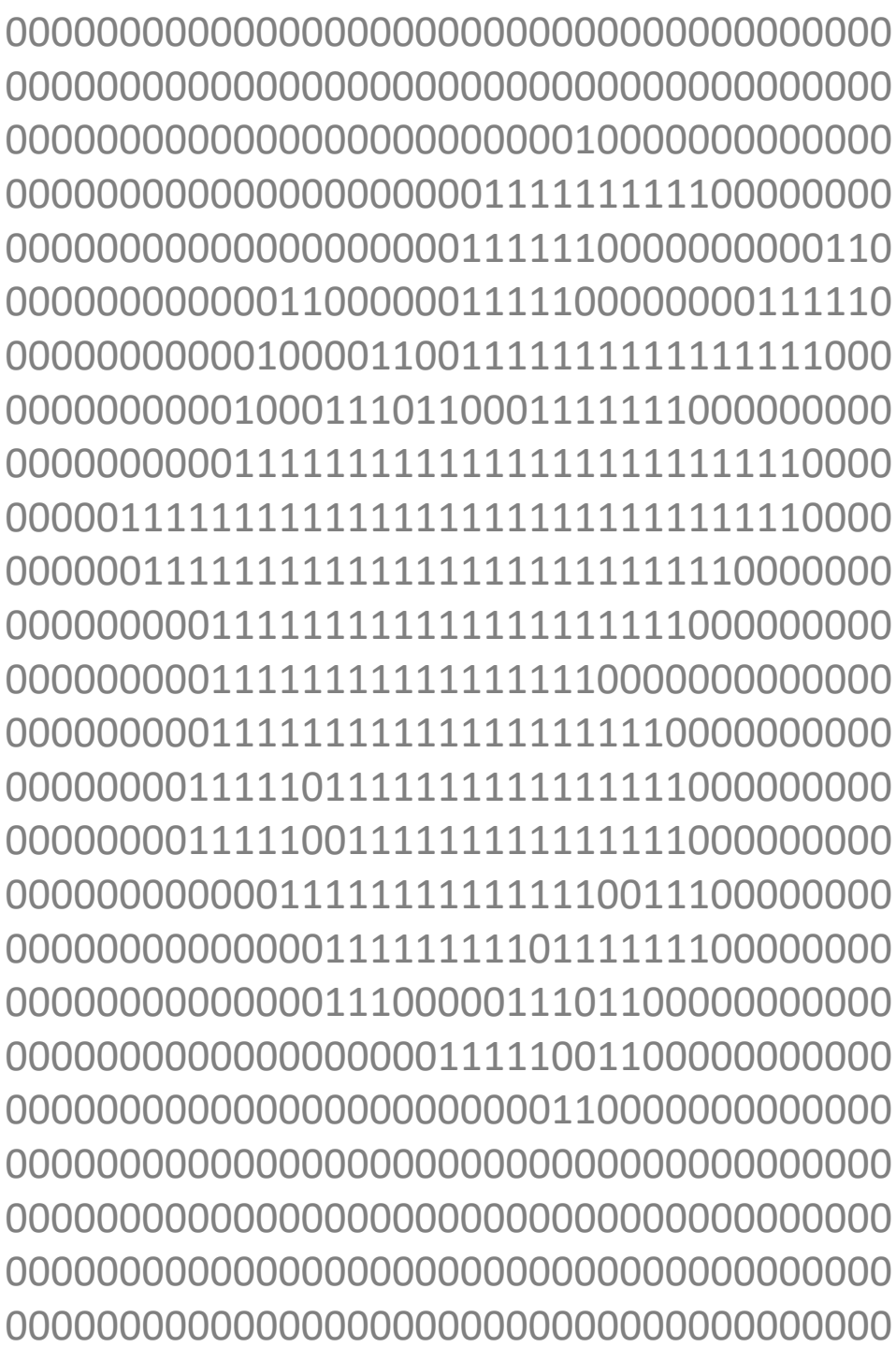


azevedolab.net

Introdução à Física Computacional

Aula 06

Scatter Plot (Versão 4)

Programa: *scatter_plot4.py*

Resumo
Programa para gerar o gráfico de espalhamento para um arquivo CSV cujo o nome é dado pelo usuário. O grau do polinômio é um inteiro fornecido pelo usuário. Teste o programa para o arquivo *data2.csv*.

