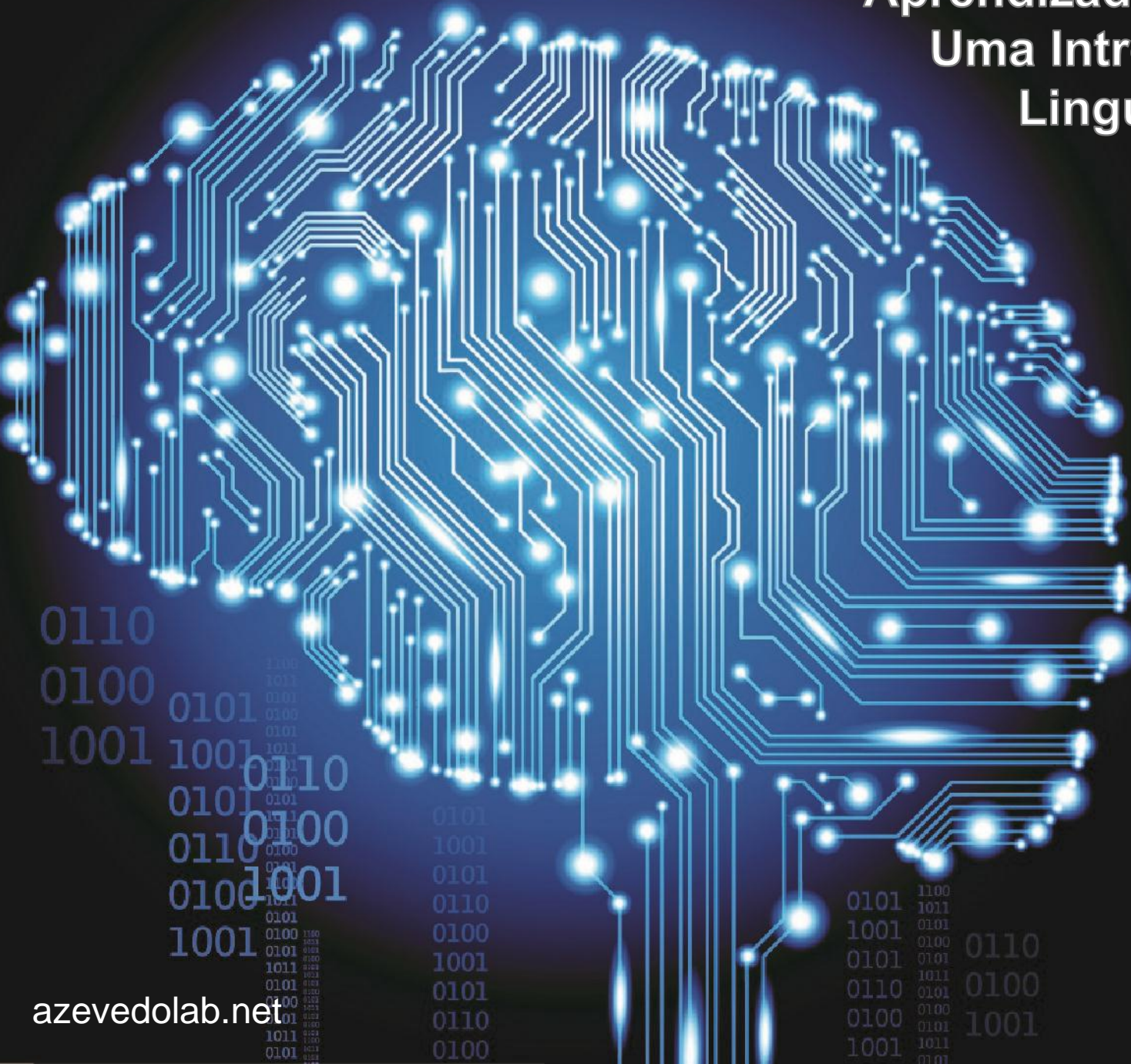
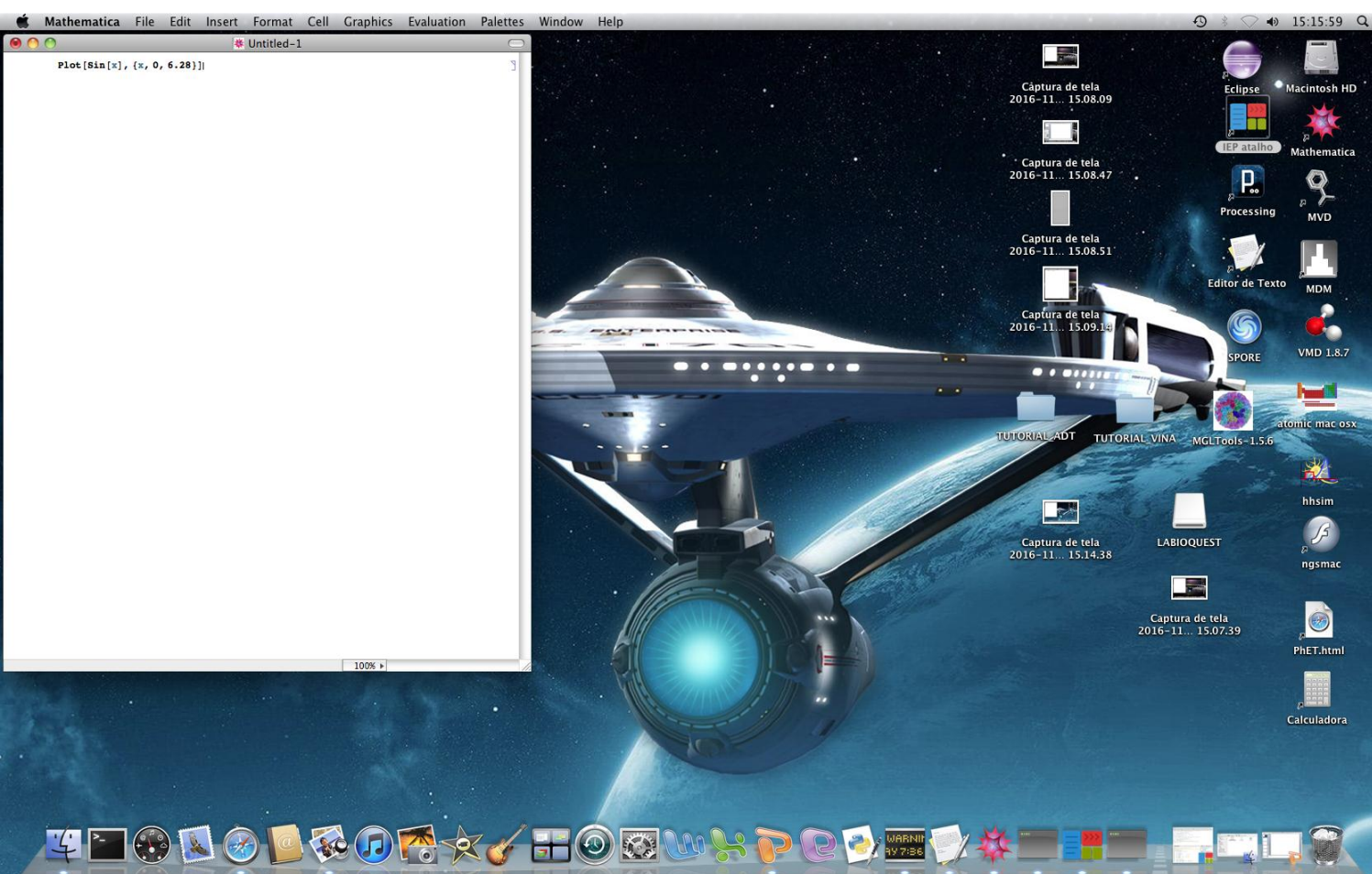


# Aprendizado de Máquina. Uma Introdução com a Linguagem Python

Aula 03

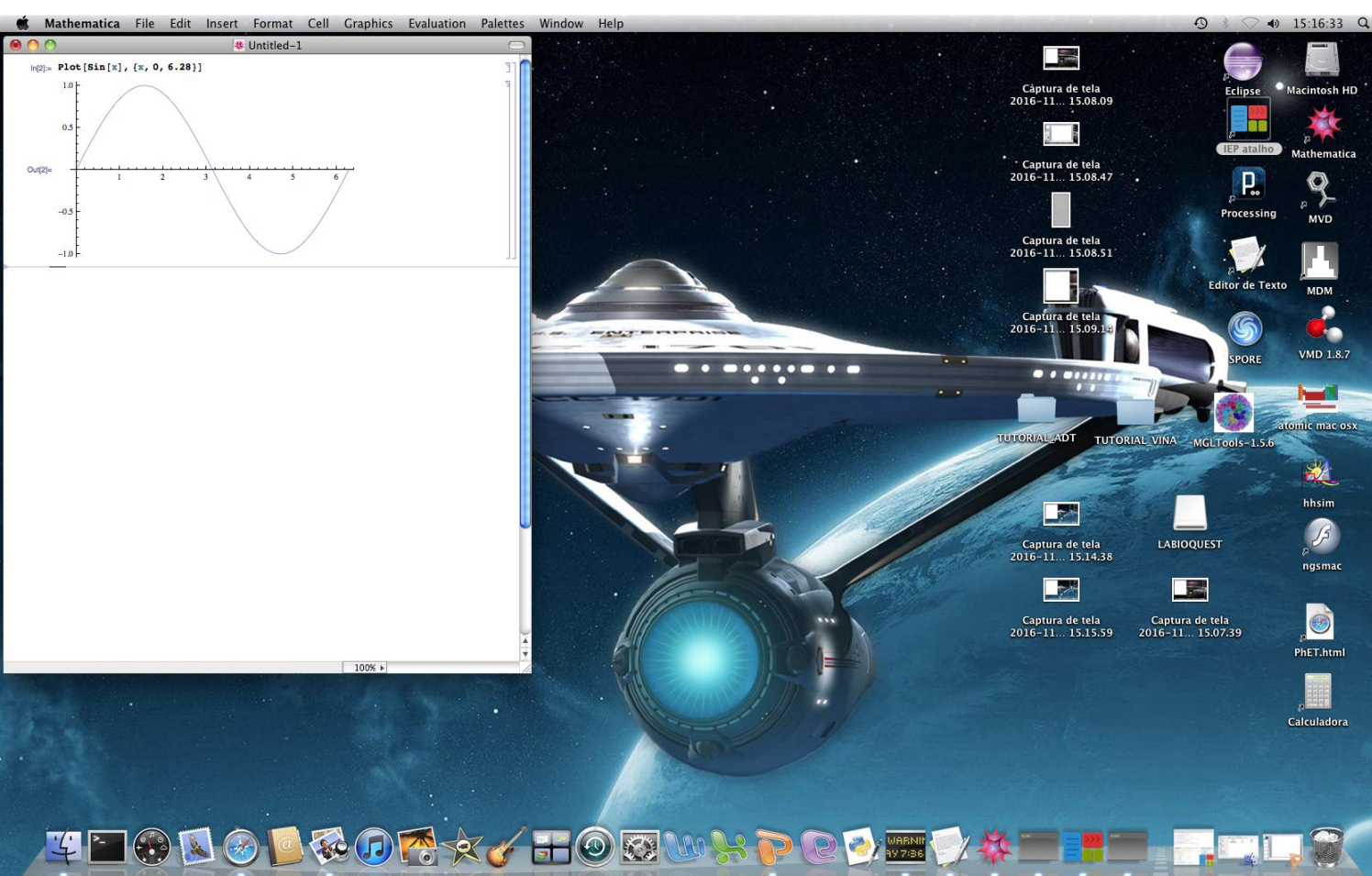


Há diversos programas pagos para computação científica, entre eles o *Mathematica*. O *Mathematica* tem um conjunto de funcionalidades, entre elas recursos para gerarmos gráficos de funções matemáticas. Por exemplo, para gerarmos o gráfico da função seno, digitamos `Plot[Sin[x], {x, 0, 6.28}]` e depois pressionamos `<shift> <Enter>`.

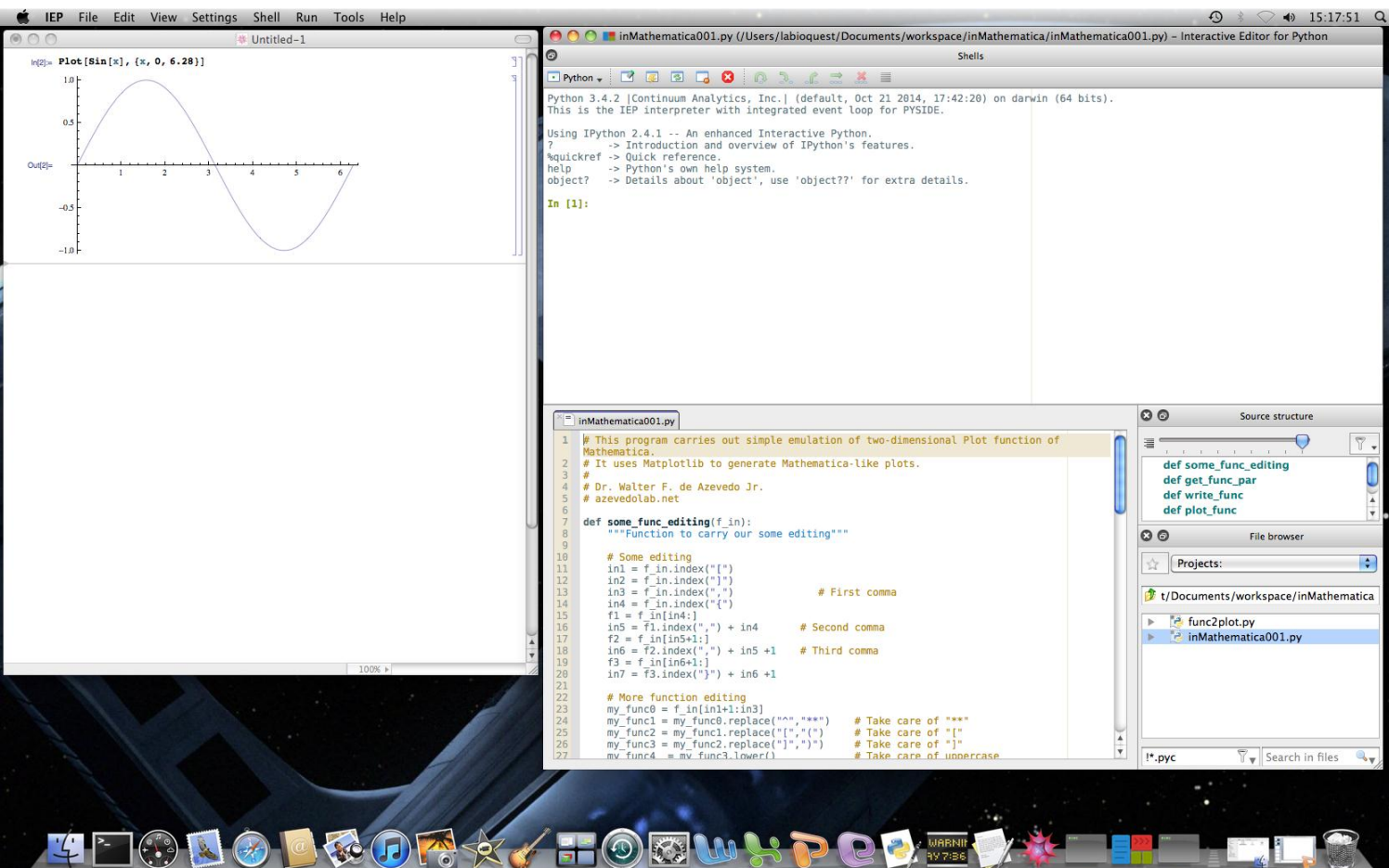




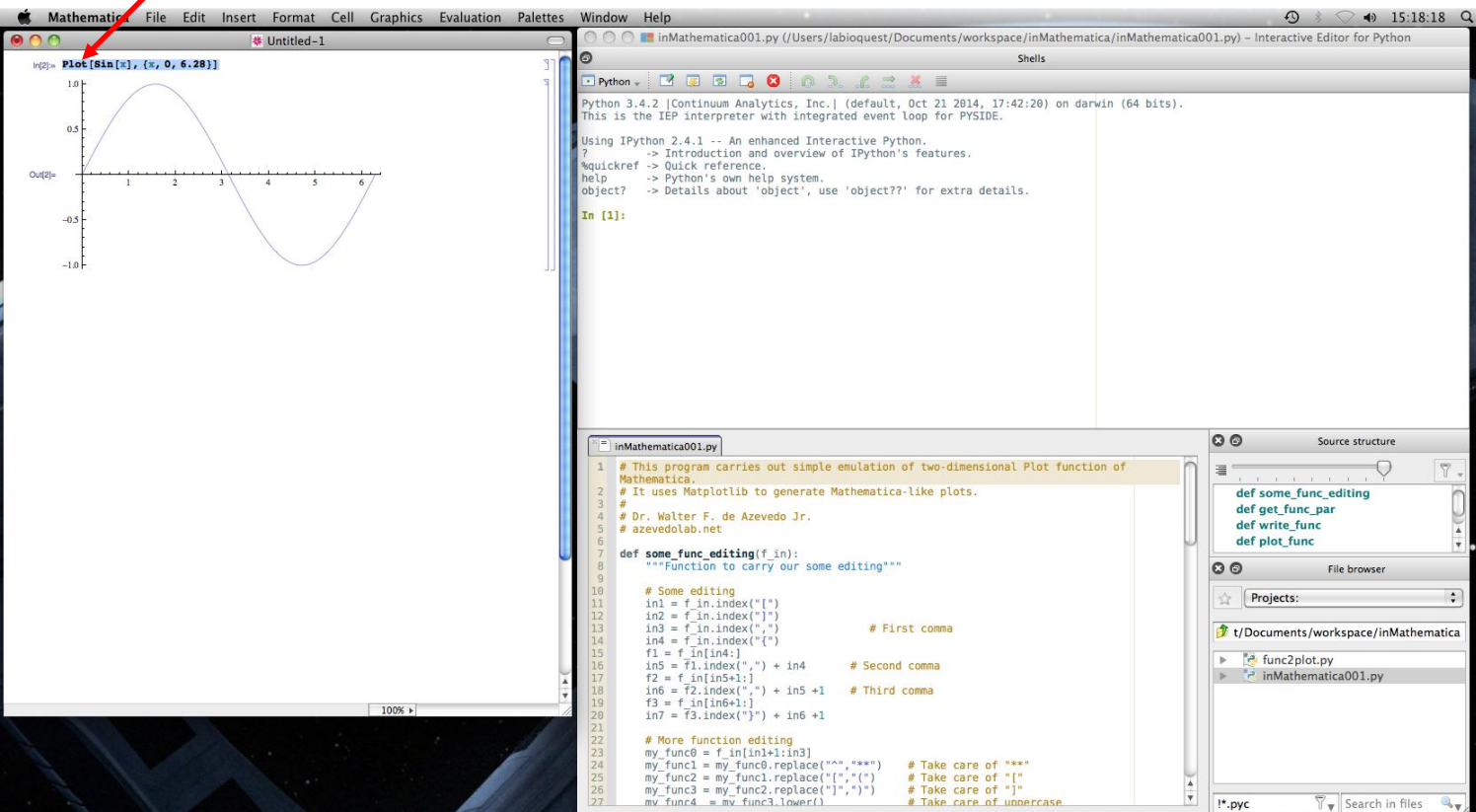
Pronto, temos o gráfico abaixo. O problema de tais programas é que são pagos. Há alternativas como o *Octave*, ou melhor ainda, o Python. O Python tem a biblioteca *Matplotlib* que possibilita que façamos os nossos próprios programas para gerarmos gráficos. Este é o assunto da aula de hoje.



