Aula 4 - NumPy e Pandas I

4.1 NumPy

O NumPy (Numerical Python) é um dos pacotes básicos mais importantes para o processamento numérico em Python. Isso pela sua estrutura de dados principal, o numpy array (array), que faz o processamento de dados de forma muito mais eficiente do que listas, por exemplo.

4.1.1Array NumPy

O array numpy é uma estrutura para armazenar dados numéricos (em sua maioria), e tem seu funcionamento como um vetor ou lista. Existem diversas formas de se criar o array. Abaixo criamos array de 3 formas distintas: usando uma lista com valores, a partir da função `range()` e a função `np.arange()` (equivalente ao range do NumPy).

In [1]:
```
import numpy as np

lista = [1,2,3,4,5] # lista normal

# Array de lista
array1 = np.array(lista)

# Array de range
array2 = np.array(range(10))

# Array de arange
array3 = np.arange(10)
```

Os arrays em NumPy podem ser processados de forma *vetorizada*, o que aumenta a eficiencia dos cálculos. Isso quer dizer que podemos realizar operações matemáticas em todos os elemntos da array sem usar laços for (sempre vai existir um laço, porém ele é realizado em funções pré-compiladas em C/C++ ou Fortran, imbutidas no pacote NumPy). Considere um vetor de 10000 elementos representado por uma lista e por um array, que deve ter seus elementos individuais multiplicados por 2. O código abaixo faz esses cálculos e coleta o tempo de processamento de cada um, usando uma lista e um array (com a função so Notebook %time).

In [2]:
```
l1 = list(range(100000))
l2 = np.arange(100000)

%time for i in range(len(l1)): l1[i] = l1[i]*2

%time l2 = l2 * 2

#%time for i in range(10)
```
```
CPU times: total: 46.9 ms
Wall time: 40 ms
CPU times: total: 0 ns
Wall time: 0 ns
```

4.1.2 Inicialização de arrays

np.arange

Existem outras formas de inicializarmos arrays. Usando `np.arange()` cria um array com valores internos. `np.arange()` possui vários argumentos que podem ser utilizados, algumas construções são mostradas abaixo:

In [3]:
```
# Valores entre 0 e 9
arr1 = np.arange(0,10)
arr1

# Valores entre 5 e 14
arr2 = np.arange(5,15)
arr2

# Valores entre 5 e 14 com passo de 0.5
arr3 = np.arange(5,15, 0.5)
arr3

# Valores entre -3 e 9 com passo de 0.5
arr4 = np.arange(-3, 10)
arr4
```
Out[3]:
```
array([-3, -2, -1,  0,  1,  2,  3,  4,  5,  6,  7,  8,  9])
```

np.zeros() e np.ones()

Podemos ainda inicializar arrays com valores nulos ou com valores unitários usando as funções `np.zeros()` e `np.ones()`:

In [4]:
```
# Array com 10 elementos nulos
arr0 = np.zeros(10)
arr0

# Array com 10 elementos iguais a 1
arr0 = np.ones(10)
arr0
```
Out[4]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

np.random()

`np.random` fornece diversas ferramentas para a geração de dados aleatórios em arrays. Abaixo algumas opções (extraídas de https://numpy.org/doc/1.16/reference/routines.random.html)

rand(d0, d1, ..., dn)	Random values in a given shape.
randn(d0, d1, ..., dn)	Return a sample (or samples) from the "standard normal" distribution.
randint(low[, high, size, dtype])	Return random integers from *low* (inclusive) to *high* (exclusive).
random_integers(low[, high, size])	Random integers of type np.int between *low* and *high*, inclusive.
random_sample([size])	Return random floats in the half-open interval [0.0, 1.0).
random([size])	Return random floats in the half-open interval [0.0, 1.0).
ranf([size])	Return random floats in the half-open interval [0.0, 1.0).
sample([size])	Return random floats in the half-open interval [0.0, 1.0).
choice(a[, size, replace, p])	Generates a random sample from a given 1-D array
bytes(length)	Return random bytes.

