2),
                ('terceiro',3)])
>>> num
{'primeiro': 1, 'segundo': 2, 'terceiro': 3}
```

Um *loop* sobre um dicionário retorna as chaves.

## Example

```
>>> for k in num:
        print(k,'\t',num[k])
primeiro    1
segundo     2
terceiro    3
```

# Dicionários - Métodos

**Método get()**: usado para obter o valor correspondente a uma chave, ou retorna valor *default* caso a chave não exista.

**Example**

```
>>> massa = {'mercurio':3.3e23,'venus':4.87e24,
    'terra':5.97e24}
>>> massa.get('terra')
5.97e+24
>>> massa.get('plutao',-1)
-1
```

**Métodos keys(), values() e items()**: retornam, respectivamente, todas as chaves do dicionário, todos os valores e todos os pares chave-valor (como tuplas). Esses métodos retornam *iterable objects*.

# Dicionários - Métodos

## Example

```
>>> planetas = massa.keys()
>>> planetas
dict_keys(['mercurio', 'venus', 'terra'])
>>> for p in planetas:
        print(p,end=' ')
mercurio venus terra
```

Pode-se transformar as chaves numa lista usando o construtor **list**

## Example

```
>>> lista_planetas = list(massa.keys())
>>> lista_planetas
['mercurio', 'venus', 'terra']
```

Veja o exemplo `exemplo01_planetas.ipynb`.

# NumPy Arrays

**NumPy** é o pacote padrão para programação científica em Python. O módulo NumPy implementa de forma eficiente operações matemáticas. Para usar os métodos do módulo, devemos importá-lo no início do programa:

```
import numpy as np
```

Os objetos do NumPy são **arrays**, que é um conjunto ordenado de valores, mas que possuem diferenças crucias em relação a listas:

- O número de elementos de um array é **fixo**. Não se pode adicionar ou remover itens de um array.
- Os elementos de um array são todos do mesmo tipo.
- Arrays podem ter $n$ dimensões. Por exemplo, arrays com $n = 2$ são matrizes.
- Operações com arrays são **mais rápidas** do que com listas.

# Criando Arrays

Vamos ver diversas formas de criar um array.

## Array a partir de lsitas ou tuplas

```
>>> a = np.array([1.,2,3.1])
>>> a
array([1. , 2. , 3.1])
>>> a[0]
1.0
>>> b = np.array([ [1.,2.],[3.,4.] ])  #2D array
>>> b
array([[1., 2.],
       [3., 4.]])
>>> b[0,0]
1.0
>>> b[1,0]
3.0
```

# Criando Arrays

## Array com todas as entradas iguais a zero

```
>>> np.zeros(5)
array([0., 0., 0., 0., 0.])
>>> np.zeros(5,dtype=int)
array([0, 0, 0, 0, 0])
```

## Array com todas as entradas iguais a um

```
>>> np.ones(6)
array([1., 1., 1., 1., 1., 1.])
>>> np.ones((3,4))    #matrix 3 x 4
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

# Criando Arrays

## Array com todas as entradas iguais a um dado valor

```
>>> np.full(4, 3.14)
array([3.14, 3.14, 3.14, 3.14])
>>> np.full((4,3), 3.14)
array([[3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14]])
```

## Array como matrix identidade

```
>>> np.eye(3)      #mesmo que np.identity(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

# Criando Arrays

## Array "vazio"

```
>>> np.empty(6)
array([1., 1., 1., 1., 1., 1.])
```

## Criando array com o método logspace()

```
>>> np.logspace(0,2,4)
array([  1., 4.64158883, 21.5443469 , 100.])
#4 valores entre 10**0 e 10**2
```

## Criando array com o método random()

```
>>> np.random.random((3,3))
array([[0.19571846, 0.567335  , 0.10199358],
       [0.68696384, 0.40271097, 0.57285044],
       [0.31473533, 0.63296434, 0.2625094 ]])
#array 3x3 com numeros entre 0 e 1.
```

# Criando Arrays

## Criando array com o método arange()

```
>>> np.arange(7)
array([0, 1, 2, 3, 4, 5, 6])
>>> np.arange(1.5,3.0,0.5)
array([1.5, 2. , 2.5])
>>> np.arange(6.5,0,-1)
array([6.5, 5.5, 4.5, 3.5, 2.5, 1.5, 0.5])
```

A sintaxe do método **arange()** é np.arange(inicio, fim, passo). Se apenas um número for dado, por exemplo, np.arange(N), será criado um array de zero até o valor N-1, com passo de um.

# Criando Arrays

A função `np.linspace(x,y,N)` gera $N$ números entre x e y, com y **incluso**.

## Criando array com o método linspace()

```
>>> np.linspace(0,10,6)
array([ 0.,  2.,  4.,  6.,  8., 10.])
>>> z,dz = np.linspace(0.,2*np.pi,100,retstep=True)
>>> dz
0.0634665182543926
```

A opção `retstep = True` retorna o tamanho do passo.

## Warning V

Note a diferença entre arange() e linspace(). Use linspace() sempre que desejar um array de tamanho precisamente $N$.

# Criando Arrays

Podemos criar arrays usando funções anônimas pelo método `np.fromfunction()`. Os argumentos devem ser um função e uma tupla com a forma do array desejado. Note que o número de argumentos da função deve ser o mesmo da dimensão do array.

## Criando array com o método fromfunction()