O programa *inMathematica.py* é um emulador do programa *Mathematica* para gráficos. Este programa foi escrito em Python e faz uso da biblioteca *Matplotlib*. Ao executarmos seu código podemos digitar os comandos idênticos ao do *Mathematica* para gerarmos gráficos. Vamos ver um exemplo.



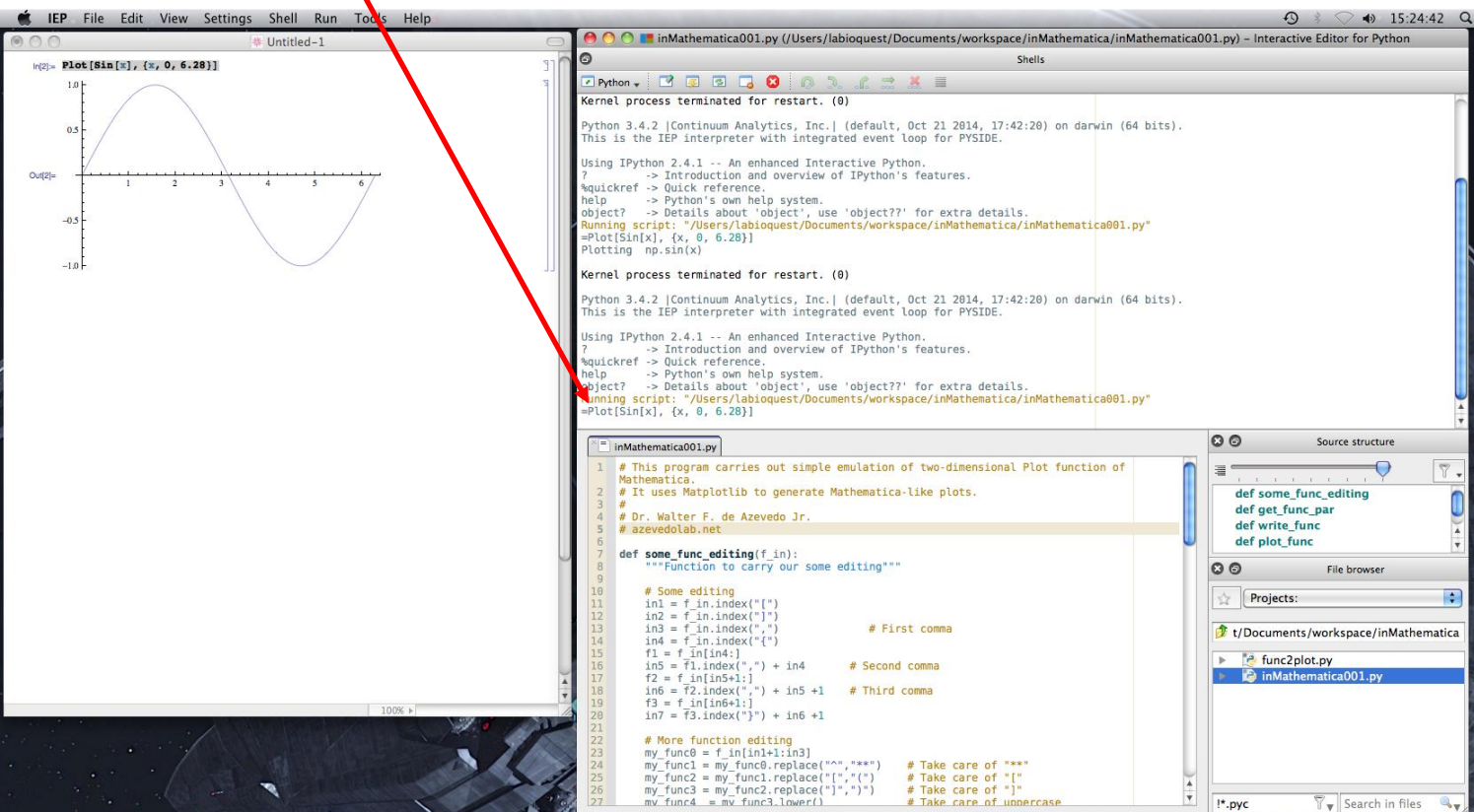
Marque a linha de comandos do *Mathematica*, como indicado abaixo. Pressione **<Cmd> c** (para Mac OS X) ou **<Ctrl> c** (para Windows e Linux).



Captura de tela  
2016-11-01 15:17:51



Faça *<Cmd> v* para o *shell* de comandos e pressione *<Enter>*.



```
Python 3.4.2 |Continuum Analytics, Inc.| (default, Oct 21 2014, 17:42:20) on darwin (64 bits).
This is the IEP interpreter with integrated event loop for PYSIDE.

Using IPython 2.4.1 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
Running script: "/Users/labioquest/Documents/workspace/inMathematica/inMathematica001.py"
=Plot[Sin[x], {x, 0, 6.28}]
Plotting np.sin(x)

Kernel process terminated for restart. (0)

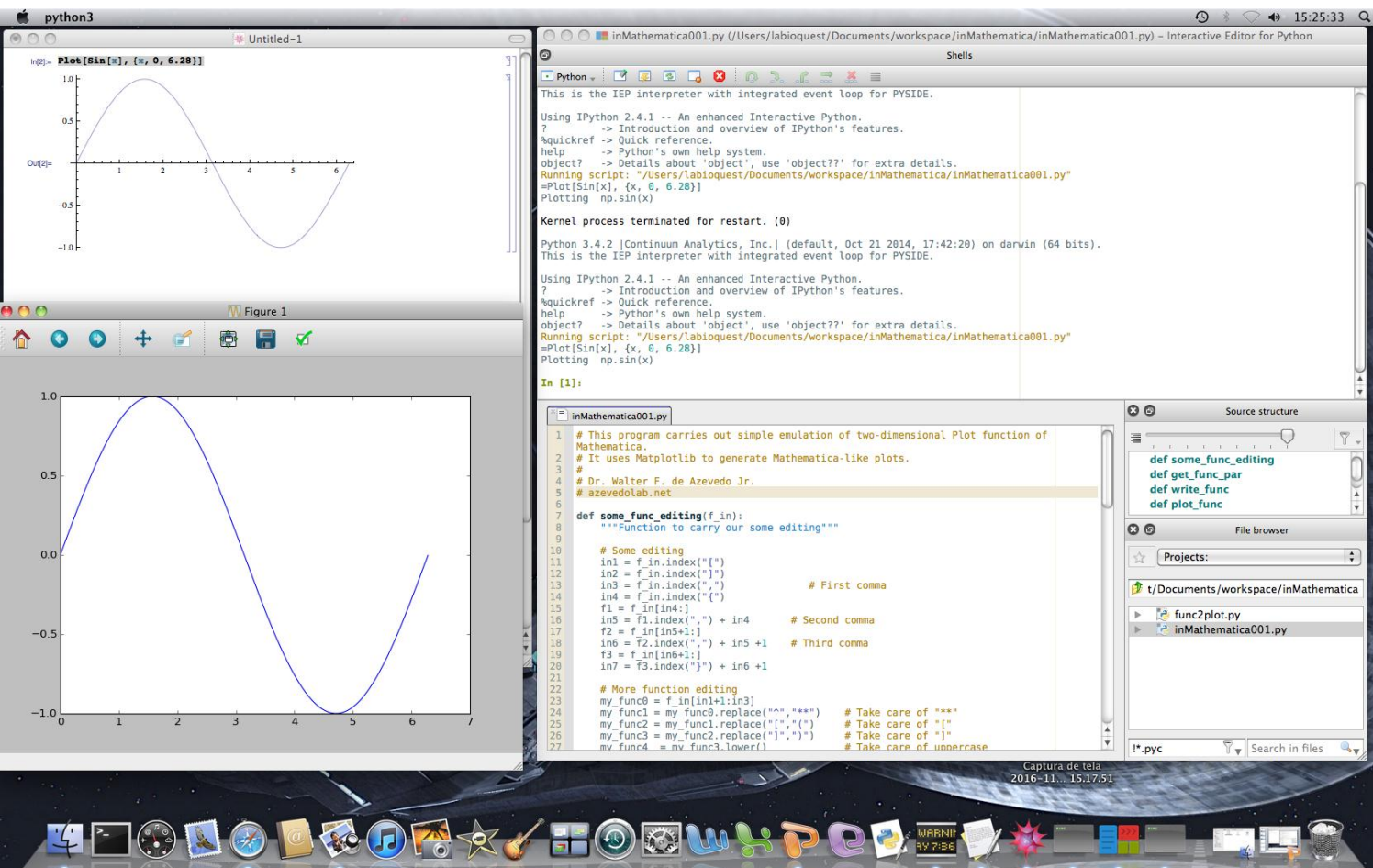
Python 3.4.2 |Continuum Analytics, Inc.| (default, Oct 21 2014, 17:42:20) on darwin (64 bits).
This is the IEP interpreter with integrated event loop for PYSIDE.

Using IPython 2.4.1 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
Running script: "/Users/labioquest/Documents/workspace/inMathematica/inMathematica001.py"
=Plot[Sin[x], {x, 0, 6.28}]
```

```
1 # This program carries out simple emulation of two-dimensional Plot function of
2 Mathematica.
3 # It uses Matplotlib to generate Mathematica-like plots.
4 # Dr. Walter F. de Azevedo Jr.
5 # azevedolab.net
6
7 def some_func_editing(f_in):
8     """Function to carry our some editing"""
9
10    # Some editing
11    in1 = f_in.index("[")
12    in2 = f_in.index(",")
13    in3 = f_in.index(",")
14    in4 = f_in.index("[")
15    f1 = f_in[in4:]
16    in5 = f1.index(",") + in4
17    f2 = f_in[in5:]
18    in6 = f2.index(",") + in5 + 1
19    f3 = f_in[in6:]
20    in7 = f3.index("]") + in6 + 1
21
22    # More function editing
23    my_func0 = f_in[in1+1:in3]
24    my_func1 = my_func0.replace(" ", "")
25    my_func2 = my_func1.replace("(", "")
26    my_func3 = my_func2.replace(")", "")
27    mv_func4 = mv_func3.lower()
```

Captura de tela  
2016-11-15 17:51

Pronto, nosso emulador de *Mathematica* gerou o gráfico da função seno com custo zero. Isto sem contar que você ainda pode dizer: “I did it!”



Podemos formatar a saída da função *print()*, para definirmos o número de casas de um número de ponto flutuante (*float*) com o símbolo %. A linha abaixo traz uma função *print()*, para mostrar a variável *x* com até oito casas, sendo três casas após o ponto decimal.

Indica até oito algarismos

Força o uso de três algarismos, após o ponto decimal

```
# Shows result
```

```
print("\n The value of  x = %8.3f"%x)
```

O símbolo % (fora das aspas mas dentro da função *print()*) indica que a variável à direita (*x*) seguirá a formatação indicada

O símbolo *f* (depois do % mas dentro da função *print()*) indica o uso de formatação de ponto flutuante

O símbolo %, dentro da função *print()*, indica o uso de formatação



Para números inteiros usamos *d*, como mostrado abaixo.

Indica até oito algarismos

```
# Shows result
```

```
print("\n The value of  x = %8d"%x)
```

O símbolo % (fora das aspas mas dentro da função *print()*) indica que a variável à direita (*x*) seguirá a formatação indicada

O símbolo *d* (depois do % mas dentro da função *print()*) indica o uso de formatação de inteiro

O símbolo % (dentro da função *print()*) indica o uso de formatação

Para strings usamos s, como mostrado abaixo.

Indica até oito caracteres

# Shows result

```
print("\n The value of  x = %8s"%x)
```

O símbolo % (fora das aspas mas dentro da função *print()*) indica que a variável à direita (x) seguirá a formatação indicada

O símbolo s (depois do % mas dentro da função *print()*) indica o uso de formatação de string

O símbolo % (dentro da função *print()*) indica o uso de formatação

Abaixo temos o programa *ex\_print.py* que mostra a aplicação do *print()* formatado. Veja que a variável *x* pode ser submetida ao *print()* como *float*, inteiro e string. O fato de separarmos 12 casas para o valor atribuído à variável *x*, indica que ao ser mostrado este valor será deslocado para direita.

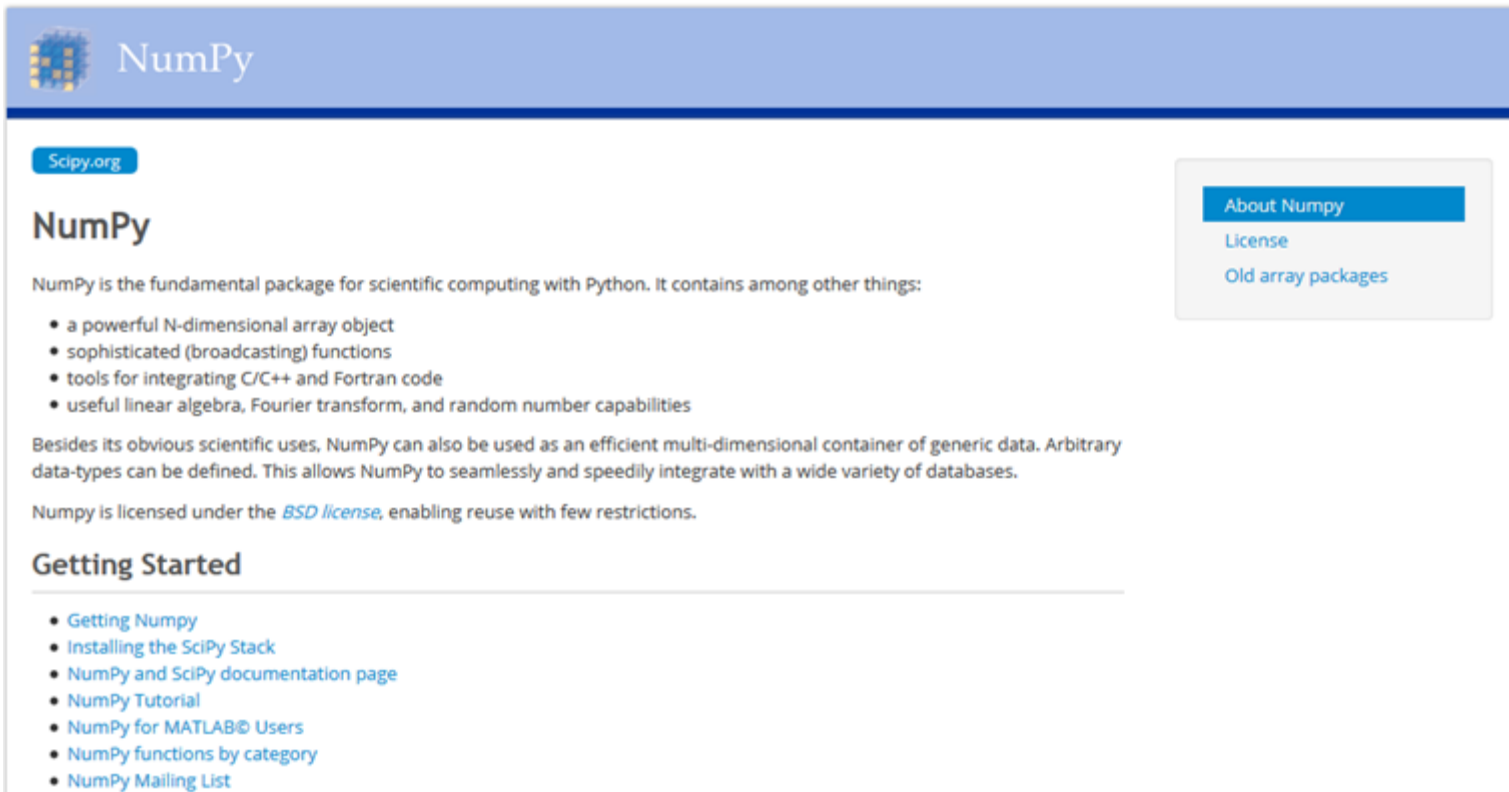
```
x = 8345.1359
print("\n The value of  x = %12.3f"%x)
print("\n The value of  x = %12d"%x)
print("\n The value of  x = %12s"%x)
x = "Python"
print("\n The value of  x = %12s"%x)
```

Abaixo temos o resultado da execução do código.

```
The value of  x =      8345.136
The value of  x =      8345
The value of  x =      8345.1359
The value of  x =      Python
```



Uma das bibliotecas mais usadas em computação científica em Python é a *NumPy*. Abaixo temos a página de entrada do projeto *NumPy*. Se você instalou o Python via pyzo, a biblioteca do *NumPy* já está instalada, caso contrário veja as instruções em [www.numpy.org](http://www.numpy.org), de como fazer a instalação.