O código abaixo cria arrays de números aleatórios de diversas formas:

In [5]:
```
# Amostra de 10 números aleatórios gerados pela distribuição Normal Padrão
rand_arr1 = np.random.randn(10)
rand_arr1

# Amostra de 10 números aleatórios gerados uniformemente entre 0 e 5
rand_arr2 = np.random.randint(5, size = 10)
rand_arr2

# Amostra de 10 números aleatórios gerados uniformemente entre 100 e 200
rand_arr3 = np.random.randint(100,200, size = 10)
rand_arr3
```
Out[5]:
```
array([134, 112, 179, 169, 188, 101, 114, 182, 164, 116])
```

4.1.3 Arrays multidimensionais (N-dimensional array)

Arrays multidimensionais podem ser pensados como matrizes. Podemos criar arrays multidimensionais (ndarrays) das mesmas formas vistas acima, porém especificamos as suas dimensões. Abaixo alguns exemplos.

In [6]:
```
# A partir de uma lista de listas
lista_lista = [[1,2,3], [4,5,6]]
nd_arr1 = np.array(lista_lista)
nd_arr1

# Matriz 2x3 de aleatorios
nd_arr2 = np.random.randn(2,3)
nd_arr2

# Matriz 2x3 de zeros - passamos uma tupla com as dimensões
nd_arr3 = np.zeros((2,3))
nd_arr3

# Matriz 2x3 de 1 - passamos uma tupla com as dimensões
nd_arr3 = np.ones((2,3))
nd_arr3

#Criando uma matriz identidade 5x5:
iden = np.identity(5)
iden
```
Out[6]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

Podemos verificar o tamanho dos arrays usando o método `.shape()`. Este método retorna uma tupla com o número de elementos referente ao número de dimensões do array, e para cada dimensão, o número representa a quantidade de elementos que existe nela. Considere o exemplo:

In [7]:
```
# Matriz 2x3 de zeros - passamos uma tupla com as dimensões
nd_arr3 = np.zeros((2,3))
print(nd_arr3)

print(nd_arr3.shape)

print("Numero de linhas : \n", nd_arr3.shape[0])
print("Numero de colunas : \n", nd_arr3.shape[1])
```
```
[[0. 0. 0.]
 [0. 0. 0.]]
(2, 3)
Numero de linhas :
 2
Numero de colunas :
 3
```

4.1.4 Aritmética com arrays

Como dissemos, a grande vantagem de usar arrays está no processamento vetorizado, o que permite expressar operações matemáticas em lotes sem usar laços `for`. Qualquer operação matemática aplicada em um array faz a operação ser aplicada a todos os seus elementos. Considere os exemplos abaixo:

In [8]:
```
# Gera uma matriz 3x4 com dados aleatorios entre 2-100
arr4 = np.random.randint(2,6, size=(4,4))
print("Aleatorios :\n", arr4)

# Multiplica a linha 0 por 2:
arr4[0] = arr4[0]*2
print("Multiplica linha 0 por 2 : \n" ,arr4)

# Linha 0 - 1
arr4[0] = arr4[0] - 1
print("Linha 0 - 1 : \n" ,arr4)

# Eleva todos os elementos ao quadrado:
arr4 = arr4**2
print("Todos os elementos^2 : \n" ,arr4)

# Linha:
arr4[1] = arr4[1] - arr4[0]
print("Linha 1 -   linha 1 - linha 0 : \n" ,arr4)
```
```
Aleatorios :
 [[2 5 4 5]
 [5 4 2 4]
 [2 2 2 2]
 [5 5 4 5]]
Multiplica linha 0 por 2 :
 [[ 4 10  8 10]
 [ 5  4  2  4]
 [ 2  2  2  2]
 [ 5  5  4  5]]
Linha 0 - 1 :
 [[3 9 7 9]
 [5 4 2 4]
 [2 2 2 2]
 [5 5 4 5]]
Todos os elementos^2 :
 [[ 9 81 49 81]
 [25 16  4 16]
 [ 4  4  4  4]
 [25 25 16 25]]
Linha 1 -   linha 1 - linha 0 :
 [[  9  81  49  81]
 [ 16 -65 -45 -65]
 [  4   4   4   4]
 [ 25  25  16  25]]
```

