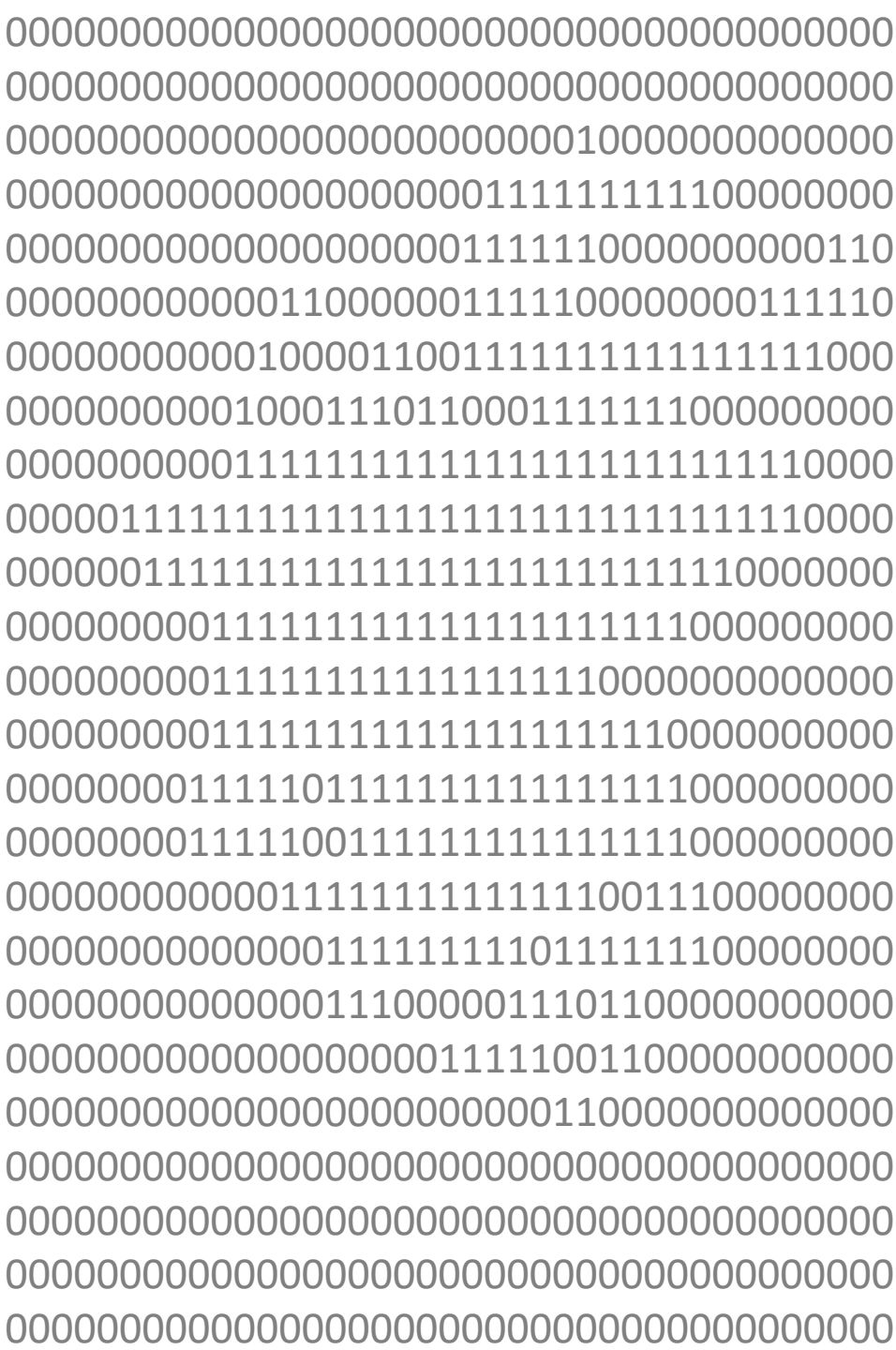


Introdução à Física Computacional

Aula 07

Scatter Plot (Versão 6)