Para a implementação do programa *scatter_plot4.py*, vamos considerar alguns recursos adicionais da biblioteca *Matplotlib*. Discutiremos esses recursos conforme explicamos o código. As primeiras linhas do programa estão indicadas abaixo. Inicialmente importamos as bibliotecas *NumPy* e *Matplotlib.*

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

# Read input data
file_in = input("Type CSV file name =>")
degree_pol = int(input("Type an integer for the degree of the polynomial =>"))

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

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

Na sequência lemos com *input()* o nome do arquivo CSV e o grau do polinômio. Depois usamos o método *genfromtxt()* da biblioteca *NumPy* para lermos o arquivo CSV. Omitimos a primeira linha com a opção *skip_header = 1* e fixamos o delimitador como vírgula a partir do parâmetro *delimiter=“,”*.

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

# Read input data
file_in = input("Type CSV file name =>")
degree_pol = int(input("Type an integer for the degree of the polynomial =>"))

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

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

Como o conteúdo do arquivo CSV foi atribuído à variável *my_csv*, selecionamos a primeira coluna com a linha `x = my_csv[:,0]` e a segunda com `y = my_csv[:,1]`. Às variáveis *x* e *y* foram atribuídos *arrays* com o conteúdos das colunas lidas do arquivo CSV.

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

# Read input data
file_in = input("Type CSV file name =>")
degree_pol = int(input("Type an integer for the degree of the polynomial =>"))

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

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

Agora identificamos os valores mínimo e máximo do *array x* com os métodos *np.min()* e *np.max()* da biblioteca *NumPy.* Em seguida usamos esses valores para definirmos o *array x1*, que será usado posteriormente para gerarmos o gráfico do modelo obtido por regressão.

```
# Get the minimum and maximum values of x
x_min = np.min(x)
x_max = np.max(x)

# Set up an axis for the least-square polynomial
x1 = np.linspace(x_min,x_max,1000)

# Generate plot
plt.scatter(x,y,marker= "^",color = "black", s = 50)

# Least-squares polynomial fitting (degree must be an integer)
z = np.polyfit(x,y,degree_pol) # Get the equation coefficients
p = np.poly1d(z) # Generate polynomial equation
print("Best fit polynomial equation: \n",p)
```

Usamos o comando *plt.scatter()* para gerarmos gráfico de espalhamento dos dados lidos do arquivo CSV. Veja que usamos o parâmetro *marker*=“^” para definirmos que os pontos do gráfico têm o formato de triângulo. No link https://matplotlib.org/3.1.0/api/markers_api.html#module-matplotlib.markers você tem a lista completa de formatos de pontos.

```
# Get the minimum and maximum values of x
x_min = np.min(x)
x_max = np.max(x)


# Set up an axis for the least-square polynomial
x1 = np.linspace(x_min,x_max,1000)


# Generate plot
plt.scatter(x,y,marker= "^",color = "black", s = 50)


# Least-squares polynomial fitting (degree must be an integer)
z = np.polyfit(x,y,degree_pol) # Get the equation coefficients
p = np.poly1d(z) # Generate polynomial equation
print("Best fit polynomial equation: \n",p)
```

Ainda no comando *plt.scatter()*, definimos a cor dos pontos do gráfico de espalhamento com o parâmetro *color = "black".* Também definimos o tamanho em pixels do ponto com o parâmetro *s = 50*. Há outros parâmetros que podem ser explorados na definição do gráfico de espalhamento, informações adicionais no link: https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.scatter.html.

```
# Get the minimum and maximum values of x
x_min = np.min(x)
x_max = np.max(x)


# Set up an axis for the least-square polynomial
x1 = np.linspace(x_min,x_max,1000)


# Generate plot
plt.scatter(x,y,marker= "^",color = "black", s = 50)


# Least-squares polynomial fitting (degree must be an integer)
z = np.polyfit(x,y,degree_pol) # Get the equation coefficients
p = np.poly1d(z) # Generate polynomial equation
print("Best fit polynomial equation: \n",p)
```

Na sequência geramos o modelo por regressão disponível no método *np.polyfit()*. No *np.polyfit()* usamos os *arrays x* e *y* e um inteiro atribuído à varável *degree_pol*. Os coeficientes do modelo de regressão obtidos pelo *np.polyfit()* são retornados na forma de *array* e atribuído à variável *z.* Depois geramos um polinômio com *np.poly1d()*. A equação gerada é atribuída à variável *p* e os resultados mostrados na tela.

```
# Get the minimum and maximum values of x
x_min = np.min(x)
x_max = np.max(x)


# Set up an axis for the least-square polynomial
x1 = np.linspace(x_min,x_max,1000)


# Generate plot
plt.scatter(x,y,marker= "^",color = "black", s = 50)


# Least-squares polynomial fitting (degree must be an integer)
z = np.polyfit(x,y,degree_pol) # Get the equation coefficients
p = np.poly1d(z) # Generate polynomial equation
print("Best fit polynomial equation: \n",p)
```

Agora geramos o gráfico do modelo obtido por regressão. Para isto usamos o comando *plt.plot()*, onde o eixos são os *arrays x1* e *p(x1)*. Usamos o *array x1* que cobre a mesma faixa de *x* mas tem mais elementos, o que gera um gráfico com melhor resolução. Selecionamos a cor azul *(color = "blue")* para o gráfico. A linha seguinte mostra o gráfico na tela.