Percebe-se que as operações algébricas ficam muito facilitadas com os arrays. Considere o código abaixo, que encontra a inversa da seguinte matriz:

```
M = [[4, 3, 3, 4],
     [4, 3, 3, 2],
     [8, 3, 5, 5],
     [5, 6, 3, 4]]
```

In [9]:
```
M = np.array([[10. ,3. ,3. ,4.],[2. ,3. ,3. ,2.],[8. ,3. ,5. ,5.],[5. ,6. ,3. ,4.]])
MI = np.identity(4)
#print("INVERSA :",np.linalg.inv(M))
#print("M \n", M)
for i in range(M.shape[0]):
    pivo  = M[i,i]
    M[i] = M[i] / pivo
    MI[i] = MI[i] / pivo
    #print("Pivo : ", pivo,"\n", M)
    for j in range(M.shape[1]):
        if i != j:
            MI[j] = MI[j] - MI[i] * M[j,i]
            M[j] = M[j] - M[i] * M[j,i]
print("Inversa : \n",MI)
```
```
Inversa :
 [[ 0.28125     0.15625    -0.1875     -0.125     ]
 [ 0.13541667  0.26041667 -0.3125      0.125     ]
 [ 0.09375     0.71875    -0.0625     -0.375     ]
 [-0.625      -1.125       0.75        0.5       ]]
```

Por sorte, podemos conferir o resultado pelo próprio NumPy...

In [10]:
```
M = np.array([[10. ,3. ,3. ,4.],[2. ,3. ,3. ,2.],[8. ,3. ,5. ,5.],[5. ,6. ,3. ,4.]])
print("Inversa pelo NumPy : \n",np.linalg.inv(M))
```
```
Inversa pelo NumPy :
 [[ 0.28125     0.15625    -0.1875     -0.125     ]
 [ 0.13541667  0.26041667 -0.3125      0.125     ]
 [ 0.09375     0.71875    -0.0625     -0.375     ]
 [-0.625      -1.125       0.75        0.5       ]]
```

Uma observação importante é em relação ao tipo numérico dos arrays. Considere o seguinte caso, em que uma matriz é criada e a primeira linha substituida por ela /10.

In [11]:
```
M = np.array([[10 ,3 ,3 ,4],[2 ,3 ,3 ,2],[8 ,3 ,5 ,5],[5 ,6 ,3 ,4]])
M[0] = M[0]/10
print(M[0])
```
```
[1 0 0 0]
```

O resultado não é o esperado, pois o tipo dos dados foi inferido como inteiro. Podemos verificar o tipo de dados usando o `dtype` (no caso abaixo, int32):

In [12]:
```
print(M.dtype)
```
```
int32
```

O problema pode ser corrigido ao se inicializar os valores da matriz, colocando um ponto após os números, indicando que são reais:

In [13]:
```
M = np.array([[10. ,3. ,3. ,4.],[2. ,3. ,3. ,2.],[8. ,3. ,5. ,5.],[5. ,6. ,3. ,4.]])
M[0] = M[0]/10
print(M[0])
print(M.dtype)
```
```
[1.  0.3 0.3 0.4]
float64
```

Ou ainda especificando o próprio tipo dos dados:

In [14]:
```
M = np.array([[10 ,3 ,3 ,4],[2 ,3 ,3 ,2],[8 ,3 ,5 ,5],[5 ,6 ,3 ,4]], dtype=np.float64)
M[0] = M[0]/10
print(M[0])
print(M.dtype)
```
```
[1.  0.3 0.3 0.4]
float64
```

4.1.5 Fatiamento de arrays

O fatiamento de arrays permite visualizar partes do mesmo. Para arrays unidimensionais a sintaxe é muito parecida com o fatiamento de listas. Considere os exemplos abaixo.