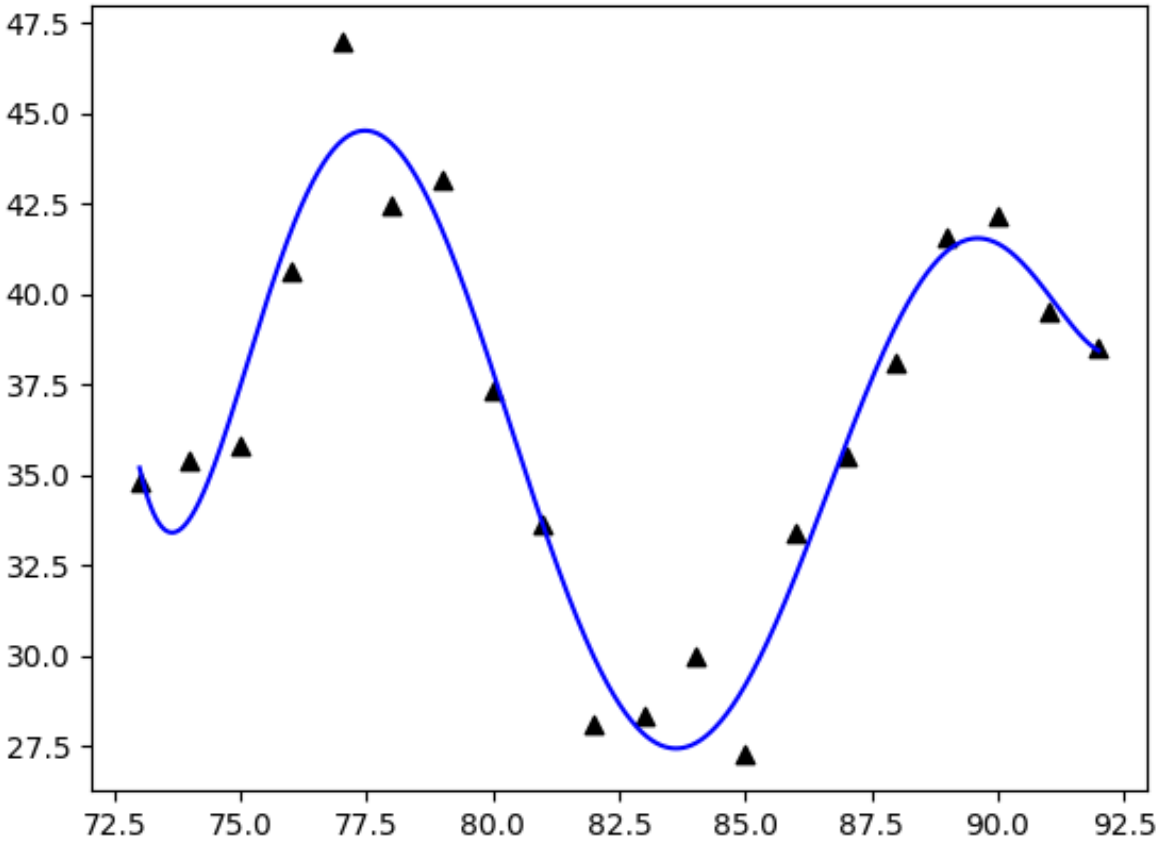
Programa: *scatter_plot6.py*

Resumo
Programa para testar diferentes modelos gerados por regressão linear. O usuário fornece os graus do polinômio menor e maior a serem testados. O programa testa todos os polinômios de graus indicados pelo usuário e mostra o desempenho de cada modelo na tela. Por exemplo, se indicados os graus 1 e 8, o programa gera os polinômios de graus entre 1 e 8 e mostra o desempenho de cada modelo. O programa seleciona o polinômio que gerou o modelo com maior coeficiente de correlação de Spearman (*rho*) e faz o gráfico somente deste melhor. Teste o programa para uma faixa entre 1 e 8. Use o arquivo *data2.csv*. No próximo slide temos a saída esperada para o programa.

Abaixo temos os resultados variando-se os graus entre 1 e 8.

```
Type CSV file name =>data2.csv
Type minimum integer for the degree of the polynomial =>1
Type maximum integer for the degree of the polynomial =>8

Degree      rho           p-value           r         p-value
1          -0.009         9.699e-01       0.034      0.8868
2           0.498         2.553e-02       0.375      0.1038
3           0.677         1.051e-03       0.588      0.006408
4           0.820         9.807e-06       0.843      3.1e-06
5           0.905         4.122e-08       0.909      3.002e-08
6           0.977         1.323e-13       0.967      3.424e-12
7           0.971         1.086e-12       0.968      3.327e-12
8           0.974         4.037e-13       0.968      2.887e-12

Maximum rho is 0.977  for degree  6
```

Abaixo vemos o gráfico gerado pelo programa *scatter_plot6.py*.