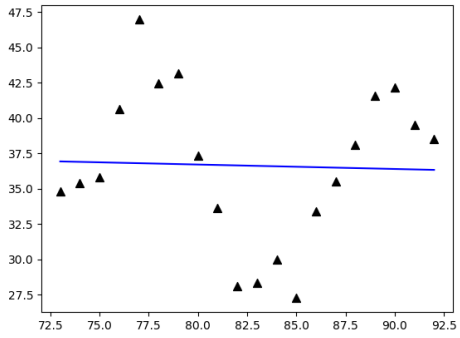
```
# Generate plot
plt.plot(x1,p(x1),color = "blue")



# Show plot
plt.show()



# Some editing of the file_in string
file_out = file_in[:len(file_in)-4]



# Save plot on png file
plt.savefig(file_out+'_degree_pol_'+str(degree_pol)+'.png')
```

Por último, fazemos alguma edição com a string atribuída à variável *file_in* para cortar os 4 caracteres finais e atribuímos a string reduzida à varável *file_out*. Em seguida adicionamos ao nome do arquivo de saída o grau do polinômio e a extensão *.png.*

```
# Generate plot
plt.plot(x1,p(x1),color = "blue")


# Show plot
plt.show()


# Some editing of the file_in string
file_out = file_in[:len(file_in)-4]


# Save plot on png file
plt.savefig(file_out+'_degree_pol_'+str(degree_pol)+'.png')
```

Abaixo temos o resultado para um polinômio de grau 1. Execute o código variando o grau do polinômio entre 1 e 8. Qual polinômio gera o melhor modelo?

```
Type CSV file name =>data2.csv
Type an integer for the degree of the polynomial =>1
Best fit polynomial equation:

-0.03148 x + 39.23
```

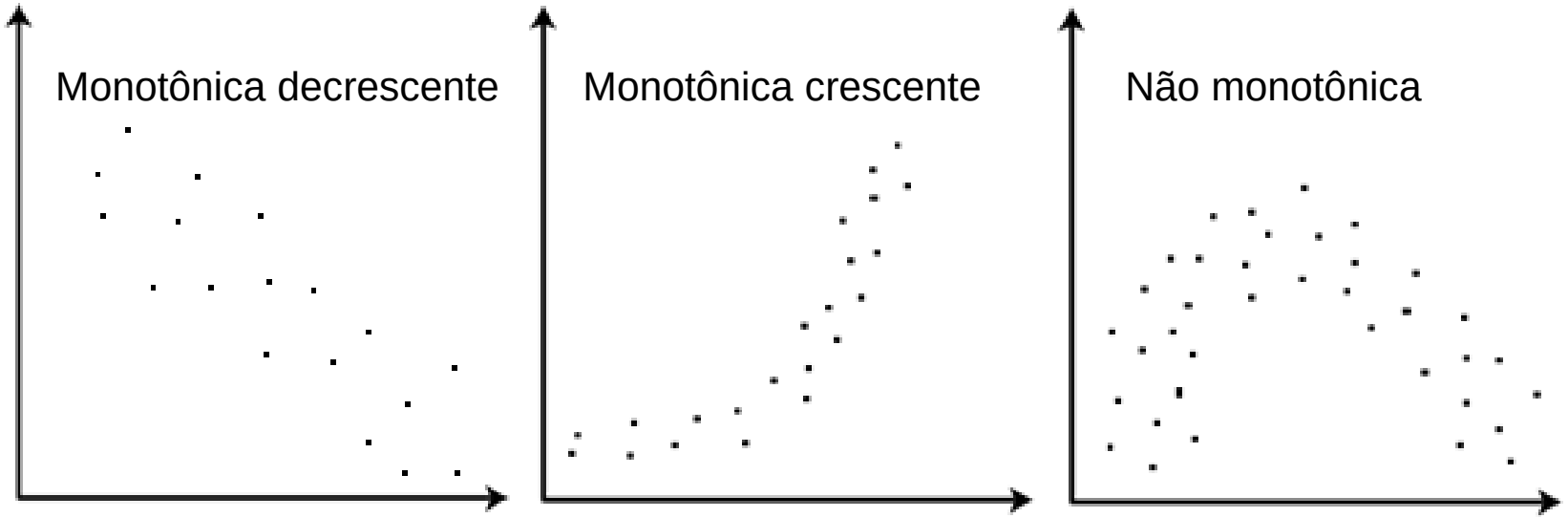


Scatter Plot (Versão 5)