In [15]:
```
# Gera 10 valores extraidos da normal padrão
arr = np.random.randn(10)
print(arr)

# Imprime os 5 primeiros valores (de 0 a 4)
print(arr[:5])

# Imprime os últimos valores, a partir do indice 5
print(arr[5:])

# Imprime os elementos de indice 2-5
print(arr[2:6])
```
```
[ 1.13844904  0.75338813 -1.10789182 -1.48875892  2.52130039 -0.63508068
 -0.22897935  0.52784607 -0.13698734  0.08685846]
[ 1.13844904  0.75338813 -1.10789182 -1.48875892  2.52130039]
[-0.63508068 -0.22897935  0.52784607 -0.13698734  0.08685846]
[-1.10789182 -1.48875892  2.52130039 -0.63508068]
```

Uma diferença importante entre o fatiamento de listas e de arrays, é que estes últimos são visualizações (views) do próprio array, ou seja, alterando a visualização também altera o array. Considere o exemplo:

In [16]:
```
arr = np.zeros(10, dtype = np.float64)
print(arr)

arr[:5] = 10
print("Alterando os valores por fatiamento :", arr)
```
```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Alterando os valores por fatiamento : [10. 10. 10. 10. 10.  0.  0.  0.  0.  0.]
```

Se quisermos uma cópia do fatiamento precisamos dizer explicitamente, usando o método `.copy()`

In [17]:
```
arr = np.zeros(10, dtype = np.float64)
print(arr)

copia = arr[:5].copy()
copia = 10
print("Copiando não altera os valores :", arr)
```
```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Copiando não altera os valores : [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Em arrays multidimensionais os fatiamentos de cada índice não são mais escalares, mas arrays unidimensionais. Considere o caso 2d:

In [18]:
```
M = np.array([[10. ,3. ,3. ,4.],[2. ,3. ,3. ,2.],[8. ,3. ,5. ,5.],[5. ,6. ,3. ,4.]])
print("Matriz original : \n", M)

# Imprime todas as linhas a partir do indice 1
print("Linhas a partir do indice 1 :\n",M[1:])

# De todas as linhas a partir do indice 1 (igual anterior), seleciona as colunas até o indice 2
print("Colunas até o indice 2, das linhas a partir do indice 1 :\n",M[1:,:3])
```
```
Matriz original :
 [[10.  3.  3.  4.]
 [ 2.  3.  3.  2.]
 [ 8.  3.  5.  5.]
 [ 5.  6.  3.  4.]]
Linhas a partir do indice 1 :
 [[2. 3. 3. 2.]
 [8. 3. 5. 5.]
 [5. 6. 3. 4.]]
Colunas até o indice 2, das linhas a partir do indice 1 :
 [[2. 3. 3.]
 [8. 3. 5.]
 [5. 6. 3.]]
```

4.1.6 Indexação booleana

Também podemos realizar operações booleanas em arrays, de forma que o resultado será um novo array de valores booleanos, de acordo com a condição. Considere o exemplo:

In [19]:
```
arr_string = np.array(["Dwight", "Michael", "Angela", "Oscar", "Michael", "Angela"])

# Condição : quais elementos do array são iguais a "Michael"?
arr_bool = arr_string == "Michael"
print(arr_bool)

# Condição : quais elementos do array são iguais a "Michael" OU "Angela"
arr_bool = (arr_string == "Michael") | (arr_string == "Angela")
print(arr_bool)
```
```
[False  True False False  True False]
[False  True  True False  True  True]
```

Também podemos fazer o processo reverso: passamos um array de booleanos para um array, e ele retorna somente os elementos (ou arrays) em que a condição é verdadeira:

In [20]:
```
arr_string = np.array(["Dwight", "Michael", "Angela", "Oscar", "Michael", "Angela"])
arr_booleano = np.array([True,False,False,True,False,False])