Neste programa, iniciamos com a importação das bibliotecas *numpy*, *matplotlib.pyplot* e *scipy.stats.* Em seguida definimos uma função *(read_input())* que lê o nome do arquivo de entrada (planilha com os dados) e os graus dos polinômios. Os dados lidos retornam depois de função *read_input()* ser chamada.

```
# Import packages
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import pearsonr, spearmanr

# Define a function to read input data
def read_input():
    """Function to read input data"""

    # Read input data
    file = input("Type CSV file name =>")
    min_d = int(input("Type minimum integer degree of the polynomial =>"))
    max_d = int(input("Type maximum integer degree of the polynomial =>"))

    return file, min_d, max_d
```

Na função *read_csv()* lemos o arquivo CSV com *genfromtxt()* do NumPy e selecionamos as colunas. O parâmetro desta função é o nome do arquivo CSV a ser lido. Os valores lidos retornam como arrays (*x_col,y_col*).

```
def read_csv(csv_in):
    """Function to read a CSV file and return x,y arrays"""

    # Read CSV file
    my_csv = np.genfromtxt (csv_in, delimiter=",", skip_header = 1)

    # Get each column from a CSV file
    x_col = my_csv[:,0]
    y_col = my_csv[:,1]

    # Return x_col,y_col
    return x_col,y_col
```

Na função *reg_model()* temos como parâmetros os arrays *x_in*, *y_in* e o grau do polinômio *(degree)* que será usado para gerarmos o modelo de regressão. A função calcula os coeficientes de correlação e retorna estes valores.

```
def reg_model(x_in,y_in,degree):
    """Function to generate a regression model and perform statistical
     analysis"""

    # Least-squares polynomial fitting (degree must be an integer)
    z = np.polyfit(x_in,y_in,degree)    # Get the equation coefficients
    p = np.poly1d(z)                    # Generate polynomial equation

    # Calculate Spearman's rank-order correlation coefficient
    rho, pvalue_rho = spearmanr(y_in,p(x_in))

    # Calculate Pearson correlation coefficient
    r, pvalue_r = pearsonr(y_in,p(x_in))

    # Return correlation coefficients
    return rho, pvalue_rho, r, pvalue_r
```

A função *find_max()* tem como parâmetro uma lista de números. Desta lista a função identifica o seu valor máximo e o seu índice. A função retorna esses valores.

```
def find_max(list_in):
    """Function to find the index of the maximum value of a list"""

    # Find maximum
    my_array = np.array(list_in)
    max_number = np.max(my_array)
    index = np.argmax(my_array)

    # Return max_number, index
    return max_number, index
```

Agora geramos os modelos para a faixa de inteiros fornecidos pelo usuário. A função *test_models()* tem como parâmetros os arrays *(x1,y1)* e os limites inteiros dos graus dos polinômios a serem testados (*min_deg_in*, *max_deg_in*). A função retorna o grau do polinômio que gerou o modelo com melhor correlação.

```
def test_models(x1,y1,min_deg_in,max_deg_in):
    """Function to evaluate the predictive performance regression models"""
    # Set up empty lists
    rho_list = []
    deg_list = []
    # Print headers
    print("\nDegree \t   rho       \tp-value      \tr\t\tp-value")
    # Looping through degrees to find the highest rho
    for i in range(min_deg_in,max_deg_in+1):
        # Call reg_model() function
        rho_in,pvalue1_in,r_in,pvalue2_in = reg_model(x1,y1,i)
        # Show result for each iteration
        print("%d\t\t%8.3f\t\t%.3e \t%8.3f\t%.4g"%(i,rho_in,pvalue1_in,r_in,pvalue2_in))
        # Append rho
        rho_list.append(rho_in)
        # Append i
        deg_list.append(i)
    # Find maximum
    max_rho, ind_max = find_max(rho_list)
    pol_degree = deg_list[ind_max]
    # Show information about the highest correlation result
    print("\nMaximum rho is %5.3f"%max_rho," for degree ",pol_degree)
    # Return pol_degree
    return pol_degree
```