The screenshot shows the NumPy website. At the top is a blue header with the NumPy logo and name. Below the header, on the left, is a blue button labeled 'Scipy.org'. The main content area has the title 'NumPy' followed by a paragraph describing it as the fundamental package for scientific computing with Python. It lists several features: a powerful N-dimensional array object, sophisticated (broadcasting) functions, tools for integrating C/C++ and Fortran code, and useful linear algebra, Fourier transform, and random number capabilities. Below this, it mentions that NumPy can also be used as an efficient multi-dimensional container of generic data and that it is licensed under the BSD license. On the right side, there is a sidebar with three links: 'About Numpy', 'License', and 'Old array packages'. At the bottom, there is a 'Getting Started' section with a list of links: 'Getting Numpy', 'Installing the SciPy Stack', 'NumPy and SciPy documentation page', 'NumPy Tutorial', 'NumPy for MATLAB® Users', 'NumPy functions by category', and 'NumPy Mailing List'.

NumPy

Scipy.org

## NumPy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

NumPy is licensed under the [BSD license](#), enabling reuse with few restrictions.

### Getting Started

- [Getting Numpy](#)
- [Installing the SciPy Stack](#)
- [NumPy and SciPy documentation page](#)
- [NumPy Tutorial](#)
- [NumPy for MATLAB® Users](#)
- [NumPy functions by category](#)
- [NumPy Mailing List](#)

About Numpy  
License  
Old array packages

Uma das principais vantagens do uso da biblioteca *NumPy*, é que dispomos de um arsenal de funções para realizar cálculos mais rápidos do que se fossem implementados em Python puro. Vamos ilustrar com um exemplo disponível em [http://www-personal.umich.edu/~kundeng/stats607/week\\_5\\_pysci-03-scipy.pdf](http://www-personal.umich.edu/~kundeng/stats607/week_5_pysci-03-scipy.pdf). Abaixo temos o código em Python puro, para o cálculo da multiplicação de todos elementos de duas listas, *a* e *b*. Cada lista tem 1.000.000 elementos. A multiplicação ocorre dentro de um loop *for*. Não usamos a *NumPy*, todos os comandos são do Python, com exceção da biblioteca *time*, usada para termos o tempo em que inicia e termina o processo e, assim, termos o tempo total de execução do programa *pure\_python.py*.

```
import time                # Imports library time
l = 10000000               # Assigns 10000000 to variable l
start = time.time()        # Assigns current time to variable start
a, b = range(l), range(l)  # Assigns a list with integer numbers from 0 to l to variables a and b
c = []                     # Assigns an empty list to variable c
for i in a:                 # Looping through all elements
    c.append(a[i] * b[i])   # Assigns the multiplication of each element of a and b to c
t = time.time() - start    # Assigns the duration to the variable t
print("Duration: %s" % t)   # Shows duration on screen
```

A linha `start = time.time()` obtém a hora local do computador. O loop `for` repete a multiplicação 1 milhão de vezes e atribui cada valor a um elemento da lista `c`. A linha `t = time.time() - start` subtrai o tempo inicial do tempo atual, após finalizado o loop `for`. Por último o resultado é mostrado na tela.

```
import time                # Imports library time
l = 10000000               # Assigns 1000000 to variable l
start = time.time()        # Assigns current time to variable start
a, b = range(l), range(l) # Assigns a list with integer numbers from 0 to l to variables a and b
c = []                     # Assigns an empty list to variable c
for i in a:                # Looping through all elements
    c.append(a[i] * b[i])  # Assigns the multiplication of each element of a and b to c
t = time.time() - start    # Assigns the duration to the variable t
print("Duration: %s" % t)  # Shows duration on screen
```

Ao executarmos o código, temos o resultado abaixo, que pode variar de computador para computador, e, mesmo em computadores idênticos temos variações, devido à disponibilidade da CPU, no momento da execução do código.

```
Duration: 8.81787896156311
```



No programa *numpy\_ex1.py*, o processamento da informação do tempo é idêntico ao programa *pure\_python.py*. A diferença está no uso da biblioteca *NumPy*. Às variáveis *a* e *b*, atribuímos *arrays* com 1 milhão de elementos e não precisamos de um loop *for* para a multiplicação de todos elementos, basta usarmos  $c = a * b$ , e pronto.

```
import numpy as np          # Imports library NumPy
import time                 # Imports library time
l = 10000000                # Assigns 10000000 to variable l
start = time.time()         # Assigns current time to variable start
a = np.arange(l)            # Assigns an array with integer numbers from 0 to l to variable a
b = np.arange(l)            # Assigns an array with integer numbers from 0 to l to variable b
c = a * b                   # Assigns the multiplication of each element of a and b to c
t = time.time() - start     # Assigns the duration to the variable t
print("Duration: %s" % t)    # Shows duration on screen
```

Ao executarmos o código, vemos que a diferença de execução é imensa, por isso recomenda-se o uso de *arrays* da *NumPy*, toda vez que tivermos dados na forma de colunas numa tabela ou, numa linguagem mais técnica, toda vez que tivermos vetores.

```
Duration: 0.06304192543029785
```

A biblioteca *NumPy* tem diversas formas de definirmos *arrays*.  
Vejam os seguintes exemplos.

A função `array()` cria um *array* a partir de uma lista. Vejamos um exemplo, o programa `numpy_ex2.py`, mostrado abaixo. Criamos uma lista com os 10 primeiros elementos da série de Fibonacci e atribuímos à variável `my_list`. Em seguida, usamos a função `np.array()`, para criarmos um *array* a partir da lista `my_list`. Vejam que, tanto a lista quanto o *array*, apresentam seus elementos indexados, e podem ser acessados individualmente, como no loop `for` que vem em seguida.

```
import numpy as np                # Imports library NumPy
my_list = [1,1,2,3,5,8,13,21,34,55] # Assigns 10 elements of Fibonacci series to the list my_list
my_array = np.array(my_list)      # Creates an array from a list and assigns it to my_array
for i in range(len(my_array)):    # Loop for to show elements of array and list
    print(my_list[i],my_array[i])
```

Ao executarmos o `numpy_ex2.py`, temos o resultado a seguir.

```
1 1
1 1
2 2
3 3
5 5
8 8
13 13
21 21
34 34
55 55
```

Veja que não há diferença na forma de acessarmos os elementos da lista e do *array*. A vantagem do uso de *arrays*, está na possibilidade de uso das funções da biblioteca *NumPy*. Outra diferença que temos que destacar, os *arrays* tratam sempre de elementos numéricos e de um só tipo, não podemos misturar inteiros com *floats*. Nas listas podemos misturar.

A função *linspace(start, stop, num)* cria um *array* que inicia em *start* e termina em *stop*, com um total de *num* números igualmente espaçados. Vejamos o código *numpy\_ex3.py*. A linha de código *my\_array = np.linspace(5,10,11)* indica a criação de um *array* com 11 elementos, que começa em 5 e termina em 10, veja que os extremos estão incluídos.

```
import numpy as np                # Imports library NumPy
# Creates an array that starts at 5, finishes at 10 and has 11 elements equally spaced
my_array = np.linspace(5,10,11)
for i in range(len(my_array)):    # Loop for to show elements of array
    print(my_array[i])
```

Ao executarmos o código, temos o resultado abaixo.

```
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5
9.0
9.5
10.0
```

A execução gera 11 elementos, que são mostrados na tela, como os extremos estão incluídos, são inseridos 9 elementos entre 5 e 10. A condição de igual espaçamento entre os números, impõe que teremos um intervalo de 0.5 entre os números do *array*.



A função *arange(start, stop, step, dtype=float)* cria um *array* que inicia em *start*, finaliza em *stop* (exclusivo) e tem um passo *step*. Podemos selecionar o tipo dos elementos do *array*, com *dtype = float* por exemplo. O código *numpy\_ex4.py* cria um *array* que inicia em 5.0 e termina em 10.0 (exclusivo), ou seja, o 10.0 não está incluído. Não é necessário explicitar a palavra chave *dtype*, basta que indiquemos o tipo como argumento da função *arange()*, como indicado na linha vermelha abaixo.

```
import numpy as np                                # Imports library NumPy
# Creates an array that starts at 5, finishes at 10 (exclusive) and has a step of 0.5
my_array = np.arange(5,10,0.5,float)
for i in range(len(my_array)):                    # Loop for to show elements of array
    print(my_array[i])
```

A execução de *numpy\_ex4.py* gera o resultado abaixo.

```
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5
9.0
9.5
```

Foram gerados 10 elementos, que são mostrados na tela, como o extremo superior (10.0) não está incluído, o último elemento do *array* é 9.5.

Podemos gerar um *array* só com zeros, usando-se a *função zeros(num, dtype=float)*, que retorna um *array* com *num* zeros. Podemos especificar o tipo com *dtype*. Abaixo temos o código *gen\_zeros.py*. A linha de código *my\_array = np.zeros(10, dtype=int)* gera um *array* com 10 elementos, todos zeros e inteiros. Como na função *arange()*, não precisamos explicitar o *dtype*.

```
import numpy as np                # Imports library NumPy
# Creates an array with 10 zeros of integer type
my_array = np.zeros(10, dtype=int)
for i in range(len(my_array)):    # Loop for to show elements of array
    print(my_array[i])
```

Ao executarmos o código, temos o resultado abaixo.

A execução de *gen\_zeros.py* gera 10 zeros, todos inteiros. O default da função *zeros()* é *dtype = float*.

Temos uma função da biblioteca *NumPy* para gerar um *array* de números “1”. Sem surpresas, a função é *ones(num, dtype=float)*, que retorna um *array* com *num* números “1”. As regras da função *zeros()* são mantidas para a função *ones()*.

```
import numpy as np                # Imports library NumPy
# Creates an array with 10 ones of float type
my_array = np.ones(10)
for i in range(len(my_array)):    # Loop for to show elements of array
    print(my_array[i])
```

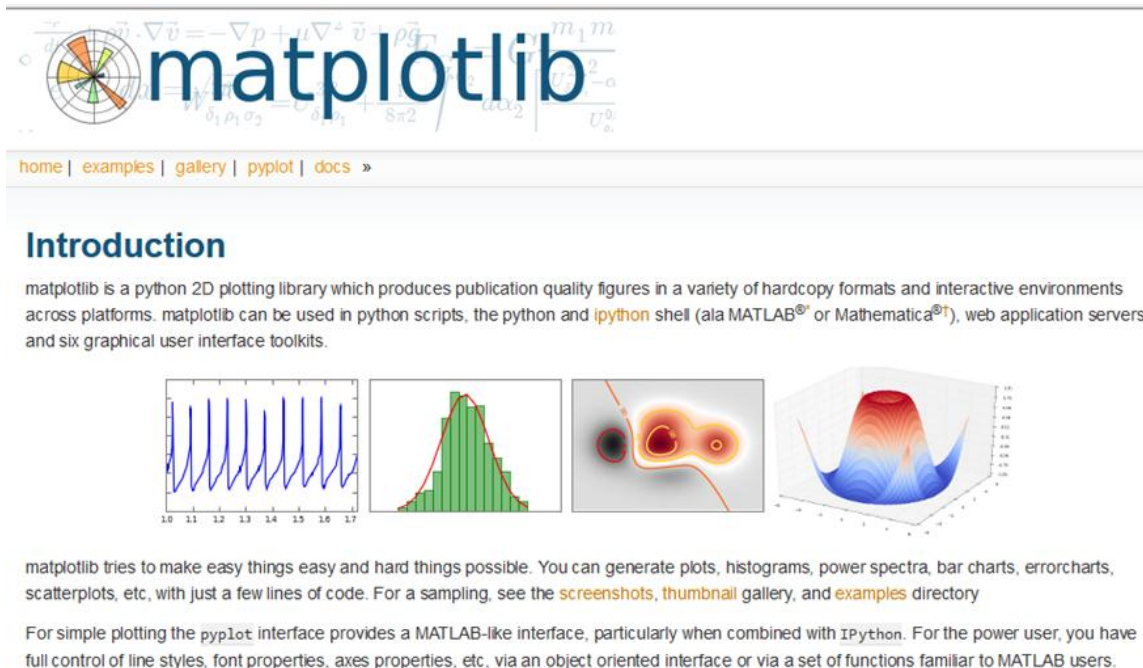
Abaixo temos o *array* de “1.0” gerados e mostrados na tela.

```
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
```

A execução de *gen\_ones.py* gera 10 números “1”, todos do tipo *float*. Como para função *zeros()*, o default da função *ones()* é *dtype = float*.



Uma das características da linguagem de programação Python, que tem atraído um grande número de fãs na comunidade de computação científica, incluindo aprendizado de máquina, é a possibilidade de uso de bibliotecas gratuitas para gráficos e análise numérica. Nas próximas aulas, introduziremos as bibliotecas *Matplotlib*, *NumPy*, *SciPy* e *Scikit-learn*.

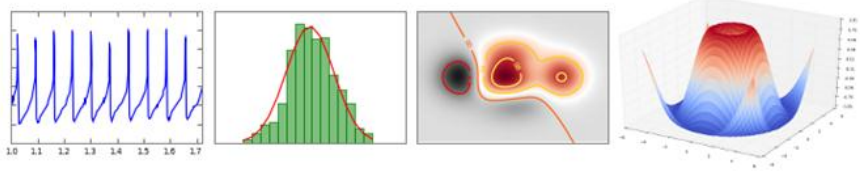


The screenshot shows the Matplotlib website homepage. At the top, there is a navigation bar with links: [home](#) | [examples](#) | [gallery](#) | [pyplot](#) | [docs](#) ». Below this is a section titled "Introduction". The text describes Matplotlib as a Python 2D plotting library that produces publication-quality figures in various formats and interactive environments. It mentions that Matplotlib can be used in Python scripts, the Python and IPython shells (ala MATLAB® or Mathematica®), web application servers, and six graphical user interface toolkits. Below the text, there are four small plots: a line plot with multiple overlapping blue lines, a histogram with green bars and a red normal distribution curve, a 2D contour plot with red and yellow regions, and a 3D surface plot with a blue and red surface. At the bottom, there is a paragraph explaining that Matplotlib tries to make easy things easy and hard things possible, and that it provides a MATLAB-like interface via the pyplot module and a more powerful object-oriented interface for advanced users.

home | examples | gallery | pyplot | docs »

## Introduction

matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and [ipython](#) shell (ala MATLAB® or Mathematica®), web application servers, and six graphical user interface toolkits.



matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code. For a sampling, see the [screenshots](#), [thumbnail gallery](#), and [examples](#) directory

For simple plotting the [pyplot](#) interface provides a MATLAB-like interface, particularly when combined with [IPython](#). For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

Página de entrada do matplotlib: <http://matplotlib.org/>

Acesso em: 24 de novembro de 2016.

Vejam os usos das bibliotecas *Matplotlib* e *NumPy*, para gerarmos o gráfico do seno. A primeira linha de código importa a biblioteca *matplotlib* para gráficos, como *plt*. Assim, todos os recursos desta biblioteca podem ser chamados a partir da notação *dot* (*.*). O mesmo acontece na segunda linha de código, com a biblioteca *numpy*, importada como *np*. O método *.linspace()*, da biblioteca *numpy*, gera um conjunto de números *floats*, entre os intervalos mostrados, no caso, entre 0 e  $2\pi$ . O valor de  $\pi$  pode ser chamado com *numpy*, por meio de *np.pi*. Dentro dos parênteses do método *linspace()*, o último número indica o número de pontos gerados, entre os valores máximo e mínimo.

```
import matplotlib.pyplot as plt
import numpy as np
# Defines x range with linspace
x = np.linspace(0, 2*np.pi, 100)
# Defines sine function
sine_func = np.sin(x)
# Creates plot
plt.plot(x, sine_func)
# Shows plot
plt.show()
# Saves plot on png file
plt.savefig('sine1.png')
```

Os números gerados com o método *np.linspace(0,2\*np.pi,100)* são atribuídos à variável *x*. Veja que o último número (100) indica o número de *floats* gerados no intervalo estabelecido, pelos dois primeiros números, dentro dos parênteses.

A linha de comando *sine\_func = np.sin(x)*, define a função seno, a partir da biblioteca *numpy*. O resultado é atribuído à variável *sine\_func*. O método *plt.plot(x,sine\_func)* gera o gráfico cartesiano, os valores do eixo *x* são aqueles atribuídos à variável *x*, e os valores do eixo *y*, são aqueles atribuídos à variável *sine\_func*.