```
>>> np.fromfunction(lambda i,j: 2*i*j, (4,3))
array([[ 0.,  0.,  0.],
       [ 0.,  2.,  4.],
       [ 0.,  4.,  8.],
       [ 0.,  6., 12.]])
```

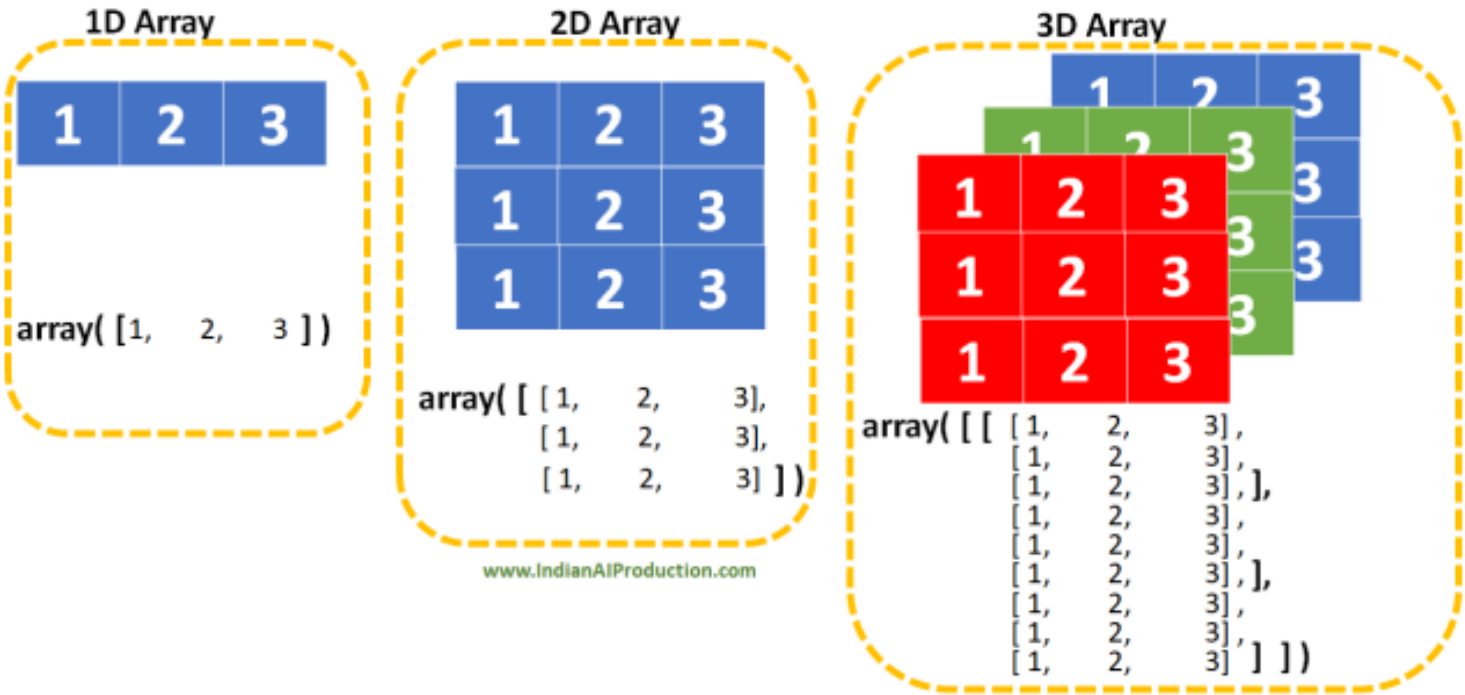
- Note que `i` varia de 0 a 3, e `j` de 0 a 2
- Note que cada par (i,j) é uma entrada da matrix.

# Arrays - Atributos

## Atributos de um array

```
>>> a = np.array([ [1,0,1], [1,2,2] ])
>>> a.shape
(2, 3)
>>> a.ndim
2
>>> a.size
6
>>> a.dtype
dtype('int64')
>>> a.nbytes
48
```

# Arrays - Arrays Multidimensionais




**Exemplo:** Teste `np.random.random((2,3,4))`.

# Operações com Arrays

O grande poder do NumPy reside na realização de operações em todos os elementos de um array sem a necessidade de *loops* explícitos. Esse tipo de operação é chamada **vetorização**, e é muito mais rápida que *for loops*.

## Example

```
>>> a = np.array([1.3,2.5,10.1])
>>> b = np.array([9.3,0.2,1.2])
>>> a + b
array([10.6, 2.7, 11.3])
>>> a*b
array([12.09, 0.5, 12.12])
>>> a/b
array([0.13978495, 12.5, 8.41666667])
>>> a/b + 1
array([ 1.13978495, 13.5,  9.41666667])
>>> a**2
array([1.69, 6.25, 102.01])
```

# Operações com Arrays

Além de mais rápido, a sintaxe do NumPy é mais concisa.

## Cálculo com for loop

```
def calc_inverso(vec):
    result = np.empty(len(vec))
    for i in range(len(vec)):
        result[i] = 1./vec[i]
    return result
In [1]: a = np.random.random(100000) #100k entradas
In [2]: %timeit calc_inverso(a)
20.6 ms ± 86.1 µs per loop
```

## Cálculo vetorizado

```
In [1]: %timeit 1./a
67 µs ± 74 ns per loop
```

**O cálculo vetorizado é 300x mais rápido!!**

# Operações com Arrays

## Produtos

```
>>> a = np.array( [1.,2.,3.])
>>> b = np.array( [2.,4.,5.])
>>> np.dot(a,b) # produto interno, (mesmo que a @ b)
25.0
>>> np.cross(a,b)   #produto vetorial
 array([-2.,  1.,  0.])
```

## Operadores de comparação e lógica