A função tem um loop *for* onde são gerados modelos de regressão para a faixa de graus definida pelo usuário. No loop é chamada a função *reg_model()* que gera os modelos de regressão. Em Python podemos chamar uma função a partir de outra função. Adiante chamamos a função *find_max()*.

```
def test_models(x1,y1,min_deg_in,max_deg_in):
    """Function to evaluate the predictive performance regression models"""
    # Set up empty lists
    rho_list = []
    deg_list = []
    # Print headers
    print("\nDegree \t   rho       \tp-value      \tr\t\tp-value")
    # Looping through degrees to find the highest rho
    for i in range(min_deg_in,max_deg_in+1):
        # Call reg_model() function
        rho_in,pvalue1_in,r_in,pvalue2_in = reg_model(x1,y1,i)
        # Show result for each iteration
        print("%d\t\t%8.3f\t\t%.3e \t%8.3f\t%.4g"%(i,rho_in,pvalue1_in,r_in,pvalue2_in))
        # Append rho
        rho_list.append(rho_in)
        # Append i
        deg_list.append(i)
    # Find maximum
    max_rho, ind_max = find_max(rho_list)
    pol_degree = deg_list[ind_max]
    # Show information about the highest correlation result
    print("\nMaximum rho is %5.3f"%max_rho," for degree ",pol_degree)
    # Return pol_degree
    return pol_degree
```

Na função abaixo geramos o gráfico de dispersão e a curva do melhor modelo. A função *gen_scatter_plot()* tem como parâmetros os arrays de dados (*x_plt,y_plt*), o grau do melhor polinômio (*degree_pol_in*) e o nome do arquivo de dados *(data_file).*

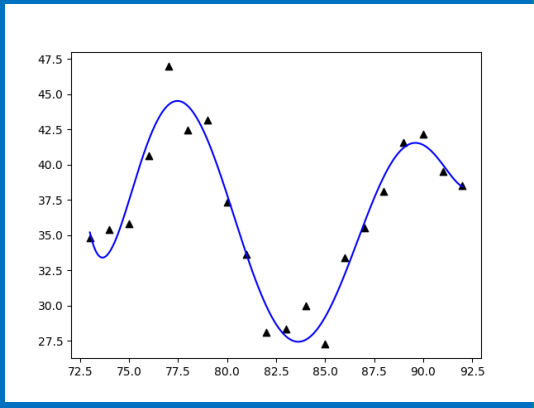
```
def gen_scatter_plot(x_plt,y_plt,degree_pol_in,data_file):
    """Function to generate scatter plot"""
    # Get the minimum and maximum values of x
    x_min = np.min(x_plt)
    x_max = np.max(x_plt)
    # Set up an axis for the least-square polynomial
    x_array = np.linspace(x_min,x_max,1000)
    # Generate plot
    plt.scatter(x_plt,y_plt,marker= "^",color = "black")
    # Recalculate polynomial for the highest correlation
    z_max = np.polyfit(x_plt,y_plt,degree_pol_in) # Get the equation coefficients
    p_max = np.poly1d(z_max)                      # Generate polynomial equation
    # Generate plot
    plt.plot(x_array,p_max(x_array),color = "blue")
    # Show plot
    plt.show()
    # Some editing of the data_file string
    plot_file_out = data_file[:len(data_file)-4]
    # Save plot on png file
    plt.savefig(plot_file_out+'_degree_pol_'+str(degree_pol_in)+'.png')
```

Agora temos a função *main()* e em seguida sua chamada. Na função *main()* temos a chamada em sequência das seguintes funções: *read_input()*, *read_csv()*, *test_models()* e *gen_scatter_plot().* A última linha de comando faz a chamada da função *main()*. Sem esta última linha o programa são seria executado.

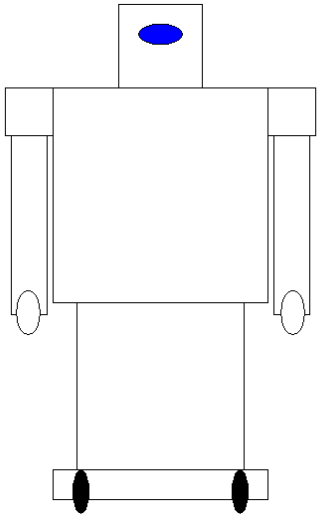
```
def main():

    # Read input data
    file_in, min_deg, max_deg = read_input()

    # Call read_csv() function
    x,y = read_csv(file_in)

    # Call test_models() function
    best_degree = test_models(x,y,min_deg,max_deg)

    # Call gen_scatter_plot() function
    gen_scatter_plot(x,y,best_degree,file_in)

# Call main() function
main()
```

Como já visto, temos os resultados variando-se os graus do polinômio de regressão entre 1 e 8. O maior coeficiente de correlação de Speraman foi obtido para o polinômio de grau 6.