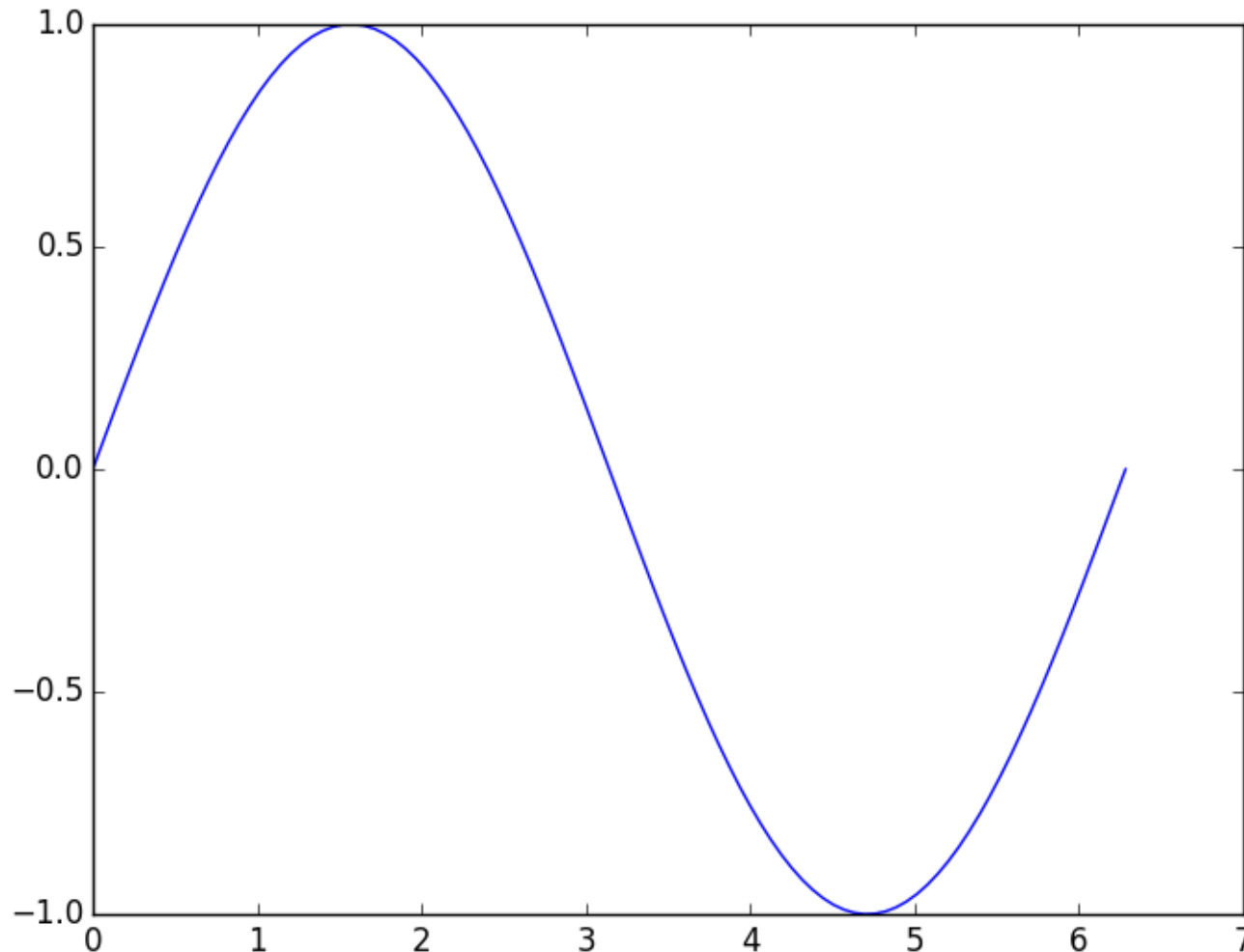
```
import matplotlib.pyplot as plt
import numpy as np
# Defines x range with linspace
x = np.linspace(0, 2*np.pi, 100)
# Defines sine function
sine_func = np.sin(x)
# Creates plot
plt.plot(x, sine_func)
# Shows plot
plt.show()
# Saves plot on png file
plt.savefig('sine1.png')
```

O método *plt.show()* mostra o gráfico na tela. Este método pode ter resultados diferentes, conforme a instalação da biblioteca *Matplotlib* e o sistema operacional que você está utilizando. O método *plt.savefig('sine1.png')* salva o gráfico gerado no arquivo *sine1.png*. Execute o código *sine1.py* e clique na pasta da aula de hoje. Localize o arquivo *sine1.png* e clique duas vezes neste. Veja o gráfico gerado.

```
import matplotlib.pyplot as plt
import numpy as np
# Defines x range with linspace
x = np.linspace(0, 2*np.pi, 100)
# Defines sine function
sine_func = np.sin(x)
# Creates plot
plt.plot(x, sine_func)
# Shows plot
plt.show()
# Saves plot on png file
plt.savefig('sine1.png')
```



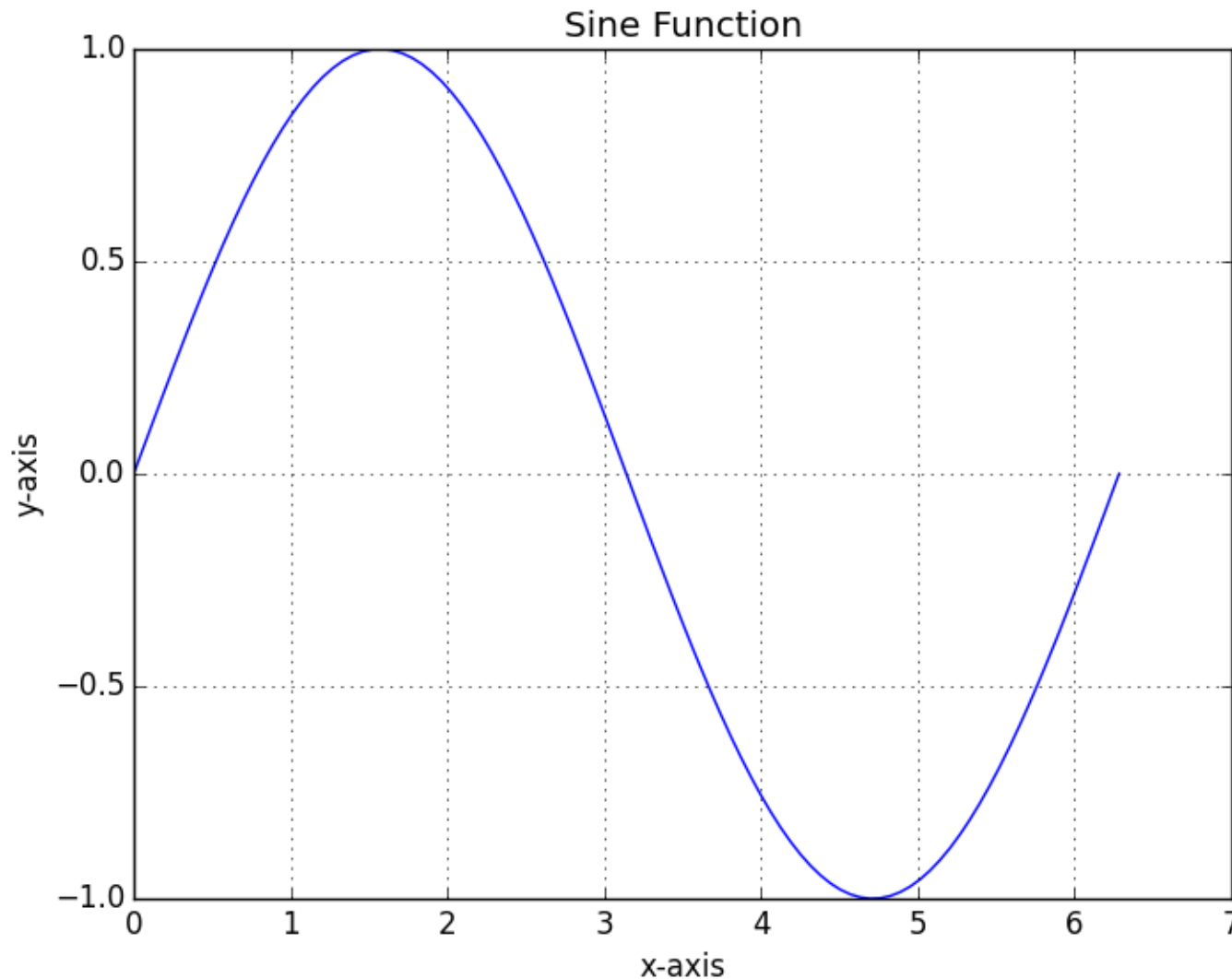
Abaixo temos o gráfico da função seno, gerado pelo programa *sine1.py*.



Como em toda biblioteca gráfica, podemos adicionar os nomes dos eixos, gradeado e título, com os métodos destacados em vermelho no código *sine2.py*. Os métodos *plt.xlabel()*, *plt.ylabel()* e *plt.title()* aceitam variáveis como argumentos, ou strings, como no código abaixo.

```
import matplotlib.pyplot as plt
import numpy as np
# Defines x range with linspace
x = np.linspace(0, 2*np.pi, 100)
# Defines sine function
sine_func = np.sin(x)
# Creates plot
plt.plot(x, sine_func)
plt.xlabel('x-axis')           # Adds axis label
plt.ylabel('y-axis')           # Adds axis label
plt.title('Sine Function')     # Adds title
plt.grid(True)                 # Adds grid to the plot
# Shows plot
plt.show()
# Saves plot on png file
plt.savefig('sine2.png')
```

Abaixo temos o gráfico da função seno, gerado pelo programa *sine2.py*.



O programa *trig1.py* gera o gráfico para as funções seno e cosseno. Para isto basta adicionarmos a variável *cos\_func*, e atribuímos a ela o resultado do método *np.cos(x)*.

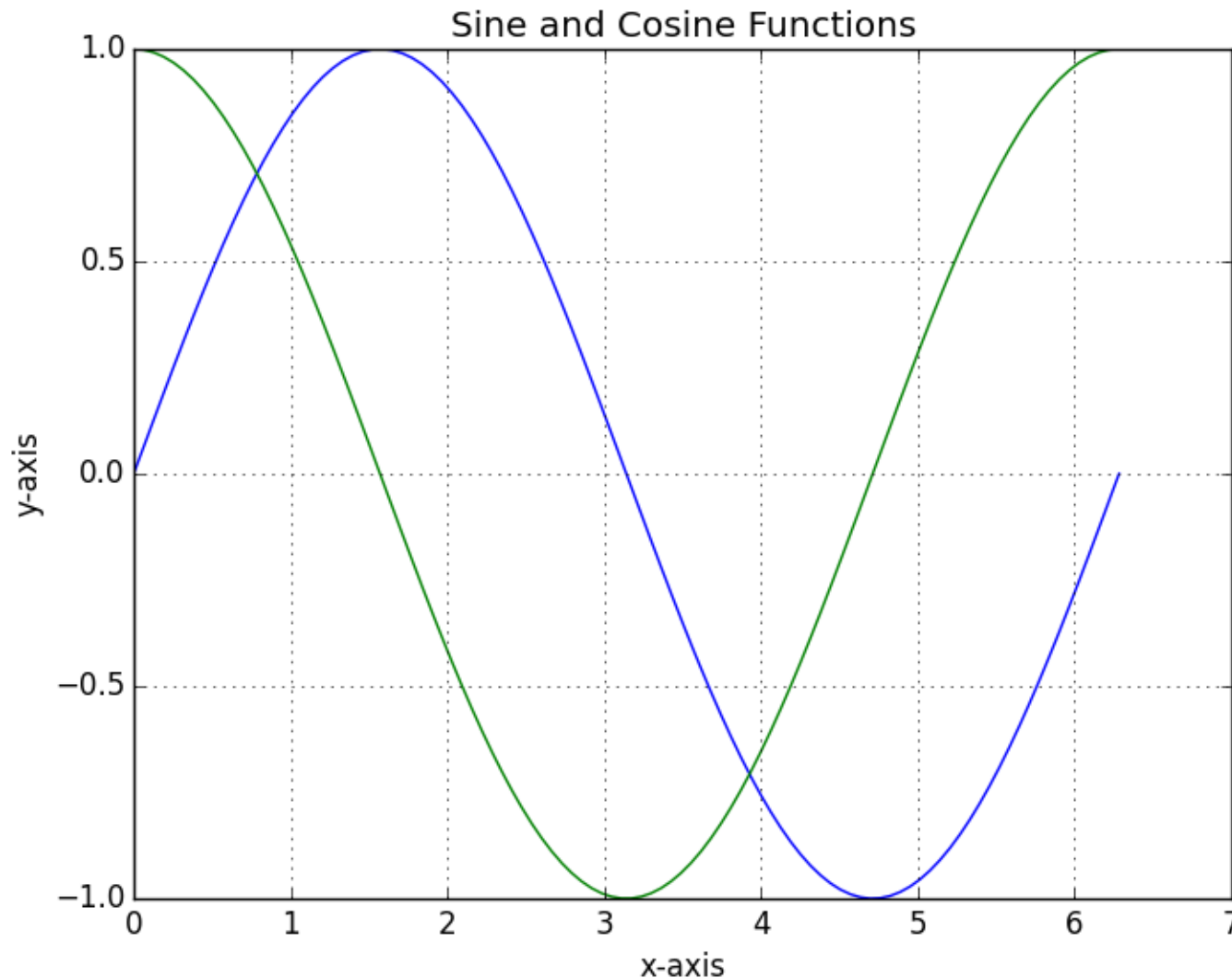
```
import matplotlib.pyplot as plt
import numpy as np
# Defines x range with linspace
x = np.linspace(0, 2*np.pi, 100)
# Defines sine and cosine functions
sine_func = np.sin(x)
cos_func = np.cos(x)
# Creates plots
plt.plot(x, sine_func)
plt.plot(x, cos_func)
plt.xlabel('x-axis')          # Adds axis label
plt.ylabel('y-axis')          # Adds axis label
plt.title('Sine and Cosine Functions') # Adds title
plt.grid(True)                 # Adds grid to the plot
# Shows plot
plt.show()
# Saves plot on png file
plt.savefig('trig1.png')
```

Temos que adicionar o método *plt.plot(x,cos\_func)* para o gráfico do cosseno, como indicado abaixo. Mudamos também o título do gráfico.

```
import matplotlib.pyplot as plt
import numpy as np
# Defines x range with linspace
x = np.linspace(0, 2*np.pi, 100)
# Defines sine and cosine functions
sine_func = np.sin(x)
cos_func = np.cos(x)
# Creates plots
plt.plot(x, sine_func)
plt.plot(x, cos_func)
plt.xlabel('x-axis')          # Adds axis label
plt.ylabel('y-axis')          # Adds axis label
plt.title('Sine and Cosine Functions') # Adds title
plt.grid(True)                # Adds grid to the plot
# Shows plot
plt.show()
# Saves plot on png file
plt.savefig('trig1.png')
```



Abaixo temos o gráfico das funções seno cosseno, geradas pelo programa *trig1.py*. Veja que linha do cosseno é verde.



# Gráfico das funções seno e cosseno

Programa: *trig2.py*

## Resumo

Programa para gerar os gráficos da função seno e cosseno, a partir das bibliotecas *Matplotlib* e *NumPy*. O usuário digita a faixa de valores para o eixo x e os títulos dos eixos x,y, bem como o título do gráfico e o nome do arquivo de saída. Ambas funções são colocadas no mesmo gráfico. O programa gera o arquivo de saída no formato PNG.

A principal novidade do código *trig2.py*, é que as informações, para gerarmos o gráfico, são lidas a partir de entradas digitadas pelo usuário. Para isto precisamos da função *input()*, como indicado no trecho em vermelho abaixo. A leitura dos valores mínimo e máximo do eixo *x*, são convertidos para *float*. As informações são atribuídas às variáveis *x\_min* e *x\_max*. As outras informações são strings, e não devem ser convertidas.

```
# Imports libraries
import matplotlib.pyplot as plt
import numpy as np

# Reads input information
x_min = float(input("Type minimum value for x-axis => "))
x_max = float(input("Type maximum value for x-axis => "))
x_label = input("Type x-axis label => ")
y_label = input("Type y-axis label => ")
plot_title = input("Type plot title => ")
output_file = input("Type output file name => ")
```

Os valores atribuídos às variáveis de entrada, são usados como argumentos para a chamada de funções para gerarmos o gráfico. As linhas de código, que usam os valores atribuídos às variáveis, estão indicadas em vermelho. A função *plt.grid(True)* não precisa de variável, ele tem como argumento o valor lógico *True*, que habilita o desenho de grades no gráfico.

```
# Defines x range with np.linspace()
x = np.linspace(x_min, x_max, 100)

# Defines sine and cosine functions
sine_func = np.sin(x)
cos_func = np.cos(x)

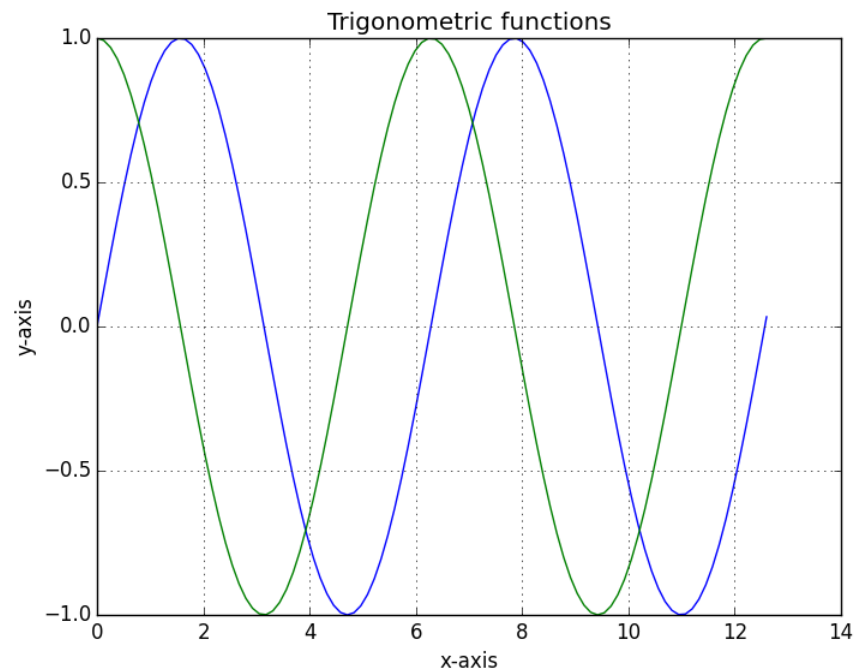
# Creates plots
plt.plot(x, sine_func)
plt.plot(x, cos_func)
plt.xlabel(x_label)      # Adds axis label
plt.ylabel(y_label)     # Adds axis label
plt.title(plot_title)   # Adds title
plt.grid(True)           # Adds grid to the plot

# Shows plot
plt.show()

# Saves plot
plt.savefig(output_file)
```

Abaixo temos os dados de entrada e o gráfico gerado.

```
Type minimum value for x-axis => 0
Type maximum value for x-axis => 12.6
Type x-axis label => x-axis
Type y-axis label => y-axis
Type plot title => Trigonometric functions
Type output file name => trig2.png
```





Já vimos alguns programas para gráficos de funções trigonométricas. Veremos agora, com mais detalhes, diversos programas em Python para gerar gráficos, a partir de recursos da biblioteca *Matplotlib*. Abaixo temos um código simples, para exibição de pontos num gráfico bidimensional. As linhas de código estão em vermelho, os comentários em preto. A primeira linha importa a biblioteca *Matplotlib* como *plt*. Depois temos a linha com o *.plot()*, onde foram especificados 8 valores, que representam coordenadas no eixo y. A biblioteca *Matplotlib* usa valores implícitos para o eixo x, começando em zero até o valor inteiro N-1, onde N é o número de itens na lista. A função *.show()* mostra o gráfico na tela. Por último, a função *.savefig()*, salva o arquivo com o gráfico gerado. Veja, com 4 linhas de código, geramos o gráfico.