```
>>> a = 2*np.linspace(1,6,6)
>>> a
array([ 2.,  4.,  6.,  8., 10., 12.])
>>> t = a > 10
>>> t
array([False, False, False, False, False,  True])
```

# Operações com Arrays

**Exemplo**: Vamos implementar o cálculo abaixo:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix} + 2 \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 3 & -5 \\ 0 & 2 \end{pmatrix}$$

## Código

```
>>> a = np.array([ [1,3], [2,4] ])
>>> b = np.array([ [4,-2],[-3,1] ])
>>> c = np.array([ [1,2], [2,1] ])
>>> r = np.dot(a,b) + 2*c
>>> r
array([[-3,  5],
       [ 0,  2]])
```

# Operações com Arrays - Funções

As funções disponíveis no módulo math também existem no NumPy. Teste os exemplos abaixo.

## Funções

```
theta = np.linspace(0.1,np.pi,4)
print("theta = ", theta)
print("sen(theta) = ",np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("ln(theta) = ",np.log(theta))
print("log(theta) = ",np.log10(theta))
print("exp(theta) = ", np.exp(theta))
print("modulo = ", np.absolute(np.log(theta)))
```

# Arrays - Funções de Agregação

Quando trabalhamos com um grande conjunto de dados, é comum calcularmos estatísticas para uma análise inicial. NumPy oferece vários métodos para essa tarefa.

## Example

```
#soma, media, max. e min.
>>> data = np.random.random(100)
>>> data.sum()
51.72239489031435
>>> data.mean()
0.5172239489031435
>>> data.max()
0.9946152525979709
>>> data.min()
0.0023509304159052835
```

# Arrays - Funções de Agregação

Para arrays em n-dimensões (n $>$ 1), podemos escolher o *eixo* sobre o qual os valores serão agregados.

## Example

```
#2D array
>>> M = np.random.random((3,4))
>>> M
array([[0.37019599, 0.15892146, 0.23032805, 0.37...],
       [0.17968684, 0.69242006, 0.51502879, 0.06...],
       [0.78280796, 0.63324658, 0.22553994, 0.94...]])
>>> M.sum()
5.168637614536063
>>> M.sum(axis=0)
array([1.33269079, 1.4845881 , 0.97089679, 1.380...])
>>> M.max(axis=1)
array([0.37897901, 0.69242006, 0.94005747])
```

# Arrays - Funções de Agregação

| Método | Descrição |
|---|---|
| `np.sum` | soma dos elementos |
| `np.cumsum` | soma cumulativa dos elementos |
| `np.prod` | produto dos elementos |
| `np.mean` | valor médio |
| `np.std` | desvio padrão |
| `np.var` | variância |
| `np.min` | valor mínimo |
| `np.max` | valor máximo |
| `np.argmin` | **indice** do valor mínimo |
| `np.argmax` | **indice** do valor máximo |
| `np.conj` | complex. conjugado de todos elementos |
| `np.trace` | soma dos elementos da diagonal |

Veja mais detalhes neste LINK.

# Arrays - Mudando a forma

É possível "achatar" um array, ou seja, reduzir um array multidimensional a um array 1D. Dois métodos existem: `flatten` e `ravel`.

## Método `flatten()`

```
>>> a = np.array([ [1,2,3], [4,5,6], [7,8,9] ]); a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b = a.flatten() #retorna uma cópia dos elementos
>>> b
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b[3] = 0; b
array([1, 2, 3, 0, 5, 6, 7, 8, 9])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

# Arrays - Mudando a forma

## Método `ravel()`

```
>>> c = a.ravel(); c
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> c[3] = 0
>>> c
array([1, 2, 3, 0, 5, 6, 7, 8, 9])
>>> a
array([[1, 2, 3],
       [0, 5, 6],
       [7, 8, 9]])
```

## Warning VI

Note que `flatten` retorna um array **independente**, enquanto `ravel` retorna um array que se refere aos **mesmos elementos** do array original.

# Arrays - Mudando a forma

Os métodos `resize` e `reshape` mudam a forma do array mantendo o mesmo número de elementos (size).

## Método `resize()`

```
>>> a = np.linspace(1,4,4)
>>> a
array([1., 2., 3., 4.])
>>> a.resize(2,2)
>>> a
array([[1., 2.],
       [3., 4.]])
```

Note que `resize` muda a forma do array **original**.

# Arrays - Mudando a forma

## Método `reshape()`

```
>>> a = np.linspace(1,4,4)
>>> b = a.reshape(2,2)
>>> b
array([[1., 2.],
       [3., 4.]])
>>> a
array([1., 2., 3., 4.])
>>> b[0,0] = - 20
>>> b
array([[-20.,   2.],
       [  3.,   4.]])
>>> a
array([-20.,   2.,   3.,   4.])
```

Note que **a** e **b** apontam para os mesmos elementos.

# Arrays - Array Transposto

## Método `transpose()`

```
>>> a = np.linspace(1,6,6).reshape(3,2)
>>> a
array([[1., 2.],
       [3., 4.],
       [5., 6.]])
>>> a.transpose()   #ou a.T
array([[1., 3., 5.],
       [2., 4., 6.]])
```

**Exercício**: Verifique se a matriz $M$ abaixo é ortogonal, ou seja, se $M^T M = \mathrm{I}$, onde $\mathrm{I}$ é a matriz identidade.

$$M = \frac{1}{\sqrt{6}} \begin{pmatrix} \sqrt{2} & 1 & \sqrt{3} \\ -\sqrt{2} & 2 & 0 \\ \sqrt{2} & 1 & -\sqrt{3} \end{pmatrix}$$

# Arrays - Merging

NumPy tem vários métodos para "juntar" arrays.