Programa: *scatter_plot5.py*

Resumo
Programa para gerar o gráfico de espalhamento para um 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*. O programa calcula os coeficientes de correlação de Pearson e Spearman, bem como os p-values associados a estes coeficientes.

O **coeficiente de correlação de Spearman** (ρ) é uma medida da força e direção da associação que existe entre duas variáveis medidas em função da ordem dos valores das variáveis. Para o uso do coeficiente de correlação de Spearman é necessário que as duas variáveis possam ser expressas de forma ordinal e que tenham uma relação monotônica, como indicadas nos dois primeiros gráficos abaixo.

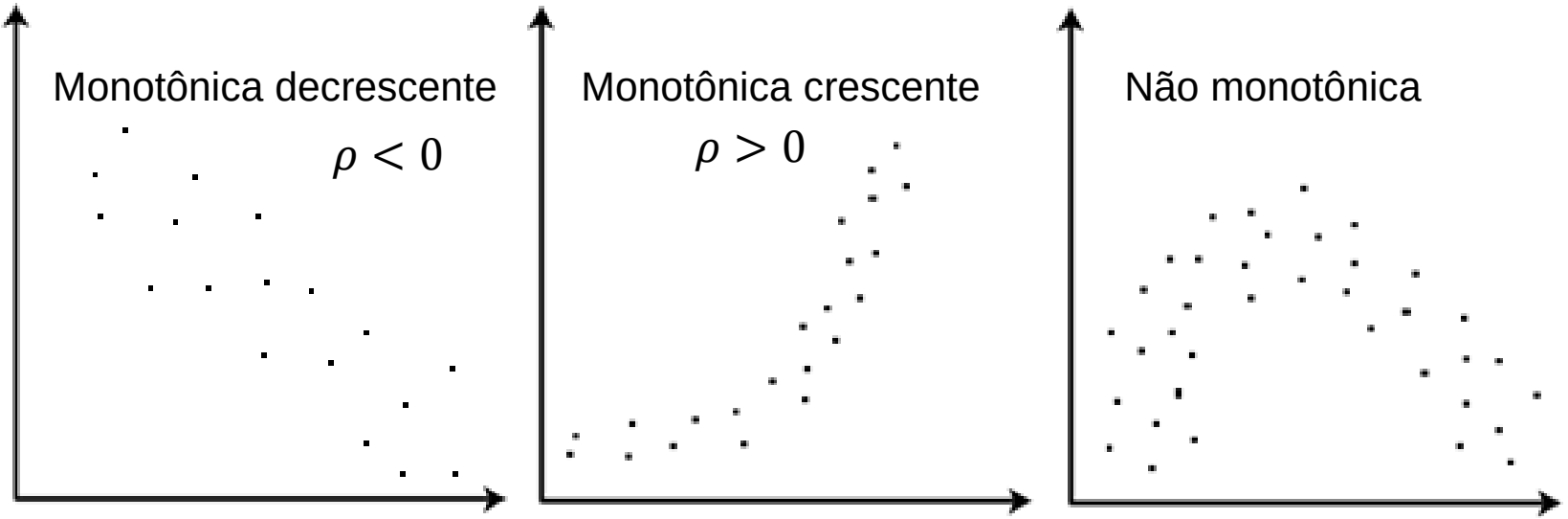