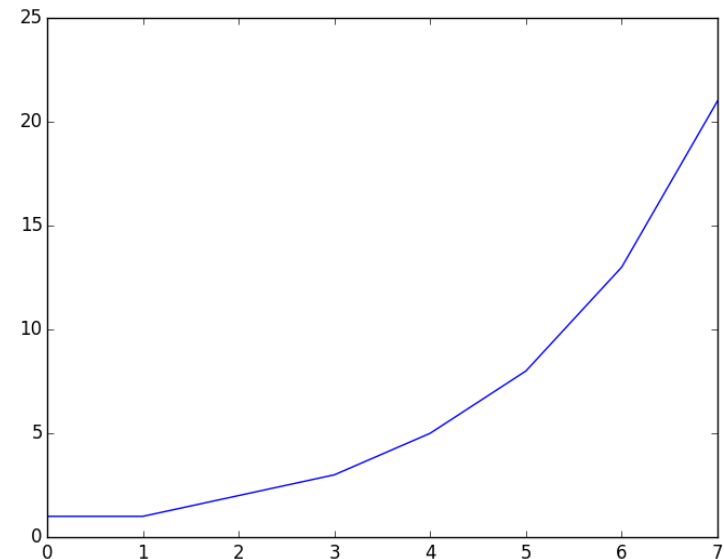
```
import matplotlib.pyplot as plt

# Generates a simple plot
plt.plot([1,1,2,3,5,8,13,21])

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("simple_plot1.png")
```

Abaixo temos o gráfico gerado.



```
import matplotlib.pyplot as plt

# Generates a simple plot
plt.plot([1,1,2,3,5,8,13,21])

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("simple_plot1.png")
```

Já vimos em Python a função *range(i,j,k)*, que gera uma lista de inteiros, iniciando em *i* e finalizando em *j*, com um passo *k*. Usaremos uma função da biblioteca *NumPy*, chamada *np.arange(x,y,z)*, que gera uma sequência de números, não necessariamente inteiros. Esta sequência inicia em *x*, termina em *y-z* e tem um passo de *z*. Vejamos o código do programa *simple\_plot2.py*, que gera uma parábola entre zero e 7.99. As linhas de código estão em vermelho e os comentários em preto.

```
# Program to illustrate the use of Matplotliblib
import matplotlib.pyplot as plt
import numpy as np

# Generates x-axis
x = np.arange(0,8,0.01)

# Generates y-axis
y = x**2

# Generates plot
plt.plot(x,y)

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("simple_plot2.png")
```

Inicialmente importamos as bibliotecas *Matplotlib* e *NumPy*. Em seguida chamamos a função *np.arange()* que gera uma sequência de números, entre 0 e 7.99, com passo 0.01. Depois geramos o eixo *y*, com  $y = x^{**2}$ . Com *plt.plot(x,y)* geramos o gráfico. O *plt.show()* mostra o gráfico na tela e *plt.savefig()* salva o arquivo com o gráfico. Efetivamente com 7 linhas de código geramos nosso gráfico.

```
# Program to illustrate the use of Matplotlib
import matplotlib.pyplot as plt
import numpy as np

# Generates x-axis
x = np.arange(0,8,0.01)

# Generates y-axis
y = x**2

# Generates plot
plt.plot(x,y)

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("simple_plot2.png")
```

Abaixo temos o gráfico gerado.

```
# Program to illustrate the use of Matplotlib
import matplotlib.pyplot as plt
import numpy as np

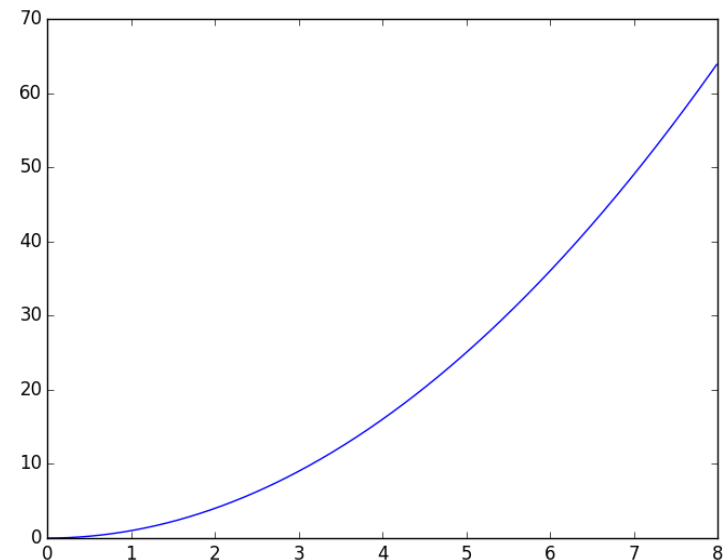
# Generates x-axis
x = np.arange(0,8,0.01)

# Generates y-axis
y = x**2

# Generates plot
plt.plot(x,y)

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("simple_plot2.png")
```



Vejamos agora um programa com gráficos múltiplos. O programa *multiple\_plots1.py* gera três curvas num mesmo gráfico para a mesma faixa de x.

```
# Program to illustrate the use of Matplotlib
```

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

```
import numpy as np
```

```
# Generates x-axis
```

```
x = np.arange(0,8,0.01)
```

```
# Generates y-axis and respective plot
```

```
y = x**2
```

```
plt.plot(x,y)
```

```
# Generates y-axis and respective plot
```

```
y = x**2.5
```

```
plt.plot(x,y)
```

```
# Generates y-axis and respective plot
```

```
y = x**3
```

```
plt.plot(x,y)
```

```
# Shows plot
```

```
plt.show()
```

```
# Saves plot on png file
```

```
plt.savefig("multiple_plots1.png")
```



A principal novidade do código, é que geramos o eixo *y* três vezes e, em cada vez, chamamos o *plt.plot(x,y)*. Como mudamos o *y*, temos três gráficos distintos.

```
# Program to illustrate the use of Matplotlib
```

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

```
import numpy as np
```

```
# Generates x-axis
```

```
x = np.arange(0,8,0.01)
```

```
# Generates y-axis and respective plot
```

```
y = x**2
```

```
plt.plot(x,y)
```

```
# Generates y-axis and respective plot
```

```
y = x**2.5
```

```
plt.plot(x,y)
```

```
# Generates y-axis and respective plot
```

```
y = x**3
```

```
plt.plot(x,y)
```

```
# Shows plot
```

```
plt.show()
```

```
# Saves plot on png file
```

```
plt.savefig("multiple_plots1.png")
```

Abaixo temos o gráfico gerado. Veja que a biblioteca *Matplotlib* gera as curvas em cores diferentes, a primeira curva em azul, depois verde e por último vermelha.

```
# Program to illustrate the use of Matplotlib
import matplotlib.pyplot as plt
import numpy as np

# Generates x-axis
x = np.arange(0,8,0.01)

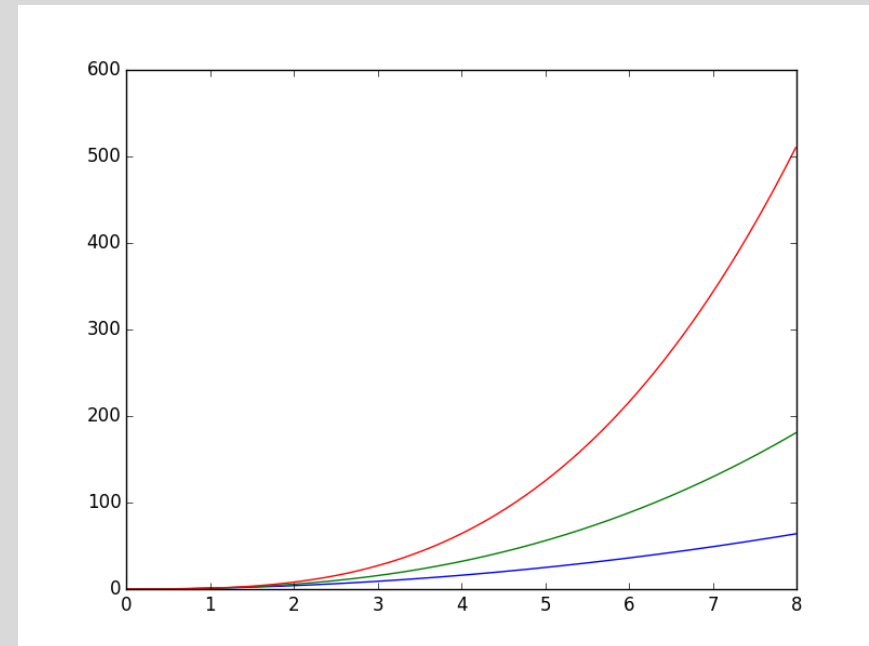
# Generates y-axis and respective plot
y = x**2
plt.plot(x,y)

# Generates y-axis and respective plot
y = x**2.5
plt.plot(x,y)

# Generates y-axis and respective plot
y = x**3
plt.plot(x,y)

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("multiple_plots1.png")
```



Podemos gerar as três curvas com um `plt.plot()` somente, como indicado na linha de código em vermelho abaixo. O programa *multiple\_plots2.py* tem um número menor de linhas e gera o mesmo gráfico.

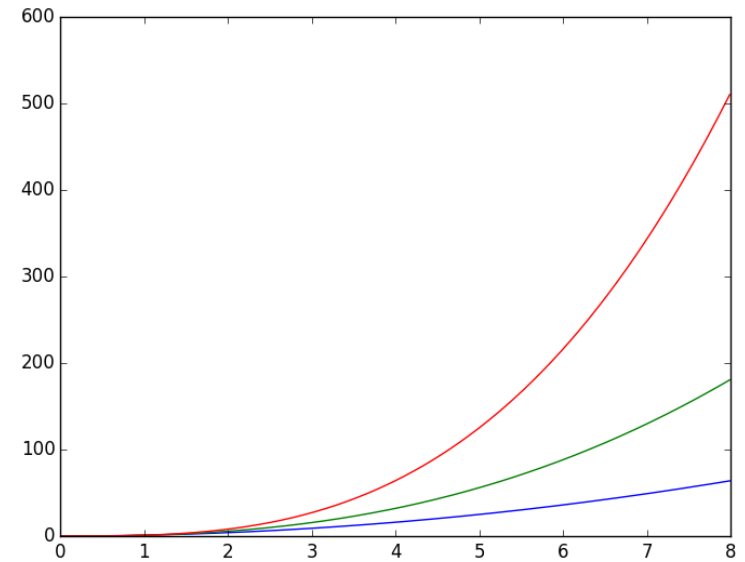
```
# Program to illustrate the use of Matplotlib
import matplotlib.pyplot as plt
import numpy as np

# Generates x-axis
x = np.arange(0,8,0.01)

# Generates all plots in one command
plt.plot(x,x**2,x,x**2.5,x,x**3 )

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("multiple_plots2.png")
```



Vimos com o programa *trig2.py*, como incluir informações diversas, tais como, rótulos para os eixos, título do gráfico, além do gradeado de fundo do gráfico, como se fosse um papel milimetrado. Agora vamos colocar legendas no nosso gráfico. Para incluirmos tais legendas, adicionamos a palavra chave *label*, como argumento da função *plt.plot()*. Por exemplo, para adicionarmos a legenda “Function  $x^2$ ”, temos que usar o comando:

```
plt.plot(x,x**2,label="Function  $x^2$ " )
```

Após a inclusão da palavra chave “label”, precisamos da função *plt.legend()*, que insere a legenda criada no gráfico. As opções de posicionamento da legenda, com a palavra chave *loc*, são as seguintes: *best*, *upper right*, *upper left*, *lower left*, *lower right*, *right*, *center left*, *center right*, *lower center*, *upper center*, *center*. Abaixo temos o comando, para posicionarmos a legenda à esquerda acima.

```
plt.legend(loc='upper left')
```

No próximo slide, temos o código completo para o programa *multiple\_plot3.py* .

O programa *multiple\_plot3.py* traz informações completas sobre o gráfico gerado.

```
# Program to illustrate the use of Matplotlib
import matplotlib.pyplot as plt
import numpy as np
# Generates x-axis
x = np.arange(0,8,0.01)

# Generates plots with legends
plt.plot(x,x**2,label="Function x**2" )
plt.plot(x,x**2.5,label="Function x**2.5" )
plt.plot(x,x**3,label="Function x**3" )

# Positioning the legends
plt.legend(loc='upper left')

plt.xlabel('x-axis')           # Adds axis label
plt.ylabel('y-axis')          # Adds axis label
plt.title('Multiple Functions') # Adds title
plt.grid(True)                # Adds grid to the plot

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("multiple_plots3.png")
```

O gráfico está mostrado abaixo.

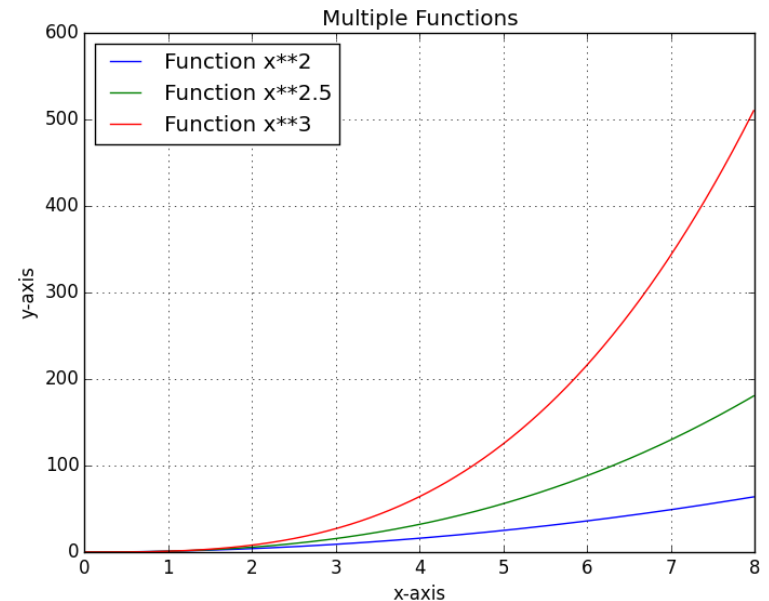
```
# Program to illustrate the use of Matplotlib
import matplotlib.pyplot as plt
import numpy as np
# Generates x-axis
x = np.arange(0,8,0.01)
```

```
# Generates plots with legends
plt.plot(x,x**2,label="Function x**2" )
plt.plot(x,x**2.5,label="Function x**2.5" )
plt.plot(x,x**3,label="Function x**3" )
```

```
# Positioning the legends
plt.legend(loc='upper left')
```

```
plt.xlabel('x-axis')           # Adds axis label
plt.ylabel('y-axis')          # Adds axis label
plt.title('Multiple Functions') # Adds title
plt.grid(True)                 # Adds grid to the plot
```

```
# Shows plot
plt.show()
# Saves plot on png file
plt.savefig("multiple_plots3.png")
```





A biblioteca *Matplotlib* tem diversas opções de gráficos, como histograma. O programa *histogram\_plot1.py* gera uma lista de números aleatórios e faz o gráfico da ocorrência destes números, dividido em faixas. Para gerar os números aleatórios, usamos a função *np.random.randn(1000)*, que gera um *array* com 1000 números, que seguem uma distribuição gaussiana. Para mostrar o histograma, usamos *plt.hist(y)*, que gera o histograma dividido em 10 faixas. O valor 10 é o padrão, podemos modificar o número de faixas, a partir da inclusão do número de faixas desejado, como argumento da função *plt.hist()*, por exemplo: *plt.hist(y,20)* gera um histograma com 20 divisões.

```
# Program to generate a histogram plot for a distribution of pseudo-random numbers

import matplotlib.pyplot as plt
import numpy as np

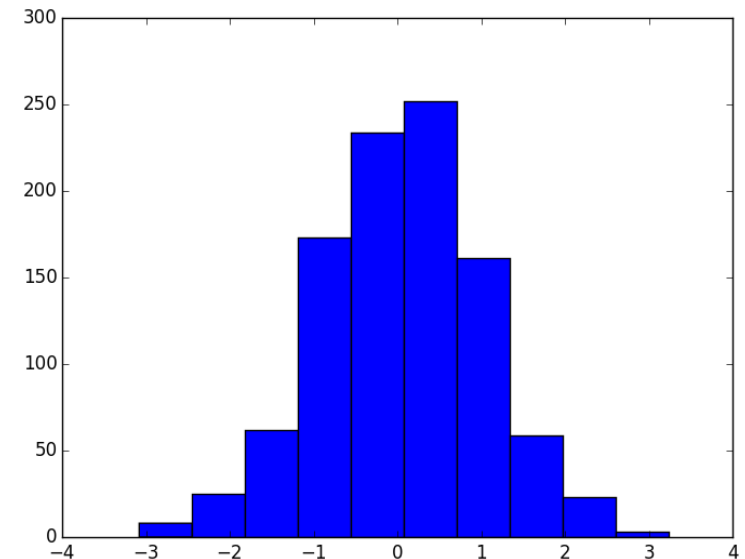
# Generates an array of pseudo-random numbers using np.random.randn()
y = np.random.randn(1000)

# Generates plot
plt.hist(y)

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("histogram_plot1.png")
```

Abaixo temos o histograma gerado, vemos que o eixo y indica a ocorrência da faixa indicada no eixo x.



```
# Program to generate a histogram plot for a d
import matplotlib.pyplot as plt
import numpy as np

# Generates an array of pseudo-random numbers
y = np.random.randn(1000)

# Generates plot
plt.hist(y)

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("histogram_plot1.png")
```

O programa *histogram\_plot2.py* gera um histograma com 20 divisões, como mostrada na figura abaixo. Só modificamos as linhas indicadas em vermelho.