## Example

```
>>> a = np.zeros(4)
>>> b = np.ones(4)
>>> c = 2*np.ones(4)
>>> np.vstack((a,b,c))
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.]])
>>> np.hstack((a,b,c))
array([0., 0., 0., 0., 1., 1., 1., 1., 2., 2., 2., 2.])
```

**Exercício:** Teste e explique o resultado dos seguintes comandos: `np.hsplit(b,2)` e `np.hsplit(b,3)`.

# Arrays - Slicing

Muitas vezes precisamos obter um "subarray" a partir de um array, ou seja, um array com apenas alguns elementos do array original. Para isso, existe uma técnica chamada **slicing**. A sintaxe é:

**[inicio:fim:passo]**

onde "início" é o índice (posição) da primeira entrada desejada, e "fim" o índice do último elemento, que NÃO entrará no novo array. Esse comando vai gerar um array com entradas a[inicio], a[inicio + passo], a[inicio+2*passo], ....a[inicio+N*passo], com a posição "inicio+N*passo" $<$ fim.
O array que retorna dessa operação **não** é um cópia, ou seja, não é um novo objeto.

# Arrays - Slicing

## Example

```
>>> a = np.linspace(1,6,6); a
array([1., 2., 3., 4., 5., 6.])
>>> a[:3]   #mesmo que a[0:3]
array([1., 2., 3.])
>>> a[1:4:2]
array([2., 4.])
>>> a[1:]
array([2., 3., 4., 5., 6.])
>>> a[3::-2]
array([4., 2.])
>>> a[::-1]
array([6., 5., 4., 3., 2., 1.])
```

# Arrays - Slicing

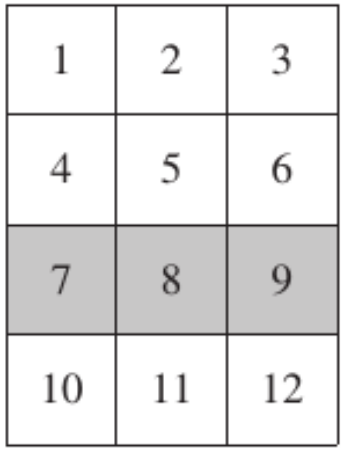
Em arrays multidimensionais, podemos aplicar o *slicing* em cada dimensão.

## Example

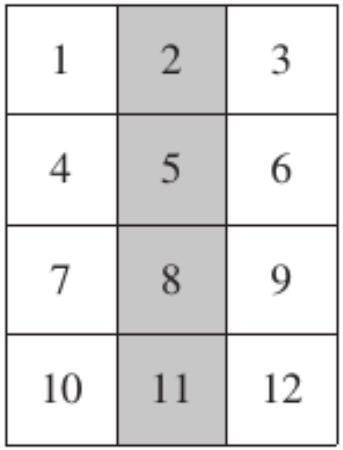
```
>>> a = np.linspace(1,12,12).reshape(4,3); a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.],
       [10., 11., 12.]])
>>> a[2,:]    #todos na terceira linha
array([7., 8., 9.])
>>> a[:,1]   #todos na segunda coluna
array([ 2.,  5.,  8., 11.])
>>> a[1:-1,1:]   #segunda e terceira linhas!
array([[5., 6.],
       [8., 9.]])
```

**Exercício**: Teste `a[:,1] = 0`.

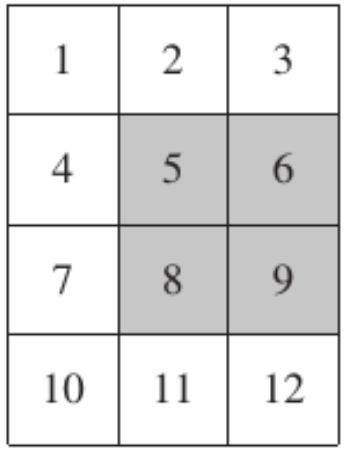
# Arrays - Slicing



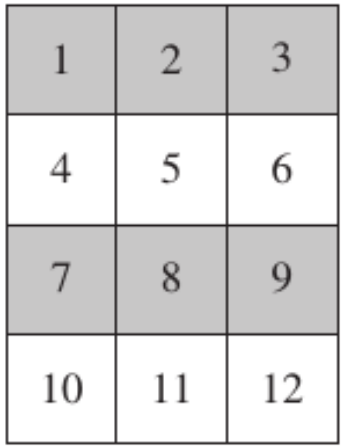

`a[2, :]`



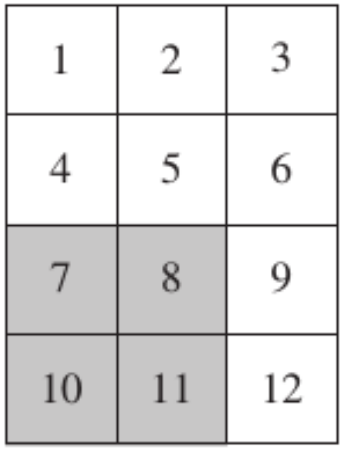

`a[:, 1]`



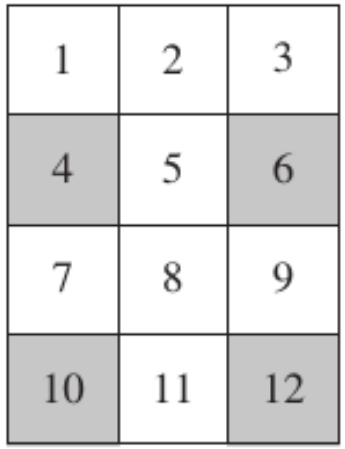

`a[1:-1, 1:]`




`a[::2, :]`




`a[2:, :2]`




`a[1::2, ::2]`

# Arrays - Indexação Avançada

É possível indexar arrays usando listas ou arrays de inteiros. Nesse caso, um novo array é criado.