O **coeficiente de correlação de Spearman** (ρ) é dado pela seguinte equação,

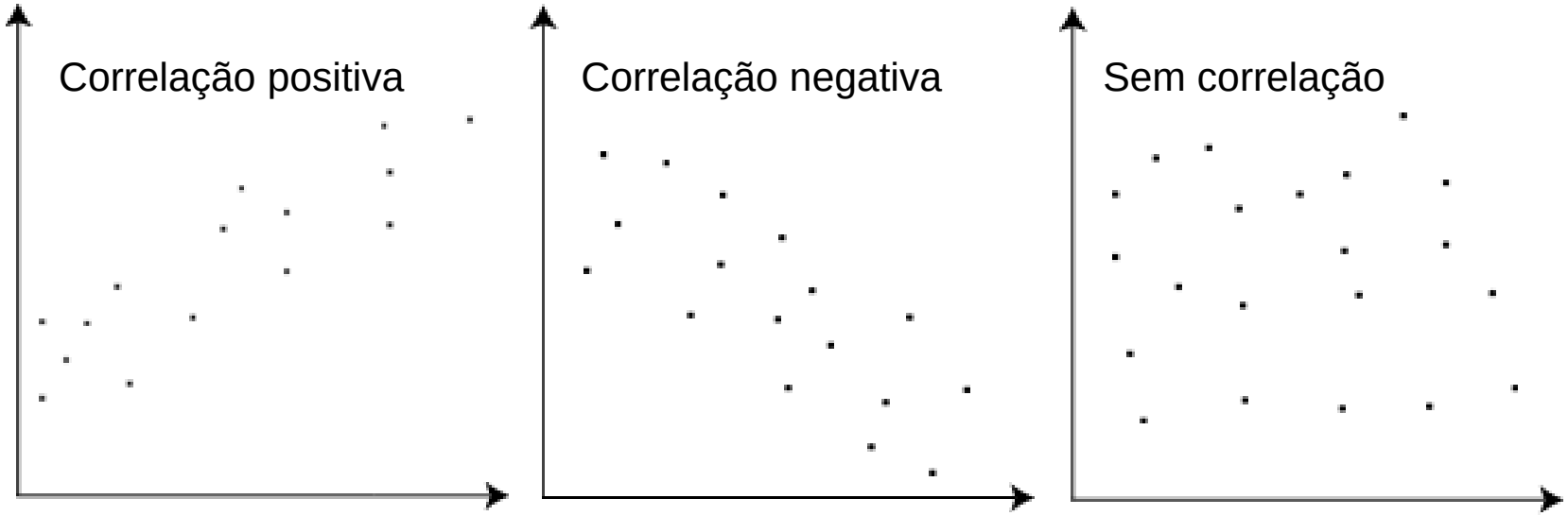
$$\rho = 1 - \frac{6\sum_{i=1}^{n} d_i^2}{n(n-1)}$$

onde d_i é a diferença de posto de cada observação e *n* é o número de observações.

Os valores do coeficiente de correlação de Spearman variam entre -1 e 1. Valores próximos de 1 indicam uma correlação positiva, como no gráfico da relação monotônica crescente ($\rho > 0$). Valores negativos para ρ indicam um gráfico de dispersão com uma relação monotônica decrescente.



O **coeficiente de correlação de Pearson** (*r*) é uma medida da força da associação linear que existe entre duas variáveis. Nos gráficos abaixo temos situações com correlação positiva, negativa e sem correlação.