# Seleciona somente os elementos em que arr_booleano == True
print(arr_string[arr_booleano])
```
```
['Dwight' 'Oscar']
```

Também podemos fazer uma indexação booleana em arrays multidimensionais. Nesses casos as condições verdadeiras retornam arrays de dimensões menores. Considere o seguinte caso:

In [21]:
```
# Gerando uma matriz 5x4 de elementos entre 0 e 9
ndarray = np.random.randint(5,10, size=(5,4))

# Gerando um array de booleanos com o mesmo número de elementos da primeira dimensão da Matriz (5)
arr_bool = np.array([True,False,False,False,False])

# Imprimindo somente as linhas de ndarray que satisfazem as condições de arr_bool
print(ndarray[arr_bool])
```
```
[[9 8 9 5]]
```

Combinando as duas indexações nos fornece uma poderosa ferramenta para a análise de dados. Considere o seguinte cenário: temos os dados de produção de uma indústria de pães, em que a cada vez que um lote é produzido, uma amostra de 5 pães é verificada pela qualidade, aferindo o peso total. Os tipos de pães são armazenados em um array chamado `arr_paes` e as coletas dos pesos em uma ndarray chamado `arr_pesos`. Os valores são os seguintes: `arr_paes = np.array(["frances","italiano","sirio","frances","sirio"])`

```
arr_pesos = np.array([[3.0,2.8,3.1,3.0,3.23],
                      [5.0,5.3,4.95,4.9,5.23],
                      [3.0,2.8,3.1,3.0,3.23],
                      [6.0,6.8,6.1,6.0,6.23],
                      [3.0,2.8,3.1,3.0,3.23]])
```

Podemos realizar filtros na matriz de pesos com base nos pães que desejamos verificar. Considere os exemplos:

In [22]:
```
arr_paes = np.array(["frances","italiano","sirio","frances","sirio"])
arr_pesos = np.array([[3.0,2.8,3.1,3.0,3.23],
                      [5.0,5.3,4.95,4.9,5.23],
                      [3.0,2.8,3.1,3.0,3.23],
                      [6.0,6.8,6.1,6.0,6.23],
                      [3.0,2.8,3.1,3.0,3.23]])

# Filtrando todas as linhas que contêm medidas do pão francês
arr_frances = arr_pesos[arr_paes == "frances"]
print("Linhas pao frances :\n", arr_frances)

# Filtrando todas as linhas que contêm medidas do pão sirio
arr_sirio = arr_pesos[arr_paes == "sirio"]
print("Linhas pao sirio :\n", arr_sirio)

# Filtrando todas as linhas que contêm medidas do pão sirio OU frances
arr_frances_sirio = arr_pesos[(arr_paes == "sirio") | (arr_paes == "frances")]
print("Linhas pao sirio ou frances :\n", arr_frances_sirio)
```
```
Linhas pao frances :
 [[3.   2.8  3.1  3.   3.23]
 [6.   6.8  6.1  6.   6.23]]
Linhas pao sirio :
 [[3.   2.8  3.1  3.   3.23]
 [3.   2.8  3.1  3.   3.23]]
Linhas pao sirio ou frances :
 [[3.   2.8  3.1  3.   3.23]
 [3.   2.8  3.1  3.   3.23]
 [6.   6.8  6.1  6.   6.23]
 [3.   2.8  3.1  3.   3.23]]
```

Note que a indexação booleana, diferentemente do fatiamento, não produz uma view do array, mas sim uma cópia! Ou seja, alterar o resultado de uma indexação booleana não altera os valores originais. Considere o exemplo abaixo:

In [23]:
```
arr_paes = np.array(["frances","italiano","sirio","frances","sirio"])
arr_pesos = np.array([[3.0,2.8,3.1,3.0,3.23],
                      [5.0,5.3,4.95,4.9,5.23],
                      [3.0,2.8,3.1,3.0,3.23],
                      [6.0,6.8,6.1,6.0,6.23],
                      [3.0,2.8,3.1,3.0,3.23]])