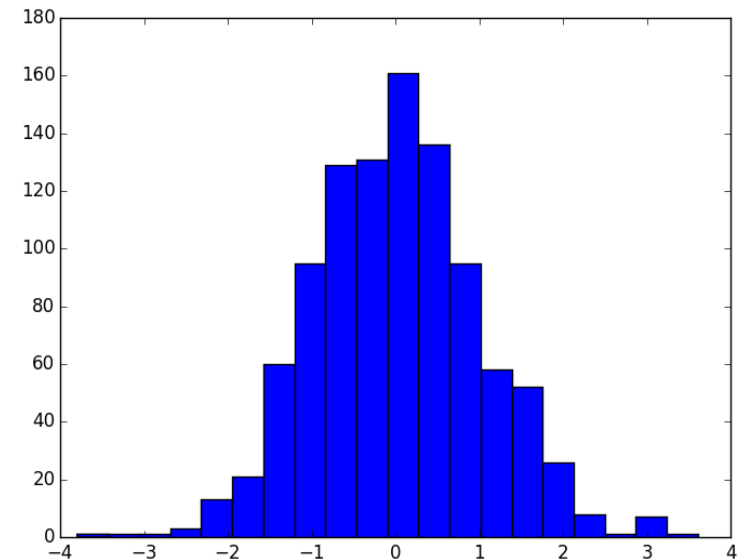
```
# Program to generate a histogram plot for a d
import matplotlib.pyplot as plt
import numpy as np

# Generates an array of pseudo-random numbers
y = np.random.randn(1000)

# Generates plot
plt.hist(y, 20)

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("histogram_plot2.png")
```



Outro gráfico usado para estudo de distribuições, é o gráfico de dispersão. Na biblioteca *Matplotlib* temos a função *plt.scatter()*, que gera gráficos de dispersão. No programa *scatter\_plot1.py*, temos a aplicação da função *np.random.randn(1000)* duas vezes, uma para gerar os números do eixo x e outra para gerar os números do eixo y. A função *plt.scatter(x,y)* faz o gráfico de distribuição dos números aleatórios. O código está mostrado abaixo.

```
# Program to generate a scatter plot for two distributions of pseudo-random numbers

import matplotlib.pyplot as plt
import numpy as np

# Generates two arrays of pseudo-random numbers using np.random.randn()
x = np.random.randn(1000)
y = np.random.randn(1000)

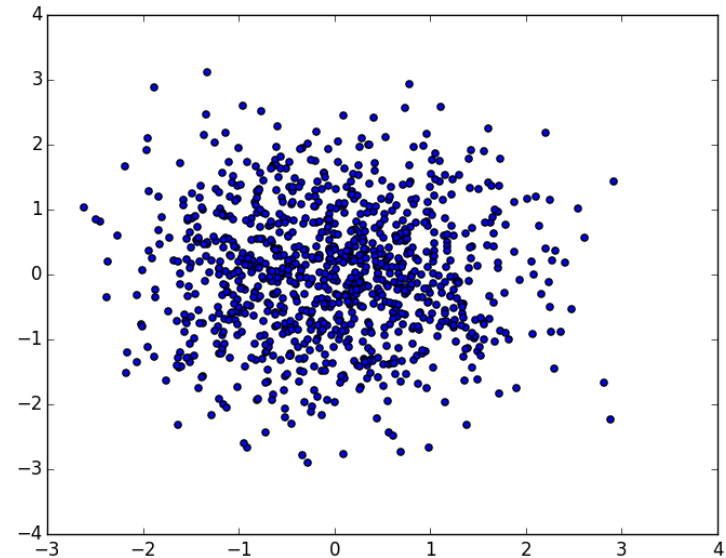
# Generates plot
plt.scatter(x, y)

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig("scatter_plot1.png")
```

Abaixo temos o gráfico de espalhamento gerado.

```
# Program to generate a scatter plot for two  
  
import matplotlib.pyplot as plt  
import numpy as np  
  
# Generates two arrays of pseudo-random numbers  
x = np.random.randn(1000)  
y = np.random.randn(1000)  
  
# Generates plot  
plt.scatter(x, y)  
  
# Shows plot  
plt.show()  
  
# Saves plot on png file  
plt.savefig("scatter_plot1.png")
```



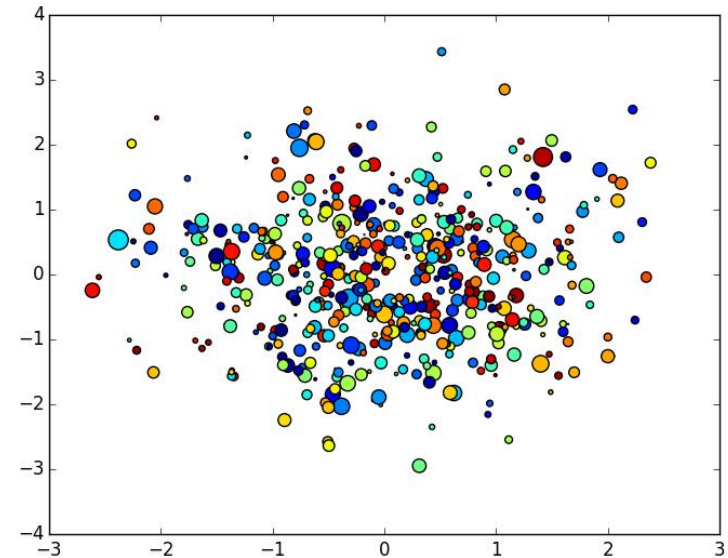
Vamos modificar o código *scatter\_plot1.py*, para que tenhamos cores nos pontos do gráfico. Além disso, vamos variar os tamanhos dos pontos. As cores e os tamanhos dos pontos irão variar de forma aleatória. Para gerarmos cores, usarmos a palavra chave *c* como argumento da função *plt.plot()*. No caso do tamanho do ponto do gráfico, usamos a palavra chave *s*. Assim, criamos distribuições aleatórias para variáveis *size* e *colors*. Para a variável *size*, temos que multiplicar por 50, visto que é expresso em pixels. O código está mostrado abaixo.

```
# Program to generate a scatter plot for two distributions of pseudo-random numbers,
using also random number
# for size and colors
import matplotlib.pyplot as plt
import numpy as np
# Generates four arrays of pseudo-random numbers using np.random.randn()
x = np.random.randn(1000)
y = np.random.randn(1000)
size = 50*np.random.randn(1000)
colors = np.random.rand(1000)
# Generates plot
plt.scatter(x, y, s = size, c = colors)

# Shows plot
plt.show()
# Saves plot on png file
plt.savefig("scatter_plot2.png")
```

O gráfico gerado está mostrado abaixo.

```
# Program to generate a scatter plot for two  
# using also random number  
# for size and colors  
import matplotlib.pyplot as plt  
import numpy as np  
# Generates four arrays of pseudo-random numbers  
x = np.random.randn(1000)  
y = np.random.randn(1000)  
size = 50*np.random.randn(1000)  
colors = np.random.rand(1000)  
# Generates plot  
plt.scatter(x, y, s = size, c = colors)  
  
# Shows plot  
plt.show()  
# Saves plot on png file  
plt.savefig("scatter_plot2.png")
```





Do ponto de vista de aplicação científica, seria interessante a possibilidade de lermos os pontos dos *arrays* *x* e *y* a partir de um arquivo externo. Você provavelmente já fez este processo, por exemplo, quando abriu uma planilha do *Excel*. O programa *Excel* usa como um dos formatos para suas planilhas, o formato CSV (*Comma Separated Values*). Arquivo no formato CSV tem os valores separados por vírgulas, como mostrado no trecho abaixo.

```
Name,MolDock Score,Rerank Score,RMSD,Interaction,Internal,HBond,LE1,LE3,Docking Score,DisplacedWater
[291]SCF_501 [A],-135.265,-108.212,0.550558,-136.511,16.8594,-3.70663,-5.63606,-4.50883,-135.927,-15.6138
[278]SCF_501 [A],-135.279,-108.201,0.552119,-136.537,16.8903,-3.69755,-5.63661,-4.5084,-135.93,-15.632
[261]SCF_501 [A],-135.287,-108.197,0.553228,-136.515,16.8589,-3.69596,-5.63694,-4.50822,-135.939,-15.6301
[258]SCF_501 [A],-135.29,-108.195,0.551817,-136.507,16.8532,-3.69173,-5.63708,-4.50813,-135.942,-15.6365
[267]SCF_501 [A],-135.289,-108.18,0.553158,-136.595,16.9356,-3.68408,-5.63702,-4.5075,-135.935,-15.6291
[299]SCF_501 [A],-135.282,-108.164,0.554487,-136.586,16.9128,-3.67745,-5.63676,-4.50681,-135.925,-15.609
[256]SCF_501 [A],-135.297,-108.161,0.554416,-136.553,16.8902,-3.68148,-5.63736,-4.5067,-135.943,-15.6342
[252]SCF_501 [A],-135.295,-108.159,0.553906,-136.553,16.8732,-3.68676,-5.63729,-4.50663,-135.947,-15.6148
[257]SCF_501 [A],-135.291,-108.158,0.553521,-136.558,16.8822,-3.68676,-5.63712,-4.50657,-135.943,-15.6151
[265]SCF_501 [A],-135.291,-108.153,0.553381,-136.626,16.9517,-3.68131,-5.63712,-4.50638,-135.937,-15.6163
[272]SCF_501 [A],-135.283,-108.153,0.553865,-136.602,16.9356,-3.69094,-5.63678,-4.50639,-135.934,-15.6167
```

A função `np.genfromtxt('data1.csv', delimiter=",", skip_header = 1)` lê o arquivo `data1.csv` e considera as colunas de dados estão separadas por vírgulas, opção indicada por `delimiter=","`. Além disso, é desconsiderada a primeira linha do arquivo CSV, com a opção `skip_header = 1`, que pula a primeira linha. Caso quiséssemos pular duas linhas, usaríamos `skip_header = 2`, e assim sucessivamente. Abaixo temos a indicação dos principais argumentos da função `np.genfromtxt()`.

```
np.genfromtxt('data1.csv', delimiter=",", skip_header = 1)
```

Número de linhas que serão puladas a partir da primeira, no caso pula-se a primeira linha.

Indica o símbolo usado para separar as colunas, no caso são vírgulas “,”.

Nome do arquivo CSV

Abaixo temos os dados do arquivo *data1.csv*. Veja que com *skip\_header = 1*, pulamos a primeira linha, que tem os nomes dos eixos.

```
np.genfromtxt('data1.csv', delimiter=",", skip_header = 1)
```

**x,y**

```
0.1,1.25
0.2,1.5
0.3,4.0
0.4,5.1
0.5,6.3
0.6,5.0
0.7,8.7
0.8,9.0
0.9,11.0
1.0,14.0
1.1,13.75
1.2,15.0
1.3,18.0
1.4,17.5
1.5,19.0
1.6,17.0
1.7,21.0
1.8,22.5
1.9,26.0
2.0,25.1
```

Para a leitura das colunas do arquivo CSV, usamos o comando `x = my_csv[:,0]`, que lê toda a primeira coluna (coluna zero). Abaixo temos um arquivo CSV, cada número no arquivo é acessado de forma matricial. Assim, considerando que começamos no número zero, o número na caixa vermelha é o `my_csv[5,0]`. O número da caixa azul é o `my_csv[11,1]`.

	0.1,1.25	
	0.2,1.5	
	0.3,4.0	
	0.4,5.1	
	0.5,6.3	
Linha 5, coluna 0	0.6,5.0	
	0.7,8.7	
	0.8,9.0	
	0.9,11.0	
	1.0,14.0	
	1.1,13.75	
	1.2,15.0	Linha 11, coluna 1
	1.3,18.0	
	1.4,17.5	
	1.5,19.0	
	1.6,17.0	
	1.7,21.0	
	1.8,22.5	
	1.9,26.0	
	2.0,25.1	

Para varremos toda coluna zero, temos que manter o número da coluna zero, e varrer todas as linhas, ou seja, poderíamos usar `my_csv[0:19,0]`. A indicação `0:19` representa que tomamos da linha 0 até a linha 19. O problema é que este comando funcionaria só para arquivo com 20 linhas. Para tornarmos de uso geral, para qualquer número de linha, usamos a faixa, com `:`, ou seja, `my_csv[:,0]`.

	0.1, 1.25	
	0.2, 1.5	
	0.3, 4.0	
	0.4, 5.1	
	0.5, 6.3	
Linha 5, coluna 0	0.6, 5.0	
	0.7, 8.7	
	0.8, 9.0	
	0.9, 11.0	
	1.0, 14.0	
	1.1, 13.75	
	1.2, 15.0	Linha 11, coluna 1
	1.3, 18.0	
	1.4, 17.5	
	1.5, 19.0	
	1.6, 17.0	
	1.7, 21.0	
	1.8, 22.5	
	1.9, 26.0	
	2.0, 25.1	

O código está mostrado abaixo. O bloco em vermelho destaca a parte da leitura do arquivo CSV, `x = my_csv[:,0]` lê toda a primeira coluna (coluna zero).

```
import numpy as np
import matplotlib.pyplot as plt

# Reads CSV file
my_csv = np.genfromtxt ('data1.csv', delimiter=",", skip_header = 1)

# Gets each column from CSV file
x = my_csv[:,0]
y = my_csv[:,1]

# Generates plot
plt.scatter(x,y)

#Least-squares polynomial fitting
z = np.polyfit(x,y, 1)
p = np.poly1d(z)
print("Best fit polinomial equation: ",p)

# Generates plot
plt.plot(x, p(x), '-')

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig('scatter_plot3.png')
```

A função `polyfit(x,y,1)` gera uma reta aproximada para os pontos (x,y). O número “1” indica uma reta (grau 1). Podemos gerar modelos para outros polinômios (graus 2,...).

```
import numpy as np
import matplotlib.pyplot as plt

# Reads CSV file
my_csv = np.genfromtxt ('data1.csv', delimiter=",", skip_header = 1)

# Gets each column from CSV file
x = my_csv[:,0]
y = my_csv[:,1]

# Generates plot
plt.scatter(x,y)

# Least-squares polynomial fitting
z = np.polyfit(x,y, 1)
p = np.poly1d(z)

print("Best fit polynomial equation: ",p)

# Generates plot
plt.plot(x, p(x), '-')

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig('scatter_plot3.png')
```



A função `poly1d(x)` gera a função matemática determinada pela função `polyfit(x,y,1)`, que foi atribuída à variável `p`.

```
import numpy as np
import matplotlib.pyplot as plt

# Reads CSV file
my_csv = np.genfromtxt ('data1.csv', delimiter=",", skip_header = 1)

# Gets each column from CSV file
x = my_csv[:,0]
y = my_csv[:,1]

# Generates plot
plt.scatter(x,y)

# Least-squares polynomial fitting
z = np.polyfit(x,y, 1)
p = np.poly1d(z)

print("Best fit polynomial equation: ",p)

# Generates plot
plt.plot(x, p(x), '-')

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig('scatter_plot3.png')
```

Ao usarmos  $p(x)$ , temos um *array* onde para cada elemento do *array*  $x$  foi calculado o valor  $p(x)$ . No caso os valores de  $p(x)$  serão usados como coordenadas do eixo  $y$ .

```
import numpy as np
import matplotlib.pyplot as plt

# Reads CSV file
my_csv = np.genfromtxt ('data1.csv', delimiter=",", skip_header = 1)

# Gets each column from CSV file
x = my_csv[:,0]
y = my_csv[:,1]

# Generates plot
plt.scatter(x,y)

# Least-squares polynomial fitting
z = np.polyfit(x,y, 1)
p = np.poly1d(z)

print("Best fit polynomial equation: ",p)

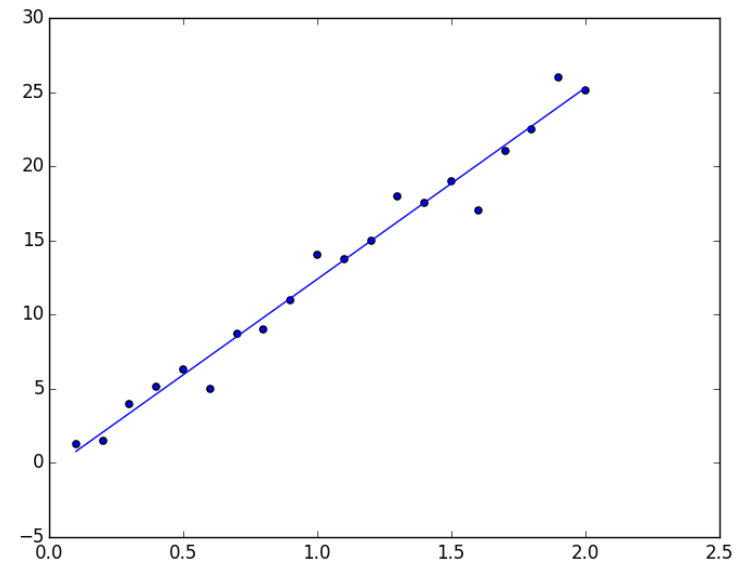
# Generates plot
plt.plot(x, p(x), '-')

# Shows plot
plt.show()

# Saves plot on png file
plt.savefig('scatter_plot3.png')
```

Abaixo temos informações sobre a reta obtida pelo programa e o gráfico gerado.