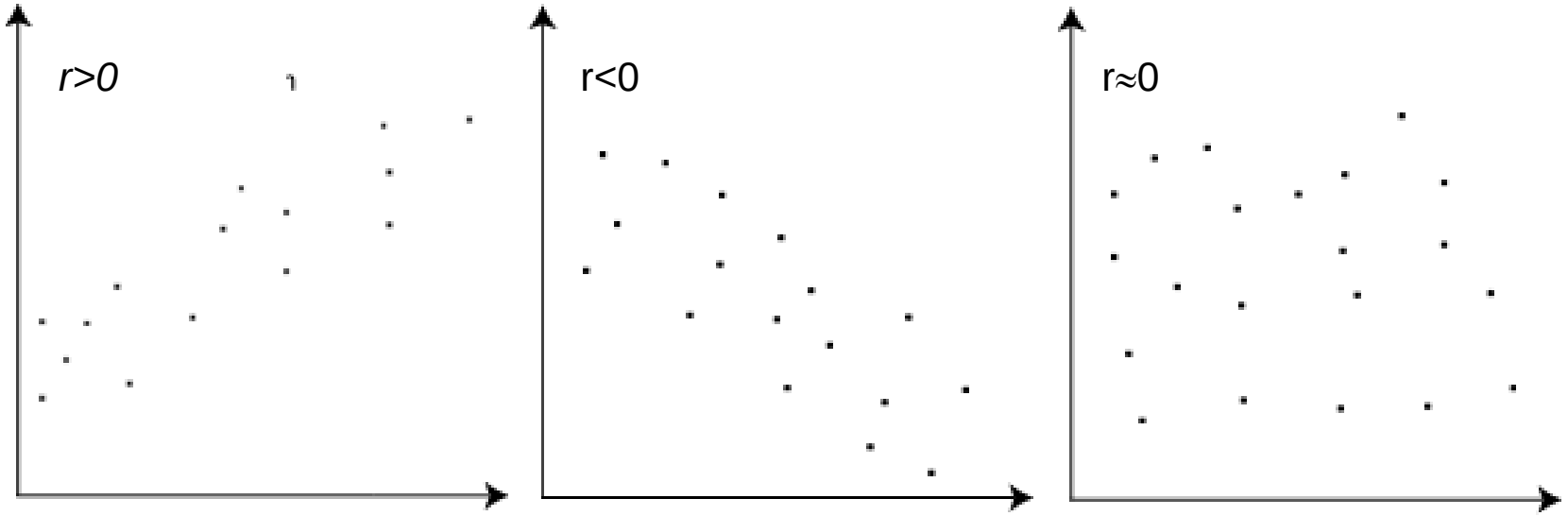


O **coeficiente de correlação de Pearson** (*r*) é dado pela expressão abaixo,

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

onde $x_1,...,x_n$ e $y_1,...,y_n$ são as medidas de ambas as variáveis e $\bar{x}$ e $\bar{y}$ são as médias aritméticas das medidas das variáveis, indicadas abaixo.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$



Como na versão anterior do código, temos que importar as bibliotecas *NumPy* e *Matplotlib*. Além disso, importamos métodos para o cálculo dos coeficientes de Pearson e Spearman da biblioteca *SciPy*. Depois lemos o nome do arquivo CSV, o grau do polinômio e abrimos o arquivo CSV com o método *genfromtxt*() visto no programa anterior. Na sequência definimos que colunas serão usadas para os *arrays x* e *y*.

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


# Read input data
file_in = input("Type CSV file name =>")
degree_pol = int(input("Type an integer for the degree of the polynomial =>"))
# Read CSV file
my_csv = np.genfromtxt (file_in, delimiter=",", skip_header = 1)
# Get each column from a CSV file
x = my_csv[:,0]
y = my_csv[:,1]
```

Nesta parte do código, achamos os valores mínimo e máximo do *array x*, para isto usamos os métodos *np.min*() e *np.max()*. Em seguida, definimos o eixo *x1* com o método *linspace*() da biblioteca *NumPy*. Como na versão anterior, faremos o gráfico de dispersão (ou espalhamento). Nas linhas seguintes, geramos o modelo de regressão com o grau do polinômio definido.

```
# Get the minimum and maximum values of x
x_min = np.min(x)
x_max = np.max(x)

# Set up an axis for the least-square polynomial
x1 = np.linspace(x_min,x_max,1000)

# Generate plot
plt.scatter(x,y,marker= "^",color = "black")

# Least-squares polynomial fitting (degree must be an integer)
z = np.polyfit(x,y,degree_pol)  # Get the equation coefficients
p = np.poly1d(z)                # Generate polynomial equation
print("Best fit polynomial equation: \n",p)
```

Usamos agora os métodos *pearsonr(y,p(x))* e *spearmanr(y,p(x))* da biblioteca *SciPy* para calcularmos os coeficientes de Pearson e Spearman, respectivamente. Veja que para o cálculo dos coeficientes, usamos como parâmetros dos métodos os *arrays y* e *p(x)*, que trazem os valores experimentais (*y*) e gerados pela equação polinomial (*p(x)*).

```
# Calculate Pearson Correlation Coefficient
r, pvalue1 = pearsonr(y,p(x))
print("\nPearson correlation coefficient: %8.3f"%r)
print("p-value: %.4g"%pvalue1)

# Calculate Spearman's Rank-Order Correlation Coefficient
rho, pvalue2 = spearmanr(y,p(x))
print("\nSpearman's rank order correlation coefficient: %8.3f"%rho)
print("p-value: %.4g"%pvalue2)
```

Veja que além dos coeficientes, os métodos *pearsonr(y,p(x))* e *spearmanr(y,p(x))* da biblioteca *SciPy* retornam os valores de p-value. Este valor indica a confiabilidade do modelo testado (equação polinomial), para modelagem de sistemas biológicos espera-se valores abaixo 0,05, ou seja, 5 %.