arr_frances = arr_pesos[arr_paes == "frances"]
print(arr_frances)
arr_frances[0] = 99
print("Alterando arr_frances :\n",arr_frances)

print("Não altera arr_pesos :\n",arr_pesos)
```
```
[[3.   2.8  3.1  3.   3.23]
 [6.   6.8  6.1  6.   6.23]]
Alterando arr_frances :
 [[99.   99.   99.   99.   99.  ]
 [ 6.    6.8   6.1   6.    6.23]]
Não altera arr_pesos :
 [[3.   2.8  3.1  3.   3.23]
 [5.   5.3  4.95 4.9  5.23]
 [3.   2.8  3.1  3.   3.23]
 [6.   6.8  6.1  6.   6.23]
 [3.   2.8  3.1  3.   3.23]]
```

4.1.7 Métodos matemáticos e estatísticos

Os arrays do NumPy possuem muitos métodos que matemáticos e estatísticos que facilitam o processamento. Alguns deles: `sum()` `mean()` `std()`,`var()` `cumsum()` `min()`, `max()` `argmin()`, `argmax()`

In [24]:
```
# Gera 20 elementos aleatorios (entre 10 e 19)
arr_rand = np.random.randint(10,20, size=(20))
print("Valores :\n", arr_rand)

# Calcula a soma
print("Soma :\n", arr_rand.sum())

# Calcula a média
print("Média :\n", arr_rand.mean())

# Calcula o desv. padrão
print("Desvio padrão :\n", arr_rand.std())

# Calcula a variancia
print("Variância :\n", arr_rand.var())

# Máximo
print("Máximo :\n",arr_rand.max())

# Indice do Máximo
print("Indice do Máximo :\n",arr_rand.argmax())

# Soma cumulativa dos elementos começando em 0
print("Soma cumulativa :\n",arr_rand.cumsum())
```
```
Valores :
 [18 14 14 15 14 15 17 16 12 18 16 16 16 14 12 15 14 13 11 14]
Soma :
 294
Média :
 14.7
Desvio padrão :
 1.8466185312619388
Variância :
 3.41
Máximo :
 18
Indice do Máximo :
 0
Soma cumulativa :
 [ 18  32  46  61  75  90 107 123 135 153 169 185 201 215 227 242 256 269
 280 294]
```

Em arrays multidimensionais podemos escolher em relação a qual eixo que desejamos coletar as informações (não todas):

In [25]:
```
arr_m = np.array([[1,1,1,1],
                  [4,5,6,6],
                  [10,4,3,2]])

print("Média por colunas", arr_m.mean(axis=0))
print("Média por linhas", arr_m.mean(axis=1))