## Example

```
>>> a = np.linspace(0.,0.5,6)
>>> a
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5])
>>> ia = [1,4,5]
>>> a[ia]
array([0.1, 0.4, 0.5])
```

Também podemos usar operadores de comparação

## Example

```
>>> b = np.linspace(-2.5,2.5,5); b
array([-2.5 , -1.25,  0.  ,  1.25,  2.5 ])
>>> b[b>0]
array([1.25, 2.5 ])
```

# Arrays - Ordenação

Podemos ordenar arrays com o método **sort()**. Para arrays 2D ou superior, devemos especificar o eixo.

## Example

```
>>> a = np.array( [5,-1,2,4,0,4])
>>> a.sort()
>>> a
array([-1,  0,  2,  4,  4,  5])
>>> b = np.array ([[0 , 3, -2], [7, 1, 3], [4, 0, -1]])
array([[ 0,  3, -2],
       [ 7,  1,  3],
       [ 4,  0, -1]])
>>> b.sort(axis=0); b
array([[ 0,  0, -2],
       [ 4,  1, -1],
       [ 7,  3,  3]])
```

# Arrays - Exemplo

**Exemplo**: Dados dois arrays de posição $x$ e tempo $t$ de uma partícula, calcule a velocidade média $\bar{v}$ para cada intervalo de tempo, utilizando *slicing*.

```
x = np.array([0., 1.3,  5. , 10.9, 18.9, 28.7, 40.])
t = np.array([0., 0.49, 1. , 1.5 , 2.08, 2.55, 3.2])
```

Com

$$\bar{v} = \frac{x_i - x_{i-1}}{t_i - t_{i-1}}$$

# Importando e Exportando Dados

**Abrindo arquivos com NumPy**

Para abrir arquivos de dados dos tipos `.txt`, `.dat` ou `.csv`, podemos usar o métodos **np.loadtxt()**. Os dados serão transformados num array. Como default, é assumido que os dados estão separados por espaços ou tabulação.

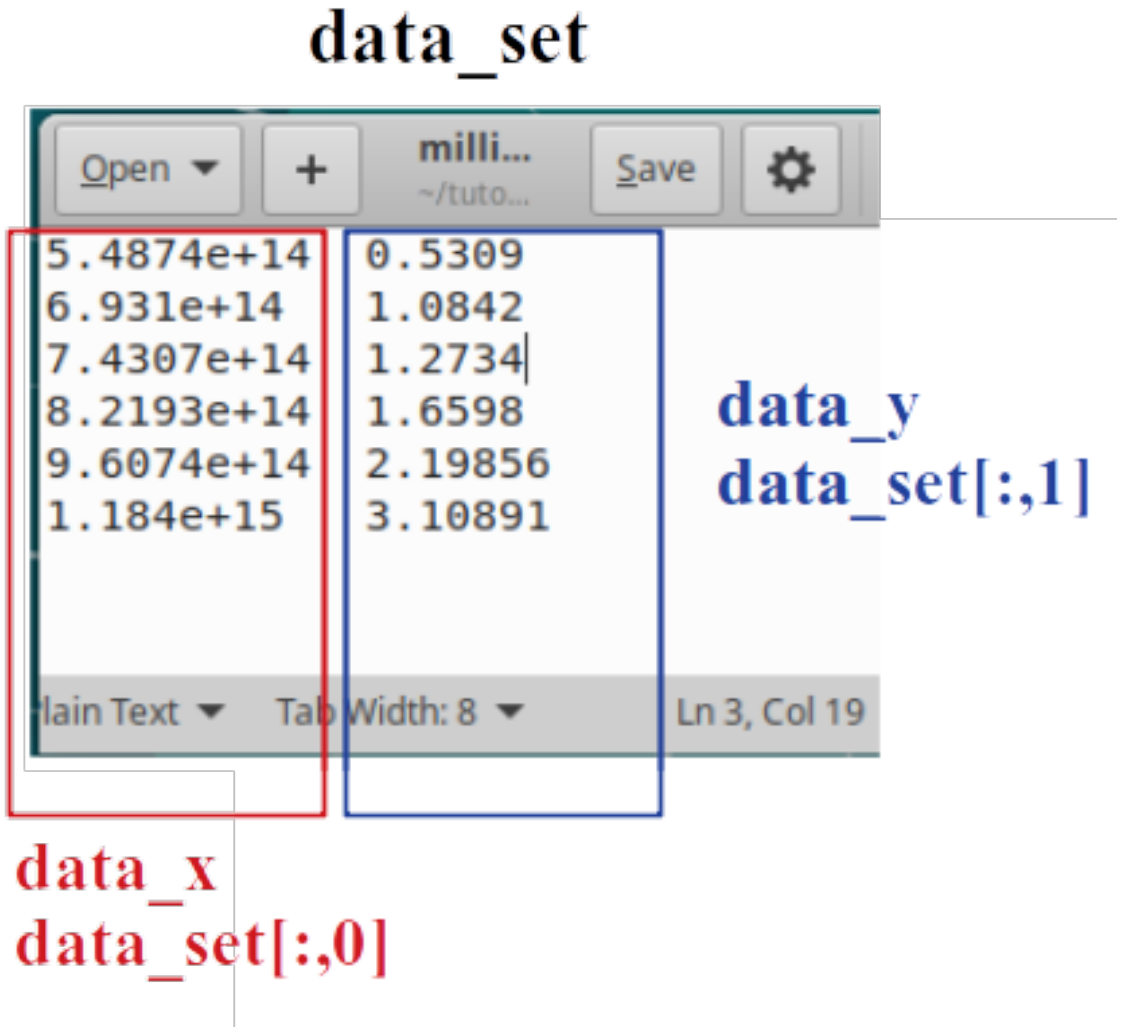
```
import numpy as np
data_set = np.loadtxt("millikan.txt")
data_x = data_set[:,0]
data_y = data_set[:,1]
```

Se os valores estiverem separados por um caractere, ele dever ser especificado usando a palavra chave `delimiter`.

```
data_set = np.loadtxt("millikan.csv",delimiter=',')
```

# Importando e Exportando Dados

A figura ilustra o exemplo acima.