```
# Calculate Pearson Correlation Coefficient
r, pvalue1 = pearsonr(y,p(x))
print("\nPearson correlation coefficient: %8.3f"%r)
print("p-value: %.4g"%pvalue1)

# Calculate Spearman's Rank-Order Correlation Coefficient
rho, pvalue2 = spearmanr(y,p(x))
print("\nSpearman's rank order correlation coefficient: %8.3f"%rho)
print("p-value: %.4g"%pvalue2)
```

Nesta parte do código mostramos os coeficientes e p-values na tela usando-se um *print*() formatado. O formato *%.4g* é usado para notação científica e fixa em 4 o número de algarismos significativos. O formato *%8.3f* gera um *float* com até 8 algarismos, sendo 3 após o ponto decimal.

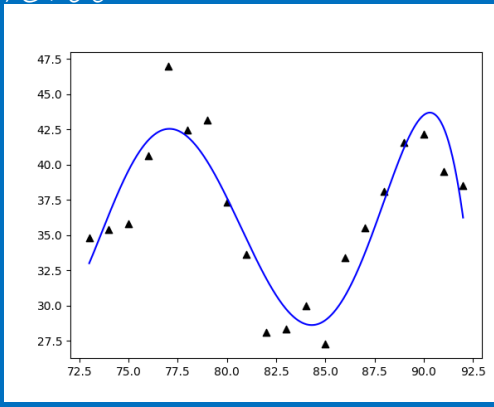
```
# Calculate Pearson Correlation Coefficient
r, pvalue1 = pearsonr(y,p(x))
print("\nPearson correlation coefficient: %8.3f"%r)
print("p-value: %.4g"%pvalue1)

# Calculate Spearman's Rank-Order Correlation Coefficient
rho, pvalue2 = spearmanr(y,p(x))
print("\nSpearman's rank order correlation coefficient: %8.3f"%rho)
print("p-value: %.4g"%pvalue2)
```

Por último geramos o gráfico do polinômio obtido por mínimos quadrados usando-se a mesma sequência de comandos vista para versão anterior.

```
# Generate plot
plt.plot(x1,p(x1),color = "blue")

# Show plot
plt.show()

# Some editing of the file_in string
file_out = file_in[:len(file_in)-4]

# Save plot on png file
plt.savefig(file_out+'_degree_pol_'+str(degree_pol)+'.png')
```

Abaixo temos o resultado para um polinômio de grau 5. Execute o código variando o grau do polinômio entre 1 e 8. Qual polinômio gera o melhor modelo?

```
Type CSV file name =>data2.csv
Type an integer for the degree of the polynomial =>5
Best fit polynomial equation:
             5          4         3        2
-0.0005079 x + 0.2051 x - 33.05 x + 2655 x - 1.064e+05 x + 1.7e+06

Pearson correlation coefficient:    0.909
p-value: 3.002e-08

Spearman's rank order correlation coefficient:    0.905
p-value: 4.122e-08
```



Todos os programas que estudamos até o momento eram sequências de comandos em Python, que começam sua execução do primeiro até o último comando. Essa abordagem de programação funciona bem para programas pequenos, até poucas centenas de linhas de código. Para programas maiores, podemos ter problemas e dificuldades na programação. Para superarmos esse obstáculo, dividimos nosso código em pedaços que executam uma determinada tarefa dentro do programa, e o isolamos do programa principal. Essa 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: *fibonacci1.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 *fibonacci1.py*. Na definição de uma função seguimos o seguinte formato, iniciamos com o comando *def* seguindo 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 interpretador Python 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**. Esta definição não executa os comandos vinculados à ela, simplesmente indica que o 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 relacionados à função estão recuados, como fazemos com blocos de comandos para loops e *if*. 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 no 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. Essa 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. O 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 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 tem que apresentar o valor a ser atribuído ao parâmetro *number_of_elements*, caso contrário vemos uma mensagem de erro. O valor a ser 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 da 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. Essa 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 o conteúdo da 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 dados a serem mostrados 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 varável *my_string*, uma lista atribuída à *my_list* e um dicionário atribuído à *aaMW*. Temos que *my_string*, *my_list* e *aaMW* são argumentos usados nas chamadas das funções, ou seja, seus valores serão atribuídos aos respectivos parâmetros, na chamada de cada função.

```
# Main program
my_string = "Python"
my_list = ["P","y","t","h","o","n"]
# Source for residue molecular weights:
# http://www.matrixscience.com/help/aa_help.html (Accessed on May 8th 2019)
# To calculate the mass of a neutral peptide or protein, sum the residue masses plus
#the masses of the terminating
# groups (e.g. H at the N-terminus and OH at the C-terminus).
# Sets aaMW dictionary
aaMW = {"A": 71.0779,"R": 156.1857,"N": 114.1026,"D": 115.0874,
        "C": 103.1429,"E": 129.114,"Q": 128.1292,"G": 57.0513,
        "H": 137.1393,"I": 113.1576,"L": 113.1576,"K": 128.1723,
        "M": 131.1961,"F": 147.1739,"P": 97.1152,"S": 87.0773,
        "T": 101.1039,"W": 186.2099,"Y": 163.1733,"V": 99.1311
        }
print("\nShowing string:")
show_string(my_string)
print("\nShowing list:")
show_list(my_list)
print("\nShowing dictionary:")
show_dict(aaMW)
```

Abaixo temos o resultado de rodarmos o programa.