print("Maior elemento", arr_m.max())
```
```
Média por colunas [5.         3.33333333 3.33333333 3.        ]
Média por linhas [1.   5.25 4.75]
Maior elemento 10
```

Exercícios I

1. Escreva os seguintes vetores como arrays numpy:

   `v1 = [10,20,30,20,10,1,0,2,5,0,20,1,4,0,20,20,20,30,40,11,44,55]`

   `v2 = [1,25,50,41,5,20,10,23,5,10,28,13,4,20,100,20,50,35,40,4,55,55]`

2. Considere as seguintes sequências matemáticas, e para cada uma delas escreva um algoritmo que armazene os elementos em um array, e calcule a soma e o desvio padrão dos valores.

   A. $\{n\}, n = 1, \ldots, 100$

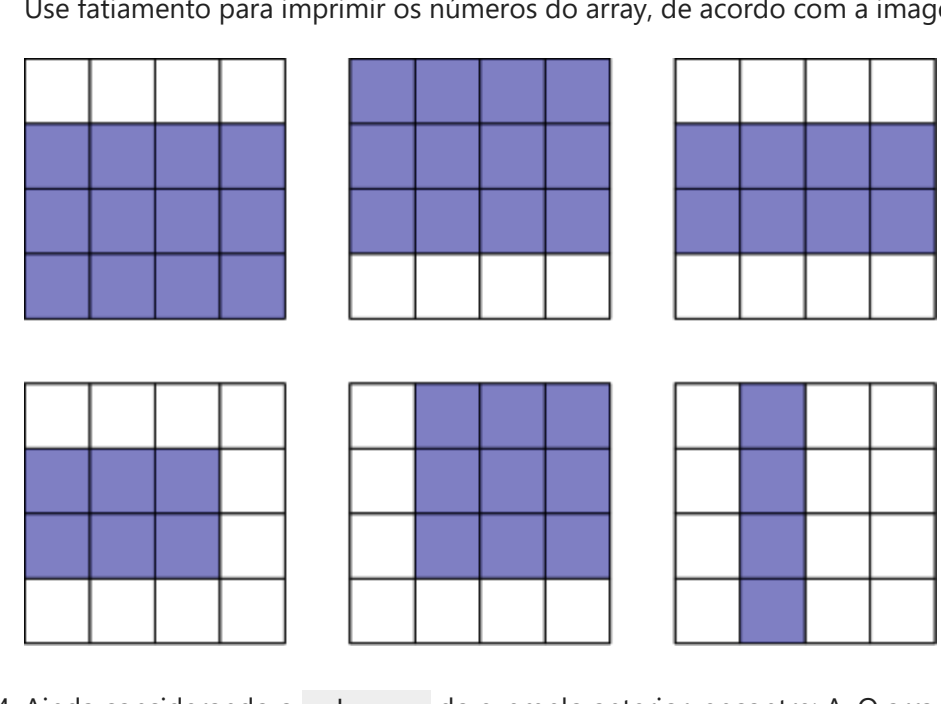
   B. $\left\{\frac{n}{n+1}\right\}, n = 1, \ldots 100 = \left\{\frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \ldots\right\}$

   C. $\left\{\frac{(-1)^n(n+1)}{3^n}\right\}, n = 1, \ldots 100 = \left\{-\frac{2}{3}, \frac{3}{9}, -\frac{4}{27}, \ldots\right\}$

3. Crie um array `arrN20` com 20 dados aleatórios extraidos da distribuição Normal padrão.
4. Crie um array `arrU20` com 20 dados aleatórios extraídos de uma distribuição uniforme, com valores entre -10 e 10.
5. Imprima a multiplicação de `arrN20` por 10.
6. Imprima a multiplicação de `arrN20` por `arrU20`, esse é o resultado esperado de uma multiplicação vetorial?
7. Gere uma ndarray `MN` 5x10 com dados extraidos da Normal padrão.
8. Gere uma ndarray `MU` 5x10 com dados extraidos de uma Uniforme(-10,60).
9. Considerando o array `arrN20`, imprima somente os valores positivos.
10. Considerando o array `arrU20`, imprima os valores entre os índices 5 e 10 incluindo o 10 (usando fatiamento).
11. Considerando o array `arrU20`, imprima todos os valores, exceto o primeiro (usando fatiamento).
12. Considerando o array `arrU20`, imprima todos os valores, exceto o último (usando fatiamento).
13. Considere o seguinte matriz:

    ```
    M = np.array([[4, 3, 3, 4],
                  [4, 3, 3, 2],
                  [5, 3, 5, 5],
                  [5, 3, 3, 4]])
    ```

    Use fatiamento para imprimir os números do array, de acordo com a imagem abaixo:



14. Ainda considerando o `ndarray` do exemplo anterior, encontre: A. O array com a soma dos elementos por linha. B. O array com a soma dos elementos por colunas. C. A soma e a média de todos os elementos.
15. Resolva os sistemas de equações lineares abaixo usando NumPy (https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html):

    A. $2x + 3y = 4$

    $x - 5y = 2$

    B. $x + 2y - 3z = 1$

    $3x - y + 2z = 0$

    $2x + y + z = 2$

16. Considere os seguintes dados de coleta de amostras de pesos de pães (como no exemplo): `arr_paes = np.array(["frances","italiano","sirio","frances","sirio"])`

    ```
    arr_pesos = np.array([[3.0,2.8,10,3.0,3.23,3.0,2.8,3.0],