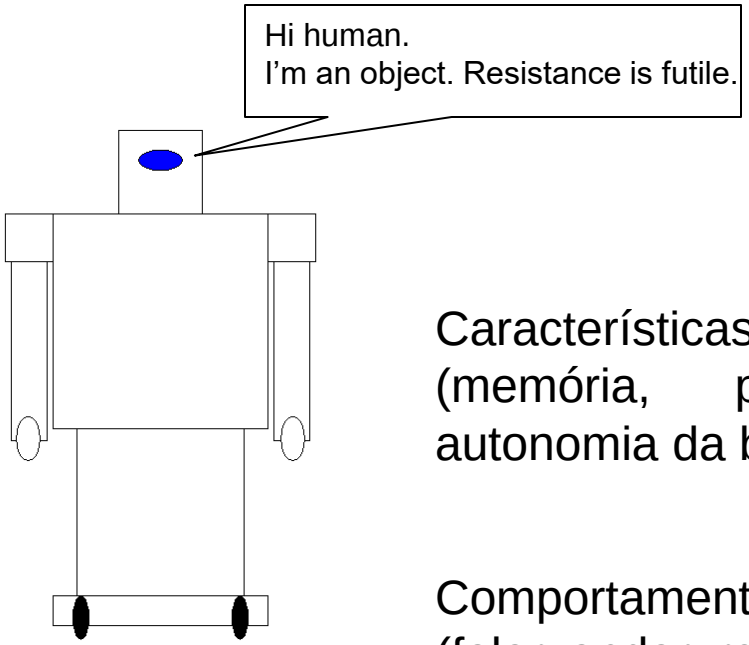
```
Type CSV file name =>data2.csv
Type minimum integer for the degree of the polynomial =>1
Type maximum integer for the degree of the polynomial =>8

Degree      rho           p-value          r          p-value
1          -0.009         9.699e-01       0.034      0.8868
2           0.498         2.553e-02       0.375      0.1038
3           0.677         1.051e-03       0.588      0.006408
4           0.820         9.807e-06       0.843      3.1e-06
5           0.905         4.122e-08       0.909      3.002e-08
6           0.977         1.323e-13       0.967      3.424e-12
7           0.971         1.086e-12       0.968      3.327e-12
8           0.974         4.037e-13       0.968      2.887e-12

Maximum rho is 0.977  for degree  6
```



Hoje veremos o uso da abordagem de **programação orientada a objeto** (*object-oriented programming (OOP)*).
A aplicação da *OOP* baseia-se no conceito de **objeto**. O uso desta abordagem permite a representação de objetos reais como objetos do programa. Em *OOP* montamos **classes** que são usadas para criarmos objetos.



Objetos reais têm características que chamamos em *OOP* de **atributos**. Esses objetos reais podem apresentar comportamentos, que em *OOP* chamamos de **métodos**. Por exemplo, um programa para simular um robô pode ter capacidade memória, potência do motor de movimento, autonomia da bateria como **atributos**. Os **métodos** podem ser falar, andar, responder perguntas etc...
Vamos discutir os principais conceitos de OOP a partir do estudo de programas simples.



Características → Atributos
(memória, potência do motor, autonomia da bateria etc...)

Comportamentos → Métodos
(falar, andar, responder perguntas etc...)

Abaixo temos o código do programa *robot1.py*. O programa inicia com a definição da **classe** *Robot*. Usamos o comando *class* seguido do nome da classe. Não é obrigatório, mas é comum usarmos nomes de classe que iniciam com letra maiúscula.
Após o nome da classe, colocamos a palavra reservada *object.* No presente caso, a classe *Robot* é baseada no objeto, um tipo fundamental interno de dado da linguagem Python. Uma classe pode ser baseada numa outra classe previamente definida.

```
# Program to illustrate the basic concepts of class, objects, and methods
class Robot(object):
     """Simple class for a robot"""
     # Constructor method
     def __init__(self, name):
          print("\nA new robot has been built!")
          self.name = name
     # Method talk()
     def talk(self):
          print("\nHi. I'm an instance of class Robot.")

def main():
     # Instantiating an object of the Robot class and assigns it to the variable r2d2
     r2d2 = Robot("R2D2")
     # Invoking the method talk()
     r2d2.talk()
main()
```