# Importando e Exportando Dados

Também é possível importar dados diretamente da web:

```
import urllib
web_file = urllib.request.urlopen("http://www-personal
.edu/~mejn/cp/data/millikan.txt")
data_set = np.loadtxt(web_file)
```

Para ler o arquivo a partir de uma determinada linha, use a opção:
`skiprows=<inteiro>`

# Importando e Exportando Dados

**Abrindo arquivos com open()**

Para abrir arquivos em formatos menos "amigáveis", usamos a função **open()**. O arquivo será lido linha por linha como um **string**. Por exemplo, suponha que um arquivo "HIVseries.csv" tenha os valores separados por vírgula.

```
my_file = open('HIVseries.csv')
data_col1 = []
data_col2 = []
for line in my_file:
    list_data = line.split(',')  #retorma uma lista
    data_col1.append( float(list_data[0]) )
    data_col2.append( float(list_data[1]) )
my_file.close()
```

# Importando e Exportando Dados

Salvando dados em um arquivo.

## Usando o método savetxt()

```
x = np.linspace(0,1,100)
y = 3*np.sin(x)**3 - np.sin(x)
np.savetxt("x_valores.dat",x)
np.savetxt("xy_values.dat",list(zip(x,y)),fmt="%8.3f")
```

A última linha salva os valores 'x' e 'y' num mesmo arquivo.

## Usando *Loops*

```
my_file = open("xy_values.txt","w")  #w=writing
for i in range(x.size):
    my_file.write("{:f}\t\t{:f}\n".format(x[i],y[i]))
my_file.close()
```

# Contornando Erros

Quando manipulados dados em um programa, podemos cair numa operação que resulte num erro e o fluxo de comandos é interrompido (*exception*). Por exemplo:

```
>>> s = 'NA'
>>> float(s)
ValueError: could not convert string to float: 'NA'
```

Para contornar essa situação, podemos usar os comandos **try** e **except**, e devemos especificar o tipo de erro que queremos contornar.

## Example

```
try:
    n = float(s)
    print('string foi convertido')
except ValueError:
    print('Coversão não possível')
```

# Importando e Exportando Dados - Exemplo

**Exemplo**: Dados de diversas estações meteorológicos do Rio de Janeiro podem ser baixados do site **Alerta Rio**. Baixe os dados da estação "Jardim Botânico", de 2019. Serão doze arquivos, um para cada mês do ano, com diversas informações e resolução temporal de 15 min. Escreva um programa que leia os arquivos e produza dois arrays, um com a precipitação total mensal, e outro com a temperatura média mensal.

Veja a solução no arquivo `exemplo03_meteorologico.ipynb`.

# Gráficos - Matplotlib

Python tem uma poderosa biblioteca para produção de gráficos de boa qualidade: **Matplotlib**. Para gráficos simples, podemos usar o módulo **pyplot** que deve ser importado da seguinte forma:

```
import matplotlib.pyplot as plt
```

No jupyter, para que o gráfico apareça numa célula do notebook, digite na primeira célula: `%matplotlib inline`. Para que o gráfico seja mostrado numa janela separada, digite na primeira célula: `%matplotlib`.

# Gráficos - Gráfico de Funções

Se quisermos fazer um gráfico de uma função, as entradas para o pyplot devem ser arrays (ou listas) correspondentes aos valores x e y. Exemplo:

## Exemplo 1 - Gráfico simples

```
x = np.linspace(-3*np.pi,3*np.pi,100)
y = np.sin(x)    #vetorização
plt.plot(x,y)
```

Para adicionar um segundo plot, basta chamar `plt.plot` novamente:

```
z = np.cos(x)
plt.plot(x,z)    #ou plt.plot(x,y,x,z)
```

# Gráficos - Legenda

Para nomear um gráfico, devemos atribuir um string ao argumento `label` da função `plot`. Para adicionar a legenda no gráfico, faça: `plt.legend()`

## Exemplo 2 - Legenda

```
x = np.linspace(-3*np.pi,3*np.pi,100)
y = np.sin(x)
z = np.cos(x)
plt.plot(x,y,label='sen(x)')
plt.plot(x,z,label='cos(x)')
plt.legend()
```

Para retirar a legenda da "caixa", use a opção `frameon=False`. Para selecionar o tamanha da fonte, use `fontsize=<inteiro>`.