*Best fit polynomial equation:*  
 $2.047x - 0.2116$



## Scatter Plot (Versão 4)

Programa: *scatter\_plot4.py*

### Resumo

Programa para gerar o gráfico de dispersão para os dados armazenados num arquivo CSV cujo o nome é dado pelo usuário. O grau do polinômio também é fornecido pelo usuário. Teste o programa para o arquivo data2.csv.

Até o momento vimos dados do tipo sequência, que são strings e números, bem como **listas**, que são formas de colocarmos um conjunto de dados e relacioná-los com sua posição numa lista, por exemplo, à variável *week\_list* foi atribuída uma lista com strings de três letras para os dias da semana, como segue:

```
week_list = ["SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT"]
```

Assim, se quisermos mostrar todo o conteúdo atribuído à variável *week\_list*, basta usarmos um loop *for* e colocarmos uma função *print()*, como mostrado abaixo.

```
for day in week_list:  
    print(day)
```

A execução do código mostrará um dia por linha, o arquivo *show\_day1.py* tem o código do programa.

Outra estrutura de dados disponível em Python é chamada **dicionário**. Funciona com se fosse um dicionário comum, onde temos associada a um verbete uma definição. Assim, na definição de um dicionário em Python, teremos pares, onde para uma **chave** colocada à esquerda, teremos um **valor** à direita. Vejamos um exemplo, o dicionário *week\_dict* tem strings de três letras para os dias da semana (chaves) e, à cada chave, foi atribuída uma string com o dia da semana em inglês (valor), como indicado abaixo:

```
week_dict = {"SUN": "Sunday", "MON": "Monday", "TUE": "Tuesday",  
            "WED": "Wednesday", "THU": "Thursday", "FRI": "Friday",  
            "SAT": "Saturday"}
```

Cada definição do dicionário é chamada de **item**, assim temos 7 itens em *week\_dict*. Para chamarmos uma definição do dicionário, podemos usar a chave para recuperar o valor, como segue:

```
print(week_dict["SUN"])
```

A função `print(week_dict["SUN"])` mostrará o valor atribuído à string "SUN", ou seja, "Sunday".

Vejamos o programa *show\_day2.py*, que mostra os valores associados às chaves que trazem as strings de três letras.

```
week_list = ["SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT"]

week_dict = {"SUN": "Sunday", "MON": "Monday", "TUE": "Tuesday",
             "WED": "Wednesday", "THU": "Thursday", "FRI": "Friday",
             "SAT": "Saturday"}

for day in week_list:
    print(week_dict[day])
```

No código temos a lista *week\_list* com os dias, com as strings de três letras. No dicionário, temos associado à cada string de três letras o valor com a string com o dia da semana em inglês. O loop *for* mostra para cada chave lida da lista *week\_list*, o valor associado. O resultado é que temos os dias para cada string de três letras mostrados na tela. Veja que na formação de qualquer dicionário, sempre temos um elemento à esquerda, chamado de chave, separado por dois pontos (:), associado a um valor, que pode ser um número, string ou lista.

Todos os programas que estudamos até o momento eram sequências de comandos em Python, que começam sua execução do primeiro até o último comando. Tal abordagem de programação funciona bem para programas pequenos, até poucas centenas de linhas de código. Para programas maiores, tal abordagem pode causar problemas e dificultar a programação. Uma forma de superarmos tal obstáculo, é dividirmos nosso código em pedaços que executam uma determinada tarefa dentro do programa, e isolá-lo do programa principal, só chamando-o quando necessário. Esta parte do programa é o que chamamos de **função**. Para ilustrar o uso de funções em Python, usaremos um programa para o cálculo da sequência de Fibonacci. Esta sequência começa com o número “1”, o segundo número também é “1”, do terceiro número em diante temos a seguinte regra de cálculo, o número é o resultado da soma dos dois números anteriores na sequência, assim o terceiro número é 2, o quarto número é 3, o sexto é 5, e assim vai. Veja abaixo a lista dos 10 primeiros elementos da sequência de Fibonacci.

$$\{1, 1, 2, 3, 5, 8, 13, 21, 55\}$$



# Sequência de Fibonacci

## Programa: *fibonacci.py*

### Resumo

Programa para gerar a sequência de Fibonacci. O número de elementos da sequência é dado pelo usuário. A sequência será gerada na função *generate\_Fibonacci()*, que tem como parâmetro o número de elementos da sequência. O programa tem uma função para mostrar informações sobre o programa, chamada *show\_header()*. Há uma terceira função, chamada *show\_list()*, que mostra a sequência na tela.

Mostraremos inicialmente cada uma das três funções do programa *fibonacci.py*. Na definição de uma função seguimos o seguinte formato, iniciamos com o comando *def* seguido do nome da função, depois temos dois parênteses e, por último, dois pontos : . Para definirmos a função *show\_header()*, seguimos o modelo acima, como indicado em vermelho no código abaixo.

```
def show_header():  
    """Shows information about this program"""  
    print("\nProgram to calculate the Fibonacci sequence")
```

Usamos o comando *def* seguido do nome da função e finalizamos a linha com : . A linha de código *def show\_header()*: diz ao computador que o bloco de comandos a seguir é para ser usado como a função *show\_header()*. Basicamente estamos dizendo que o bloco de comandos chama-se *show\_header()* e, todas as vezes que chamarmos a função, todo o bloco será executado. A linha inicial e os comandos que seguem é o que chamamos **definição da função**. A definição da função não executa os comandos vinculados à ela, simplesmente indica que esse conjunto de comandos está vinculado à função, que pode ser chamada a partir do programa principal. Resumindo, a definição da função mostra o que será feito, quando a função for chamada, mas não executa o código.

A primeira linha do bloco de comandos da função *show\_header()* traz a *docstring*, que é um comentário explicando a tarefa realizada. A *docstring* tem que estar entre três aspas duplas. Seu uso é facultativo, mas se for usá-la, esta tem que vir logo em seguida à linha com a definição da função, e, como destacado, entre três aspas duplas. A função *show\_header()* mostra uma mensagem na tela com a função *print()*. Para chamar a função *show\_header()*, temos que programar sua chamada a partir do programa principal. Veja que os comandos, que serão executados, quando a função for chamada, estão recuados, como fazemos com blocos de comandos para loops e *ifs*. O recuo identifica o bloco de comandos vinculados à função.

```
def show_header():  
    """Shows information about this program"""  
    print("\nProgram to calculate the Fibonacci sequence")
```

Assim, teremos mais para frente do código do programa principal, uma linha com *show\_header()*. A função *show\_header()* não tem entradas e não retorna valores, simplesmente mostra uma mensagem na tela ao ser chamada. A seguir temos uma função com entrada e saída.

A próxima função gera os números da sequência. Esta função precisa de uma entrada, ou seja, o número de elementos da sequência. Chamamos a entrada de **parâmetro da função**, na presente função é *number\_of\_elements*. Veja que *number\_of\_elements* aparece entre parênteses na definição da função *generate\_Fibonacci()*, indicada em vermelho no código abaixo.

```
def generate_Fibonacci(number_of_elements):  
    """This function generates the Fibonacci sequence"""  
    fibonacci = []  
    fibonacci.append(1)  
    fibonacci.append(1)  
    # Looping through to generate Fibonacci sequence  
    for i in range(2, number_of_elements):  
        fibonacci.append(fibonacci[i-2] + fibonacci[i-1])  
    return fibonacci
```

Podemos pensar no parâmetro como uma variável dentro de parêntesis na definição da função. Podemos ter vários parâmetros na função. Quando a função *generate\_Fibonacci()* for chamada no programa principal, é necessário que tenhamos um valor indicado entre parênteses. Esse valor entre parênteses é **o argumento da função**. Por exemplo, no programa principal temos a chamada da função com a linha de código *my\_list = generate\_Fibonacci(number)*, ao parâmetro *number\_of\_elements* será atribuído o valor de *number*, que é o argumento da função.

A primeira linha da função é a *docstring* e, em seguida, os comandos para a sequência de Fibonacci. Definimos a lista *fibonacci*. Atribuiremos, a cada elemento da lista *fibonacci*, o número correspondente. Ao primeiro e o segundo elementos são atribuídos o número “1”. Usamos o método *.append()*, para atribuirmos valores aos elementos da lista. Em seguida temos um loop *for*, para gerarmos os próximos elementos da sequência. O limite superior do loop é indicado pela variável *number\_of\_elements* que, como todo parâmetro de uma função, é considerada uma variável interna. Depois do loop *for*, temos o comando *return fibonacci*, que traz a lista *fibonacci* para a posição do programa principal, onde a função *generate\_Fibonacci()* foi chamada. A função, ao ser chamada, tem que ter o valor a ser atribuído ao parâmetro *number\_of\_elements*, caso contrário temos uma mensagem de erro. O valor, retornado pela função *generate\_Fibonacci()*, é atribuído à variável *my\_list* no programa principal.

```
def generate_Fibonacci(number_of_elements):  
    """This function generates the Fibonacci sequence"""  
    fibonacci = []  
    fibonacci.append(1)  
    fibonacci.append(1)  
    # Looping through to generate Fibonacci sequence  
    for i in range(2,number_of_elements):  
        fibonacci.append(fibonacci[i-2] + fibonacci[i-1])  
    return fibonacci
```

Nossa última função tem como parâmetro a lista a ser mostrada na tela, a função *show\_list()*. Veja que os parâmetros podem ser strings, listas ou até dicionários. O parâmetro *list\_in* traz a sequência de Fibonacci, que foi gerada na função *generate\_Fibonacci()*. A função *show\_list()* tem uma *docstring* e, em seguida, um loop *for* que mostra cada elemento da sequência de Fibonacci, um em cada linha. Depois da execução da função *show\_list()*, o programa retorna ao programa principal. Destacamos que, ao contrário da função *generate\_Fibonacci()*, a função *show\_list()* não retorna valores.

```
def show_list(list_in):  
    """This function shows a list on screen"""  
    for line in list_in:  
        print(line)
```

Agora temos o programa principal. Como o trabalho pesado está implementado nas funções, só precisamos chamar as funções numa sequência lógica. Inicialmente a função *show\_header()*, depois a função *generate\_Fibonacci()* e, por último, a função *show\_list()*. A função *generate\_Fibonacci()*, ao ser chamada no programa principal, retorna a lista e a atribui à variável *my\_list*, que passa a lista para a função *show\_list()*, temos assim um fluxo lógico das informações. A variável *my\_list* é o argumento da função *show\_list()*. Se trocarmos de posições as chamadas das funções *generate\_Fibonacci()* e *show\_list()* no programa principal, teremos um erro na execução do programa. Já a função *show\_header()*, pode ser chamada em qualquer parte do programa principal.

```
# main program
show_header()
number = int(input("\nType the number of elements for the Fibonacci sequence => "))
my_list = generate_Fibonacci(number)
print("\nFibonacci sequence for ",number," elements")
show_list(my_list)
```

O uso de funções facilita a programação e exercita o conceito de **abstração**. A abstração permite que você se concentre no fluxo lógico da informação no programa principal, sem se preocupar com os detalhes de cada função. É uma forma elegante de termos uma visão macroscópica do nosso programa.

Outro aspecto do uso de funções, é a reciclagem das funções. Uma vez que você tenha programado uma função, que realiza uma determinada tarefa, ao necessitar de tal tarefa, em outro programa, você pode anexá-la ao novo programa. Tal prática aumenta a eficiência e acelera o processo de programação.

Por último, destacamos que as variáveis das funções não podem ser acessadas fora das funções, esta característica chama-se **encapsulamento**. Por exemplo, a lista atribuída à variável *fibonacci*, da função *generate\_Fibonacci()*, não pode ser acessada fora da função. Assim temos o comando *return fibonacci*, que atribui a lista *fibonacci* à variável *my\_list*, no programa principal. O encapsulamento garante que os parâmetros e as variáveis, criadas dentro das funções, restrinjam-se ao domínio dessas funções, não podendo ser acessadas de outras funções ou do programa principal.

```
def generate_Fibonacci(number_of_elements):  
    """This function generates the Fibonacci sequence"""  
    fibonacci = []  
    fibonacci.append(1)  
    fibonacci.append(1)  
    # Looping through to generate Fibonacci sequence  
    for i in range(2,number_of_elements):  
        fibonacci.append(fibonacci[i-2] + fibonacci[i-1])  
    return fibonacci
```



Abaixo temos o resultado de rodarmos o programa para uma sequência de Fibonacci com 10 elementos.

```
Program to calculate the Fibonacci sequence
```

```
Type the number of elements for the Fibonacci sequence => 10
```

```
Fibonacci sequence for 10 elements
```

```
1  
1  
2  
3  
5  
8  
13  
21  
34  
55
```

# Mostra string, lista e dicionário na tela

## Programa: *show\_data.py*

### Resumo

Programa para mostrar string, lista e dicionário na tela. O programa faz uso de funções específicas para mostrar cada tipo de estrutura de dados.

Destacamos que os parâmetros das funções podem ser strings, listas ou dicionários. Vamos ilustrar as três situações com um programa que tem funções específicas para cada estrutura de dados. Temos uma função, chamada *show\_string()*, que mostrará uma string na tela. A segunda função nós já vimos, é a *show\_list()*, que mostra uma lista na tela. A última função, chamada *show\_dictionary()*, mostra um dicionário na tela. Todas as funções têm como parâmetro a estrutura de dados a ser mostrada na tela.

Abaixo temos a função *show\_string()*, o parâmetro é *string\_in*. Temos uma linha de comando na função, a função *print()* que mostra a string. A função não retorna valores ao programa principal.

```
def show_string(string_in):  
    """This function shows a string on screen"""  
    print(string_in)
```

A seguir temos a função *show\_list()*, que tem como parâmetro *list\_in* e mostra esta lista na tela.

```
def show_list(list_in):  
    """This function shows a list on screen"""  
    for line in list_in:  
        print(line)
```

A função *show\_dictionary()* tem como parâmetro *my\_dict\_in*, que é mostrado tela.

```
def show_dict(my_dict_in):  
    """This function shows a dictionary on screen"""  
    for line in my_dict_in:  
        print(line, my_dict_in[line])
```

No programa principal temos uma string atribuída à variável *my\_string*, uma lista atribuída à *week\_list* e um dicionário atribuído à *week\_dict*. Temos que *my\_string*, *week\_list* e *week\_dict* são argumentos usados nas chamadas das funções, ou seja, seus valores serão atribuídos aos respectivos parâmetros, na chamada de cada função.

```
# Main program
def main():
    my_string = "Python"

    week_list = ["SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT"]

    week_dict = {"SUN": "Sunday", "MON": "Monday", "TUE": "Tuesday",
                 "WED": "Wednesday", "THU": "Thursday", "FRI": "Friday",
                 "SAT": "Saturday"}

    print("\nShowing string:")
    show_string(my_string)

    print("\nShowing list:")
    show_list(week_list)

    print("\nShowing dictionary:")
    show_dict(week_dict)

main()
```

Outro ponto a ser destacado, definimos o programa principal como uma função, a função *main()*. Tal procedimento não é obrigatório, mas padroniza as funções, onde até o programa principal é uma função que é evocada com o *main()* no final. A definição da função *main()* segue as regras já vistas.

```
# Main program
def main():
    my_string = "Python"

    week_list = ["SUN", "MON", "TUE", "WED", "THU", "FRI", "SAT"]

    week_dict = {"SUN": "Sunday", "MON": "Monday", "TUE": "Tuesday",
                 "WED": "Wednesday", "THU": "Thursday", "FRI": "Friday",
                 "SAT": "Saturday"}

    print("\nShowing string:")
    show_string(my_string)

    print("\nShowing list:")
    show_list(week_list)

    print("\nShowing dictionary:")
    show_dict(week_dict)
```

*main()*

Abaixo temos o resultado de rodarmos o programa.

```
Showing string:  
Python
```

```
Showing list:  
SUN  
MON  
TUE  
WED  
THU  
FRI  
SAT
```

```
Showing dictionary:  
MON Monday  
FRI Friday  
THU Thursday  
SUN Sunday  
TUE Tuesday  
SAT Saturday  
WED Wednesday
```

Agora temos as ferramentas para olharmos o código *inMathematica001.py*. Começaremos com a função *main()*. Temos três funções que são evocadas da função *main()*. A primeira *get\_func\_par()* realiza a leitura da função a partir do teclado. Como esta informação segue a sintaxe do programa *Mathematica*, temos que extrair o que é relevante para usarmos a biblioteca *Matplotlib*. No caso a função retorna *f*, *x1*, *x2*. A variável *f* recebe uma string que será usada para compor a função a ser plotada. As variáveis *x1* e *x2* recebem os valores de mínimo e máximo do eixo x.

```
def main():  
  
    # Call get_func_par()  
    f,x1,x2 = get_func_par()  
  
    # Call write_func()  
    write_func(f,x1,x2)  
  
    # Call plot_func()  
    plot_func(f)  
  
main()
```



A função *write\_func()* escreve a função e os valores limites usando-se o padrão *Matplotlib* num arquivo externo, chamado *func2plot.py*. Vejam que interessante, o programa gera o código de outro programa. Isto é Python! No retângulo à direita temos um código possível para o arquivo *func2plot.py*. Vejam que nada mais é que um série de comandos do *Matplotlib*.

```
def main():
```

```
    # Call get_func_par()
```

```
    f,x1,x2 = get_func_par()
```


```
    # Call write_func()
```

```
    write_func(f,x1,x2)
```

```
    # Call plot_func()
```

```
    plot_func(f)
```

```
main()
```



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

```
import numpy as np
```

```
x = np.linspace(0.0,6.28,200)
```

```
y = np.sin(x)
```

```
plt.plot(x,y)
```

```
plt.show()
```

A função *plot\_func()* gera o gráfico. Na verdade veremos mais para frente que a função *plot\_func()* nada mais é que a execução das linhas de código contidas no arquivo *func2plot.py*.