Em seguida definimos a *docstring* com uma breve descrição da classe. A *docstring* é montada de forma similar ao que foi visto para funções.
Agora temos a definição do método construtor. Este método é invocado toda vez que é criado um objeto da classe *Robot*. Veja que a definição do método construtor ocorre com o comando *def __init__()*. Usamos o comando *def* para definição de métodos de uma classe. O método construtor é muito útil, sendo frequente seu uso para cada classe. Um dos usos do método construtor é na atribuição de valores iniciais a atributos de um objeto. No programa *robot1.py* não usamos esse recurso. Aqui usamos o método construtor para mostrarmos uma mensagem na tela.

```
# Program to illustrate the basic concepts of class, objects, and methods
class Robot(object):
     """Simple class for a robot"""
     # Constructor method
     def __init__(self, name):
          print("\nA new robot has been built!")
          self.name = name
     # Method talk()
     def talk(self):
          print("\nHi. I'm an instance of class Robot.")

def main():
     # Instantiating an object of the Robot class and assigns it to the variable r2d2
     r2d2 = Robot("R2D2")
     # Invoking the method talk()
     r2d2.talk()
main()
```

Podemos ter os atributos de um objeto automaticamente criados e inicializados logo depois que sua criação. A classe *Robot* tem um método construtor que cria e inicializa o atributo *name.* O método construtor mostra a mensagem "*\nA new robot has been built!*". Em seguida é criado um novo atributo, *name,* para o novo objeto que usa o valor do parâmetro *name* como valor. Assim, a linha de código no programa principal, *r2d2 = Robot("R2D2"),* tem como resultado a criação de um novo objeto da classe *Robot*, com o atributo *name* definido como "R2D2". Por último, o objeto é atribuído à variável *r2d2*. Veja que o programa principal é definido como uma função denominada *main()*, assim temos que chamá-la para a sua execução.

```
# Program to illustrate the basic concepts of class, objects, and methods
class Robot(object):
     """Simple class for a robot"""
     # Constructor method
     def __init__(self, name):
          print("\nA new robot has been built!")
          self.name = name
     # Method talk()
     def talk(self):
          print("\nHi. I'm an instance of class Robot.")

def main():
     # Instantiating an object of the Robot class and assigns it to the variable r2d2
     r2d2 = Robot("R2D2")
     # Invoking the method talk()
     r2d2.talk()
main()
```

A palavra reservada *self* tem como função receber a referência a um objeto invocando um método. Assim, por meio do *self*, um método pode acessar o objeto invocando ele e acessar os atributos e métodos do objeto.
No método construtor, o parâmetro *self* recebe uma referência ao novo objeto da classe *Robot,* enquanto o parâmetro *name* recebe a string *"R2D2"*.
A linha de código *self.name = name* cria o atributo *name* para o objeto e define o seu valor com o valor do parâmetro *name,* que é *"R2D2"*.

```
# Program to illustrate the basic concepts of class, objects, and methods
class Robot(object):
     """Simple class for a robot"""
     # Constructor method
     def __init__(self, name):
          print("\nA new robot has been built!")
          self.name = name
     # Method talk()
     def talk(self):
          print("\nHi. I'm an instance of class Robot.")

def main():
     # Instantiating an object of the Robot class and assigns it to the variable r2d2
     r2d2 = Robot("R2D2")
     # Invoking the method talk()
     r2d2.talk()
main()
```

O método *talk()* é definido usando-se o comando *def.* Veja que temos *self* como parâmetro do método, com a função já descrita. O método *talk()* é simplesmente uma função *print().* Veja que podemos pensar nos métodos como funções associadas a um objeto. Os métodos de uma dada classe têm que necessariamente ter o parâmetro *self*. Que no método *talk()* não é usado, mas tem que estar presente.

```
# Program to illustrate the basic concepts of class, objects, and methods
class Robot(object):
     """Simple class for a robot"""
     # Constructor method
     def __init__(self, name):
          print("\nA new robot has been built!")
          self.name = name
     # Method talk()
     def talk(self):
          print("\nHi. I'm an instance of class Robot.")

def main():
     # Instantiating an object of the Robot class and assigns it to the variable r2d2
     r2d2 = Robot("R2D2")
     # Invoking the method talk()
     r2d2.talk()
main()
```