# Gráficos - Legenda

Opções de localização da legenda

| String | Inteiro |
|---|---|
| 'best' | 0 |
| 'upper right' | 1 |
| 'upper left' | 2 |
| 'lower left' | 4 |
| 'lower right' | 4 |
| 'right' | 5 |
| 'center left' | 6 |
| 'center right' | 7 |
| 'lower center' | 8 |
| 'upper center' | 9 |
| 'upper center' | 10 |

# Gráficos - Estilo LaTeX

Para usar comandos LaTeXna sua legenda, adiciona a seguinte opção no início do seu notebook:

```
plt.rc('text', usetex=True)
```

Mas para funcionar corretamente, você precisa instalar o pacote cm-super (no Ubuntu):

```
sudo apt install cm-super
```

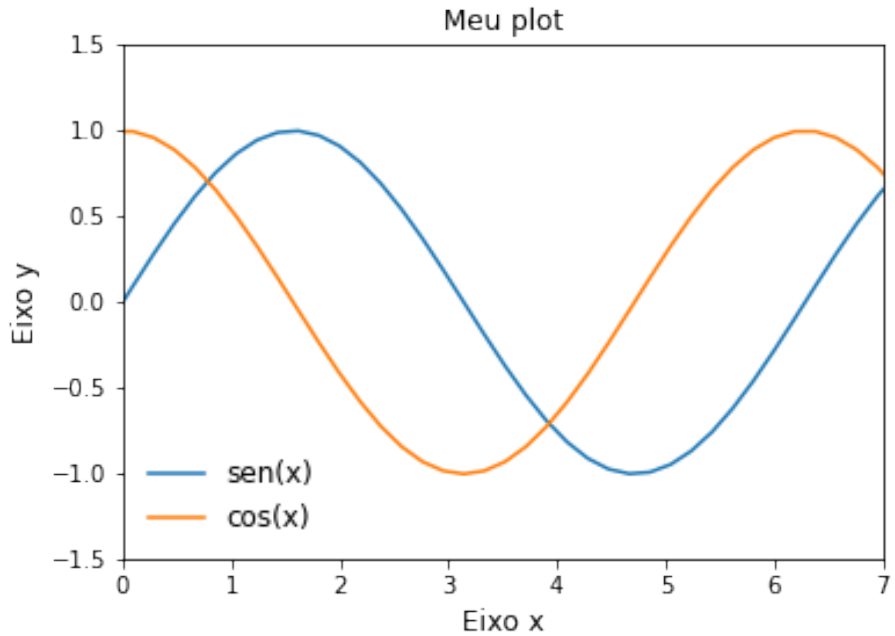
Quando escrever a legenda, coloque a letra `r` antes do string:

```
plt.plot(x,y,label=r'$\phi^2$')
plt.plot(x,z,label=r'$F_4^2$')
```

# Gráficos - Descrição dos Eixos e Título

## Exemplo 3 - Eixos e Título

```
plt.title('Meu plot')
plt.xlabel('Eixo x',fontsize=12)
plt.ylabel('Eixo y',fontsize=12)
plt.xlim(0, 7.)      #x-range
plt.ylim(-1.5,1.5)   #y-range
```



# Gráficos - Marcadores, Cores e Linhas

Existem diversas opções de marcadores, linhas e cores, que devem ser especificados por strings. Por exemplo, se quisermos linha vermelha tracejada, basta incluir 'r- -' na função plot.

## Exemplo 3 - Cores e Linhas

```
plt.plot(x,y,'r--',label='sen(x)')
plt.plot(x,y,'r--o',label='sen(x)')  #marcador 'o'
```

Também é possível passar os atributos explicitamente com `c` (color), `marker` (marcador) `ls` (estilo da linha) e `lw` (largura da lilnha)

```
plt.plot(x,y, c='r',marker='o',ls='--',lw=2)
```

Também é possível selecionar o tamanho do marcador (`markersize`), a cor (`markerfacecolor` ou `mfc`), e a cor da borda (`markeredgecolor` ou `mec`).

# Gráficos - Marcadores, Cores e Linhas

## Marcadores

| Código | Marcador |
|---|---|
| `.` | Ponto |
| `o` | Círculo |
| `+` | Cruz |
| `x` | Cruzado |
| `D` | Diamante |
| `v` | Triângulo p/ baixo |
| `^` | Triângulo p/ cima |
| `s` | Quadrado |
| `*` | Estrela |

**Estilos de linha**: (-) (–) (:) (-.)

## Cores Básicas

| Código | Cor |
|---|---|
| `r` | Vermelho |
| `g` | Verde |
| `b` | Azul |
| `c` | ciano |
| `m` | magenta |
| `y` | Amarelo |
| `k` | Preto |
| `w` | Branco |
| `brown` | Marrom |
| `gray` | Cinza |
| `purple` | Roxo |

# Gráficos - Marcadores, Cores e Linhas

## Example

```
cores = ['r','b','g','m']
linha = ['-','--',':','-.']
x = np.linspace(-10,10,200)
for i in range(1,5):
    y = x**i*np.sin(x)
    y = y/y.max()
    plt.plot(x,y,c=cores[i-1],ls=linha[i-1],
    lw=2,label=r"$x^{}\sin(x)$".format(i))
plt.legend(frameon=False)
```

# Gráficos - Estilos

O Matplotlib tem o módulo **style** com vários tipos de formatação de gráficos que podem ser usados. Para listar alguns, faça:

```
>>> plt.style.available[:10]
['Solarize_Light2', '_classic_test_patch', 'bmh',
'classic', 'dark_background', 'fast',
'fivethirtyeight',
'ggplot', 'grayscale', 'seaborn']
```

Para usar, faça, por exemplo: `plt.style.use('bmh')`
Note que a escolha do estilo afetará **todos** os gráficos da sua seção.

# Gráficos - Dois Eixos

O comando `plt.twinx()` cria um novo eixo $y$ mantendo o mesmo eixo $x$ (também existe a opção plt.twiny()).