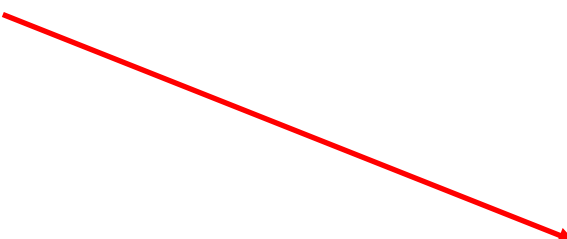
```
def main():
```

```
    # Call get_func_par()
    f, x1, x2 = get_func_par()
```

```
    # Call write_func()
    write_func(f, x1, x2)
```

```
    # Call plot_func()
    plot_func(f)
```

```
main()
```



```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0.0, 6.28, 200)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```

Vejamos as linhas de código da função *get\_func\_par()*.

```
def get_func_par():  
    """Function to read function from command line"""  
    # Get info from keyboard  
    f_ = input("=")  
    f = f_.replace(" ", "")  
    # Set up list with mathematical functions  
    math_func = [ "sin", "cos", "tan", "log10", "log", "exp"]  
    # Set up None values  
    my_func, x_min, x_max = None, None, None  
    # Check if "Plot"  
    if "Plot[" in f:  
        # Some editing  
        my_func0, x_min, x_max = some_func_editing(f)  
        # Looping through math_func  
        for line in math_func:  
            if line in my_func0:  
                my_func = my_func0.replace(line, "np."+line)  
                break  
            else:  
                my_func = my_func0  
    # Return parameters  
    return my_func, x_min, x_max
```

Faz a leitura do teclado (*input()*) e elimina os espaços.

## Continuando...

```
def get_func_par():  
    """Function to read function from command line"""  
  
    # Get info from keyboard  
    f_ = input("=")  
    f = f_.replace(" ", "")  
  
    # Set up list with mathematical functions  
    math_func = [ "sin", "cos", "tan", "log10", "log", "exp"]  
  
    # Set up None values  
    my_func, x_min, x_max = None, None, None  
  
    # Check if "Plot"  
    if "Plot[" in f:  
        # Some editing  
        my_func0, x_min, x_max = some_func_editing(f)  
        # Looping through math_func  
        for line in math_func:  
            if line in my_func0:  
                my_func = my_func0.replace(line, "np."+line)  
                break  
            else:  
                my_func = my_func0  
  
    # Return parameters  
    return my_func, x_min, x_max
```

Prepara uma lista (*math\_func*) com as funções pré-definidas do *Mathematica*

## Continuando...

```
def get_func_par():  
    """Function to read function from command line"""  
    # Get info from keyboard  
    f_ = input("=")  
    f = f_.replace(" ", "")  
    # Set up list with mathematical functions  
    math_func = [ "sin", "cos", "tan", "log10", "log", "exp"]  
    # Set up None values  
    my_func, x_min, x_max = None, None, None  
    # Check if "Plot"  
    if "Plot[" in f:  
        # Some editing  
        my_func0, x_min, x_max = some_func_editing(f)  
        # Looping through math_func  
        for line in math_func:  
            if line in my_func0:  
                my_func = my_func0.replace(line, "np."+line)  
                break  
            else:  
                my_func = my_func0  
    # Return parameters  
    return my_func, x_min, x_max
```

Atribui *None* como valores iniciais para as variáveis:  
*my\_func*, *x\_min*, *x\_max*

## Continuando...

```
def get_func_par():  
    """Function to read function from command line"""  
    # Get info from keyboard  
    f_ = input("=")  
    f = f_.replace(" ", "")  
    # Set up list with mathematical functions  
    math_func = [ "sin", "cos", "tan", "log10", "log", "exp"]  
    # Set up None values  
    my_func, x_min, x_max = None, None, None  
    # Check if "Plot"  
    if "Plot[" in f:  
        # Some editing  
        my_func0, x_min, x_max = some_func_editing(f)  
        # Looping through math_func  
        for line in math_func:  
            if line in my_func0:  
                my_func = my_func0.replace(line, "np."+line)  
                break  
            else:  
                my_func = my_func0  
    # Return parameters  
    return my_func, x_min, x_max
```

Checamos se a palavra-chave Plot foi digitada.

## Continuando...

```
def get_func_par():  
    """Function to read function from command line"""  
    # Get info from keyboard  
    f_ = input("=")  
    f = f_.replace(" ", "")  
    # Set up list with mathematical functions  
    math_func = [ "sin", "cos", "tan", "log10", "log", "exp"]  
    # Set up None values  
    my_func, x_min, x_max = None, None, None  
    # Check if "Plot"  
    if "Plot[" in f:  
        # Some editing  
        my_func0, x_min, x_max = some_func_editing(f)  
        # Looping through math_func  
        for line in math_func:  
            if line in my_func0:  
                my_func = my_func0.replace(line, "np."+line)  
                break  
            else:  
                my_func = my_func0  
    # Return parameters  
    return my_func, x_min, x_max
```

Evocamos a função *some\_func\_editing()* que irá manipular a string atribuída à variável *f*.

## Continuando...

```
def get_func_par():  
    """Function to read function from command line"""  
    # Get info from keyboard  
    f_ = input("=")  
    f = f_.replace(" ", "")  
    # Set up list with mathematical functions  
    math_func = [ "sin", "cos", "tan", "log10", "log", "exp"]  
    # Set up None values  
    my_func, x_min, x_max = None, None, None  
    # Check if "Plot"  
    if "Plot[" in f:  
        # Some editing  
        my_func0, x_min, x_max = some_func_editing(f)  
        # Looping through math_func  
        for line in math_func:  
            if line in my_func0:  
                my_func = my_func0.replace(line, "np."+line)  
                break  
            else:  
                my_func = my_func0  
    # Return parameters  
    return my_func, x_min, x_max
```

Loop *for* para verificar quais funções estão na string atribuída à variável *my\_func0*.



## Continuando...

```
def get_func_par():  
    """Function to read function from command line"""  
    # Get info from keyboard  
    f_ = input("=")  
    f = f_.replace(" ", "")  
    # Set up list with mathematical functions  
    math_func = [ "sin", "cos", "tan", "log10", "log", "exp"]  
    # Set up None values  
    my_func, x_min, x_max = None, None, None  
    # Check if "Plot"  
    if "Plot[" in f:  
        # Some editing  
        my_func0, x_min, x_max = some_func_editing(f)  
        # Looping through math_func  
        for line in math_func:  
            if line in my_func0:  
                my_func = my_func0.replace(line, "np."+line)  
                break  
            else:  
                my_func = my_func0  
    # Return parameters  
    return my_func, x_min, x_max
```

Retorna a função a ser plotada e os limites do eixo x.

Esta função (*some\_func\_editing()*) basicamente edita a string digitada.

```
def some_func_editing(f_in):
```

```
    """Function to carry our some editing"""
```

```
    # Some editing
```

```
    in1 = f_in.index("(")
```

```
    in2 = f_in.index("]")
```

```
    in3 = f_in.index(",")          # First comma
```

```
    in4 = f_in.index("{")
```

```
    f1 = f_in[in4:]
```

```
    in5 = f1.index(",") + in4      # Second comma
```

```
    f2 = f_in[in5+1:]
```

```
    in6 = f2.index(",") + in5 + 1  # Third comma
```

```
    f3 = f_in[in6+1:]
```

```
    in7 = f3.index("}") + in6 + 1
```

```
    # More function editing
```

```
    my_func0 = f_in[in1+1:in3]
```

```
    my_func1 = my_func0.replace("^", "***")    # Take care of "***"
```

```
    my_func2 = my_func1.replace("[", "(")      # Take care of "["
```

```
    my_func3 = my_func2.replace("]", ")")      # Take care of "]"
```

```
    my_func4 = my_func3.lower()                # Take care of uppercase
```

```
    # Get limits
```

```
    x_min = float(f_in[in5+1:in6])
```

```
    x_max = float(f_in[in6+1:in7])
```

```
    return(my_func4, x_min, x_max)
```

O método *.index()* encontra o índice (posição) de um caractere dentro de uma string. Este recurso é usado para acharmos as partes relevantes da string.

Esta função basicamente edita a string.

```
def some_func_editing(f_in):  
    """Function to carry our some editing"""  
  
    # Some editing  
  
    in1 = f_in.index("[")  
    in2 = f_in.index("]")  
    in3 = f_in.index(",")          # First comma  
    in4 = f_in.index("{")  
  
    f1 = f_in[in4:]  
  
    in5 = f1.index(",") + in4      # Second comma  
  
    f2 = f_in[in5+1:]  
  
    in6 = f2.index(",") + in5 + 1  # Third comma  
  
    f3 = f_in[in6+1:]  
  
    in7 = f3.index("}") + in6 + 1  
  
    # More function editing  
    my_func0 = f_in[in1+1:in3]  
  
    my_func1 = my_func0.replace("^", "***")    # Take care of "***"  
  
    my_func2 = my_func1.replace("[", "(")       # Take care of "["  
  
    my_func3 = my_func2.replace("]", ")")       # Take care of "]"  
  
    my_func4 = my_func3.lower()                 # Take care of uppercase  
  
    # Get limits  
  
    x_min = float(f_in[in5+1:in6])  
  
    x_max = float(f_in[in6+1:in7])  
  
    return(my_func4, x_min, x_max)
```

Podemos fatiar uma string com a definição dos colchetes. No código em destaque pegaremos o pedaço da string em *f\_in* que começa em *in1+1* e vai até uma letra antes do *in3*.

Esta função abre o arquivo *func2plot.py* e escreve as informações referentes à função a ser plotada. Este função gera o código *func2plot.py* que é o programa que irá gerar o gráfico. Como destacado, o programa está escrevendo um programa!

```
def write_func(f_in,x1_in,x2_in):  
    """Function to write function to plot"""  
  
    # Open file  
    fo = open("func2plot.py","w")  
  
    # Write function  
    fo.write("import matplotlib.pyplot as plt\n")  
    fo.write("import numpy as np\n")  
    fo.write("x = np.linspace("+str(x1_in)+","+str(x2_in)+",200)\n")  
    fo.write("y = "+f_in+"\n")  
    fo.write("plt.plot(x,y)\n")  
    fo.write("plt.show()\n")  
  
    # Close file  
    fo.close()
```

Por último a função *plot\_func()* que importa a função *func2plot()*. Já usamos o *import* antes. Ele pode ser usado para importamos qualquer código em Python, como o código *func2plot.py*. Veja que ao importarmos não colocamos a extensão *.py*, já é assumido que o que está sendo importado termina com *.py*.

```
def plot_func(f_in):  
    """Function to plot"""  
    print("Plotting ",f_in)  
    import func2plot
```

Na aula de hoje, veremos a solução dos exercícios de programação, propostos na aula anterior. Os programas são os seguintes.

**Exercício de programação 2.** Escreva um programa que simula um biscoito da sorte chinês. O programa deve mostrar uma entre cinco previsões, de forma aleatória, cada vez que é executado. Nome do programa: *fortune\_cookie.py*.

**Exercício de programação 3.** Escreva um programa que simula o lançamento de uma moeda 100 vezes. Depois o programa mostra o número de vezes que deu cara e que deu coroa. Nome do programa: *flip\_a\_coin.py*.

**Exercício de programação 4.** Modifique o programa *guess\_my\_number.py*, de forma que o jogador tenha um número limitado de tentativas. Se o jogador não consegue acertar o número gerado pelo computador, num número definido de tentativas, serão mostradas na tela o número certo e uma mensagem para o jogador. Nome do programa: *limited\_guess\_my\_number.py*.

**Exercício de programação 5.** Escreva um programa, onde o jogador troca de lugar com o computador. Isto é, o jogador escolhe um número aleatório entre 1 e 100 e o computador tenta que adivinhar que número é esse. Nome do programa: *robot\_guess\_game.py*.

## Programa do Exercício de programação 2: *fortune\_cookie.py*.

```
import random

print("Welcome to Lotus Restaurant. We are here to bring good Chinese food and also good luck.")
print("Try one of our fortune cookies...")

input("\nPress enter key to see your fortune.")

print("\nYour fortune today is ...")

# Set initial value
fortune = random.randint(1,5)

# Tests all five fortunes and shows it
if fortune == 1 :
    print("'Do not follow where the path may lead. Go where there is no path... and leave a trail.'")
elif fortune == 2 :
    print("'Do not fear what you don't know.'")
elif fortune == 3 :
    print("'You will have a pleasant surprise.'")
elif fortune == 4 :
    print("'If you fell you are right, stand firmly by your convictions.'")
else :
    print("'All progress occurs because people date to be different.'")
```

Vamos ao resultado do programa.

```
Welcome to Lotus Restaurant. We are here to bring good Chinese food and  
also good luck.  
Try one of our fortune cookies...  
  
Press enter key to see your fortune.  
  
Your fortune today is ...  
'You will have a pleasant surprise.'
```



## Programa do Exercício de programação 3: *flip\_a\_coin.py*.

```
import random

print("Throwing a coin 100 times is pretty fast...")

#Set initial value
coin = random.randint(0,1)
tails = 0
heads = 0
count = 0

# Flip a coin loop
while count < 100 :
    if coin == 0 :
        tails += 1
    else :
        heads += 1
    count += 1
    coin = random.randint(0,1)

print("\nI flipped a coin ",count," times and got ",tails," tails and ",heads," heads.")
```

Vamos ao resultado do programa.

```
Throwing a coin 100 times is pretty fast...
```

```
I flipped a coin 100 times and got 44 tails and 56 heads.
```

## Programa do Exercício de programação 4: *limited\_guess\_number.py*.

```
import random

print("Welcome to the number guessing game. In this game you have to guess a number between 1 and 100.")

top_guesses = int(input("How many guesses do you wanna try? "))

random_number = random.randint(1,100) # Set initial values

your_number = int(input("\nGuess a number between 1 and 100 => "))

count = 1

# Guessing loop
while count < top_guesses and your_number != random_number :

    count += 1

    if your_number < random_number :

        your_number = int(input("\nHigher..."))

    elif your_number > random_number :

        your_number = int(input("\nLower..."))

# Checks the results

if count <= top_guesses and your_number == random_number :

    print("\n\nCongratulations! You guessed it right ",random_number, "! It only took you ",count," tries!")

else :

    print("\nYou were unable to guess the right number in ",count, " tries. Better luck next time.")
```

Vamos ao resultado do programa.

```
Welcome to the number guessing game. In this game you have to guess a
number between 1 and 100.
How many guesses do you wanna try? 7

Guess a number between 1 and 100 => 46

Higher...71

Higher...86

Higher...91

Lower...89

Lower...88

Lower...87

Congratulations! You guessed it right 87 ! It only took you 7 tries!
```

## Programa do Exercício de programação 5: *robot\_guess\_game.py*.

```
import random

print("Welcome to the number guessing game.")

print("\nIn this game you have to give a number between 1 and 100 and a robot will try to guess it.")

top_guesses = int(input("\nHow many guesses do you allow the robot to try? Please be fair... "))

secret_number = int(input("\nGive a number between 1 and 100 => "))

max_number = 100

min_number = 1

robot_number = random.randint(min_number,max_number)

count = 1

while count < top_guesses and robot_number != secret_number :

    count += 1

    if robot_number < secret_number :

        print("\nRobot, I guessed ",robot_number,", I will guess higher...")

        min_number = robot_number

        robot_number = random.randint(robot_number+1,max_number)

    elif robot_number > secret_number :

        print("\nRobot, I guessed ",robot_number,", I will guess lower...")

        max_number = robot_number

        robot_number = random.randint(min_number,robot_number-1)

if count <= top_guesses and robot_number == secret_number :

    print("\nCongratulations robot! You guessed it right ",robot_number, "! It only took you ",count," tries!")

else :

    print("\nThe robot was unable to guess the right number in ",count, " tries. Better luck next time.")
```

## Vamos ao resultado do programa.

```
Welcome to the number guessing game.  
In this game you have to give a number between 1 and 100 and a robot will try to guess it.  
How many guesses do you allow the robot to try? Please be fair... 20  
Give a number between 1 and 100 => 46  
  
Robot, I guessed 21 , I will guess higher...  
  
Robot, I guessed 76 , I will guess lower...  
  
Robot, I guessed 54 , I will guess lower...  
  
Robot, I guessed 36 , I will guess higher...  
  
Robot, I guessed 38 , I will guess higher...  
  
Robot, I guessed 52 , I will guess lower...  
  
Robot, I guessed 49 , I will guess lower...  
  
Robot, I guessed 45 , I will guess higher...  
  
Robot, I guessed 48 , I will guess lower...  
  
Robot, I guessed 45 , I will guess higher...  
  
Robot, I guessed 47 , I will guess lower...  
Congratulations robot! You guessed it right 46 ! It only took you 12 tries!
```

- BRESSERT, Eli. SciPy and NumPy. Sebastopol: O'Reilly Media, Inc., 2013. 56 p.
- DAWSON, Michael. **Python Programming, for the absolute beginner**. 3ed. Boston: Course Technology, 2010. 455 p.
- HETLAND, Magnus Lie. **Python Algorithms. Mastering Basic Algorithms in the Python Language**. Nova York: Springer Science+Business Media LLC, 2010. 316 p.
- IDRIS, Ivan. **NumPy 1.5. An action-packed guide dor the easy-to-use, high performance, Python based free open source NumPy mathematical library using real-world examples. Beginner's Guide**. Birmingham: Packt Publishing Ltd., 2011. 212 p.
- KIUSALAAS, Jaan. **Numerical Methods in Engineering with Python**. 2ed. Nova York: Cambridge University Press, 2010. 422 p.
- LANDAU, Rubin H. **A First Course in Scientific Computing: Symbolic, Graphic, and Numeric Modeling Using Maple, Java, Mathematica, and Fortran90**. Princeton: Princeton University Press, 2005. 481p.
- LANDAU, Rubin H., PÁEZ, Manuel José, BORDEIANU, Cristian C. **A Survey of Computational Physics. Introductory Computational Physics**. Princeton: Princeton University Press, 2008. 658 p.
- LUTZ, Mark. **Programming Python**. 4ed. Sebastopol: O'Reilly Media, Inc., 2010. 1584 p.
- TOSI, Sandro. **Matplotlib for Python Developers**. Birmingham: Packt Publishing Ltd., 2009. 293 p.

Última atualização 23 de novembro de 2016.