No programa principal, a linha de código *r2d2 = Robot("R2D2")* cria um novo objeto da classe *Robot* e atribui este objeto à variável *r2d2.*
Você pode atribuir o novo objeto a qualquer variável, desde que siga as regras de variáveis do Python.

```
# Program to illustrate the basic concepts of class, objects, and methods
class Robot(object):
     """Simple class for a robot"""
     # Constructor method
     def __init__(self, name):
          print("\nA new robot has been built!")
          self.name = name
     # Method talk()
     def talk(self):
          print("\nHi. I'm an instance of class Robot.")

def main():
     # Instantiating an object of the Robot class and assigns it to the variable r2d2
     r2d2 = Robot("R2D2")
     # Invoking the method talk()
     r2d2.talk()
main()
```

Por último, o programa principal invoca o método *talk()*. Usamos a notação *dot* (.) como vista para métodos de bibliotecas do Python.
A linha de código *r2d2.talk()* simplesmente invoca o método *talk()* de um objeto da classe *Robot*, que foi previamente atribuído à variável *r2d2.*

```
# Program to illustrate the basic concepts of class, objects, and methods
class Robot(object):
     """Simple class for a robot"""
     # Constructor method
     def __init__(self, name):
          print("\nA new robot has been built!")
          self.name = name
     # Method talk()
     def talk(self):
          print("\nHi. I'm an instance of class Robot.")

def main():
     # Instantiating an object of the Robot class and assigns it to the variable r2d2
     r2d2 = Robot("R2D2")
     # Invoking the method talk()
     r2d2.talk()
main()
```

Abaixo temos o resultado de rodar o programa *robot1.py*.

```
A new robot has been built!


Hi. I'm an instance of class Robot.
```

No programa *robot2.py* modificamos o método *talk()* de forma que podemos mostrar o que foi atribuído ao parâmetro *name.* A modificação do método está em destaque em vermelho.

```
# Program to illustrate the basic concepts of class, objects, and methods

class Robot(object):
     """Simple class for a robot"""
     # Constructor method
     def __init__(self, name):
          print("\nA new robot has been built!")
          self.name = name
     # Method talk()
     def talk(self):
          print("\nHi. I'm an instance of class Robot. My name is ",self.name)
def main():
     # Instantiating an object of the Robot class and assigns it to the variable r2d2
     r2d2 = Robot("R2D2")
     # Instantiating an object of the Robot class and assigns it to the variable c3p0
     c3p0 = Robot("C3P0")
     # Invoking the method talk()
     r2d2.talk()
     # Invoking the method talk()
     c3p0.talk()

main()
```

Para criarmos múltiplos objetos da classe *Robot*, temos que ter uma linha de código para cada novo objeto a ser criado. Obviamente usamos como argumento strings distintas, como indicado no código abaixo.
Após a criação dos objetos, podemos invocar o método *talk()* para cada objeto criado.

```
# Program to illustrate the basic concepts of class, objects, and methods

class Robot(object):
     """Simple class for a robot"""
     # Constructor method
     def __init__(self, name):
          print("\nA new robot has been built!")
          self.name = name
     # Method talk()
     def talk(self):
          print("\nHi. I'm an instance of class Robot. My name is ",self.name)
def main():
     # Instantiating an object of the Robot class and assigns it to the variable r2d2
     r2d2 = Robot("R2D2")
     # Instantiating an object of the Robot class and assigns it to the variable c3p0
     c3p0 = Robot("C3P0")
     # Invoking the method talk()
     r2d2.talk()
     # Invoking the method talk()
     c3p0.talk()

main()
```

Abaixo temos o resultado de rodar o programa *robot2.py*.

```
A new robot has been built!

A new robot has been built!

Hi. I'm an instance of class Robot. My name is R2D2

Hi. I'm an instance of class Robot. My name is C3P0
```