```
Showing string:
Python
Showing list:
P
y
t
h
o
n
Showing dictionary:
A 71.0779
C 103.1429
D 115.0874
E 129.114
F 147.1739
G 57.0513
H 137.1393
I 113.1576
K 128.1723
L 113.1576
M 131.1961
N 114.1026
P 97.1152
Q 128.1292
R 156.1857
S 87.0773
T 101.1039
V 99.1311
W 186.2099
Y 163.1733
```

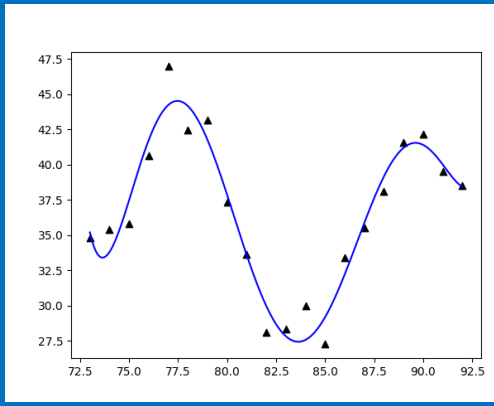
Scatter Plot (Versão 6)

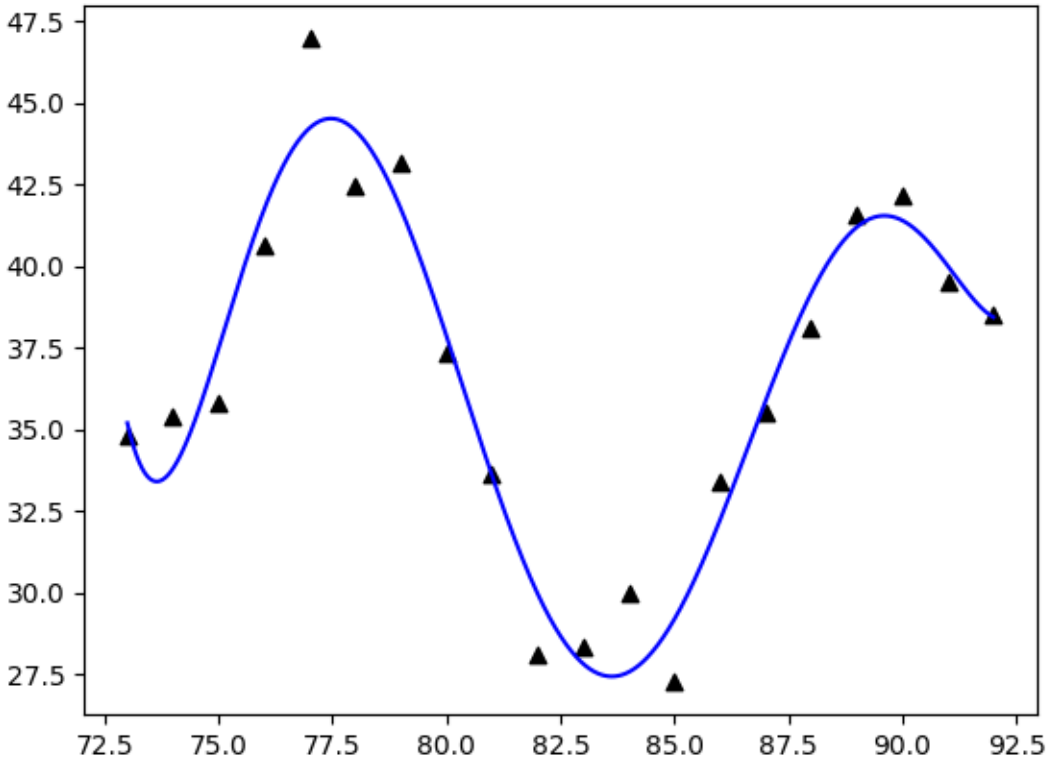
Programa: *scatter_plot6.py*

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. Use uma função para gerar os polinômios de diferentes graus.

Abaixo temos o resultado 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
```




47.5
45.0
42.5
40.0
37.5
35.0
32.5
30.0
27.5
72.5
75.0
77.5
80.0
82.5
85.0
87.5
90.0
92.5

-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.