## Example

```
line1 = plt.plot(tempo,divorcios,'b-o')
plt.ylim(4,5.2)
plt.ylabel('Divorcios por 100 mil')
plt.xlabel('Anos')
plt.twinx()
line2 = plt.plot(tempo,margarina,'r-v')
plt.ylabel('Consumo de Margarina [lb]')
lines = line1 + line2
legendas = ['Divorcios','Consumo de Margarina']
plt.legend(lines, legendas, frameon=False)
```

# Gráfico Com Barras de Erro

Gráficos com barras de erro podem ser criados com a função `plt.errorbar`. Pode-se escolher várias opções de formataçã para a barra de erro.

## Example

```
x = np.linspace(0,10,30)
dy = 0.8
y = np.sin(x) + dy*np.random.random(30)
#plt.errorbar(x,y,yerr=dy,fmt='ko')
plt.errorbar(x,y,yerr=dy,ecolor='gray',elinewidth=2,
             capsize=3,fmt='ko')
```

# Gráfico de Barras

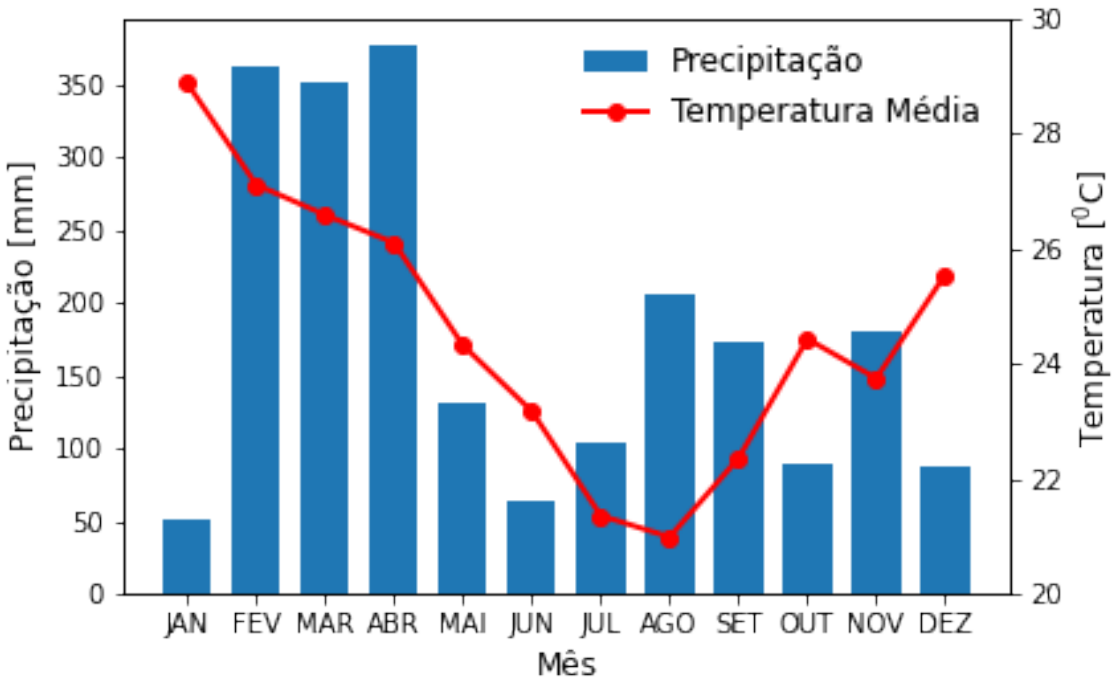
Gráficos de barras são feitos com a função `plt.bar`.

## Example

```
anos = np.arange(2014,2021)
consumo = np.array([327,392,490,643,806,981,1171])
plt.bar(anos, consumo, width=0.6,color='g',
        edgecolor=None)
plt.xlabel('Anos')
plt.ylabel('Consumo em milhares de sacas')
```

# Gráficos - Exemplo

**Exemplo**: Utilizando os dados da estação "Jardim Botânico", ano 2019, do site **Alerta Rio**, faça, numa mesma figura, um gráfico de barras para mostrar a precipitação total em cada mês, e um gráfico de linhas/pontos para mostrar a temperatura média, também em cada mês do ano. Utilize duas escalas verticais conforme o exemplo abaixo.



# Debugging

1. Leia seu código cuidadosamente.
2. Divida o seu código em pequenas partes, e assegure que cada parte faz exatamente o que você espera.
3. Teste seu código com casos conhecidos.
4. Imprima passos intermediários e os valores finais das variáveis.
5. Antecipe-se ao erro.
6. Explique o código linha por linha para outra pessoa ou para um objeto inanimado (*rubber duck debugging*).
7. Faça perguntas em fóruns online. (`https://stackoverflow.com/`)
8. Muitas vezes, debugging leva **mais tempo** do que esperamos.

# Exemplo de Aplicação - Modelo SIR

Podemos modelar epidemias através de um conjunto de equações diferenciais acopladas. Nessa modelagem, os indivíduos são separados em compartimentos: suscetível, infectado e recuperado (SIR).
Veja mais detalhes AQUI.

Verifique ESTE código em Python onde a solução numérica do conjunto de EDOs é implementada, e as solução são mostradas em diferentes gráficos.

# Para Saber Mais

**Livros**

- Spronck, Pieter. *The Coder's Apprentice*. Disponível online.
- Kinder, J.; Nelson, P. *A Student's Guide to Python for Physical Modeling*.
- Hill, Christian. *Learning Scientific Programming with Python*.
- VanderPlas, Jake. *Python Data Science Handbook*.

**Na web**

- Documentação do Python
- NumPy
- Matplotlib
- Tutorial