Por meio do método *__str__()*, podemos criar uma string que será mostrada toda vez que usarmos a função *print()* para um objeto da classe. No exemplo em vermelho abaixo, temos a definição da string. Veja que usarmos o comando *return*.

```
# Program to illustrate the basic concepts of class, objects, and methods
class Robot(object):
      """Simple class for a robot"""
      # Constructor method
      def __init__(self, name):
            print("\nA new robot has been built!")
            self.name = name
      def __str__(self):
            rep = "Robot object\n"
            rep += "name: " + self.name + "\n"
            return rep
      # Method talk()
      def talk(self):
            print("\nHi. I'm an instance of class Robot. My name is ",self.name)
def main():
      # Instantiating an object of the Robot class and assigns it to the variable r2d2
      r2d2 = Robot("R2D2")
      # Instantiating an object of the Robot class and assigns it to the variable c3p0
      c3p0 = Robot("C3P0")
      # Invoking the method talk()
      r2d2.talk()
      # Invoking the method talk()
      c3p0.talk()
      print("\nPrinting r2d2:")
      print(r2d2)
      print("\nDirectly accessing r2d2.name:")
      print(r2d2.name)
main()
```

No programa principal, a linha de código *print(r2d2)* garante que a string definida no método *__str__()* seja mostrada na tela.

```
# Program to illustrate the basic concepts of class, objects, and methods
class Robot(object):
      """Simple class for a robot"""
      # Constructor method
      def __init__(self, name):
            print("\nA new robot has been built!")
            self.name = name
      def __str__(self):
            rep = "Robot object\n"
            rep += "name: " + self.name + "\n"
            return rep
      # Method talk()
      def talk(self):
            print("\nHi. I'm an instance of class Robot. My name is ",self.name)
def main():
      # Instantiating an object of the Robot class and assigns it to the variable r2d2
      r2d2 = Robot("R2D2")
      # Instantiating an object of the Robot class and assigns it to the variable c3p0
      c3p0 = Robot("C3P0")
      # Invoking the method talk()
      r2d2.talk()
      # Invoking the method talk()
      c3p0.talk()
      print("\nPrinting r2d2:")
      print(r2d2)
      print("\nDirectly accessing r2d2.name:")
      print(r2d2.name)
main()
```

Para acessarmos um atributo da classe *Robot*, fora da definição da classe, podemos usar a notação *dot*, como indicada na linha de código *print(r2d2.name)*, onde acessamos o valor atribuído ao *name*.

```
# Program to illustrate the basic concepts of class, objects, and methods
class Robot(object):
      """Simple class for a robot"""
      # Constructor method
      def __init__(self, name):
            print("\nA new robot has been built!")
            self.name = name
      def __str__(self):
            rep = "Robot object\n"
            rep += "name: " + self.name + "\n"
            return rep
      # Method talk()
      def talk(self):
            print("\nHi. I'm an instance of class Robot. My name is ",self.name)
def main():
      # Instantiating an object of the Robot class and assigns it to the variable r2d2
      r2d2 = Robot("R2D2")
      # Instantiating an object of the Robot class and assigns it to the variable c3p0
      c3p0 = Robot("C3P0")
      # Invoking the method talk()
      r2d2.talk()
      # Invoking the method talk()
      c3p0.talk()
      print("\nPrinting r2d2:")
      print(r2d2)
      print("\nDirectly accessing r2d2.name:")
      print(r2d2.name)
main()
```

Abaixo temos o resultado de rodar o programa *robot3.py*.

```
A new robot has been built!
A new robot has been built!
Hi. I'm an instance of class Robot. My name is R2D2
Hi. I'm an instance of class Robot. My name is C3P0
Printing r2d2:
Robot object
name: R2D2
Directly accessing r2d2.name:
R2D2
```

Sequência de Fibonacci (Versão 2)
Programa: *fibonacci2.py*

Resumo
Programa para gerar a sequência de Fibonacci. O número de elementos da sequência é dado pelo usuário. Use a abordagem de programação orientada a objetos para desenvolver o código.

Scatter Plot (Versão 7)

Programa: *scatter_plot7.py*

Resumo
Programa para testar diferentes modelos gerados por regressão linear. O usuário fornece os graus do polinômio menor e maior a serem testados. O programa testa todos os polinômios de graus indicados pelo usuário e mostra o desempenho de cada modelo na tela. Por exemplo, se indicados os graus 1 e 8, o programa gera os polinômios de graus entre 1 e 8 e mostra o desempenho de cada modelo. O programa seleciona o polinômio que gerou o modelo com maior coeficiente de correlação de Spearman (*rho*) e faz o gráfico somente deste melhor. Teste o programa para uma faixa entre 1 e 8. Use o arquivo *data2.csv*. Use a abordagem de programação orientada a objetos para desenvolver o código.

-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.
-HAL, Tim, STACEY, J-P. **Python 3 for Absolute Beginners.** Springer-Verlag New York, 2009. 295 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.