      [5.0,5.3,4.95,4.9,5.23,5.,5.,6.],
      [3.0,2.8,3.1,3.0,3.23,3.,3.,3.],
      [6.0,6.8,6.1,6.0,6.23,5.8,5.9,6.],
      [3.0,2.8,3.1,3.0,3.23,3.,3.1,3.]])
    ```

    O controle de qualidade define que, se uma amostra têm variancia maior do que metade da média, existe algo errado com os dados(muita variabilidade), de forma que a amostra deve ser coletada novamente. Crie um código (usando indexação booleana) que retorne o pão (se existir algum) que precise de uma nova amostra coletada.

17. Ainda considerando os dados dos pães. A qualidade precisa saber a média dos pesos de todos os pães no primeiro e no último dia de coletas. Use fatiamentos e no máximo duas linhas para extrair as duas informações.

4.2 Pandas I

O pandas é um pacote essêncial para se realizar análise de dados, muito disso se dá pelas suas duas estruturas de dados principais, a `series` e o `Dataframe`, usados em quase todas as aplicações de mineração de dados. Utilizaremos a importação do pacote com a seguinte convenção:

In [26]:
```
import pandas as pd
```

4.2.1 Series

Uma Serie é um objeto do tipo array unidimensional contendo uma sequência de valores (de tipos semelhantes aos do NumPy) e um array associado de rótulos (*labels*) de dados, chamado *índice*. A series mais simples é composta de um array de dados:

In [27]:
```
ser1 = pd.Series([4,3,4,5])
print(ser1)
print(type(ser1))
```
```
0    4
1    3
2    4
3    5
dtype: int64
<class 'pandas.core.series.Series'>
```

Podemos acessar tanto os valores quanto os indices de uma Series pelos métodos `values` e `index`:

In [28]:
```
print(ser1.values)
print(ser1.index)
```
```
[4 3 4 5]
RangeIndex(start=0, stop=4, step=1)
```

Note que o tipo de estrutura de dados do `values` é justamente um array NumPy:

In [29]:
```
print(type(ser1.values))
```
```
<class 'numpy.ndarray'>
```

Podemos criar uma Series e alterar os valores de index para o que quisermos, considere:

In [30]:
```
ser2 = pd.Series([1,2,3,4], index=["a","b","c","d"])
print(ser2)
```
```
a    1
b    2
c    3
d    4
dtype: int64
```

Alterando elementos

Podemos usar os valores dos indices para acessar e alterar os elementos:

In [31]:
```
print(ser2["a"])

# Alterando o elemento
ser2["a"] = 999
print(ser2.values)
```
```
1
[999   2   3   4]
```

O método unique()

O método unique() é muito usado em dataframes (próxima seção), ele retorna os valores de uma series *sem repetição*. Considere o seguinte exemplo:

In [32]:
```
sr_string = pd.Series(["P1","P2","P3","P2","P1"])

#Retornando somente os valores sem repetição:
sr_string.unique()
```
Out[32]:
```
array(['P1', 'P2', 'P3'], dtype=object)
```

4.2.2 Dataframe

Um dataframe representa uma tabela de dados retangular e contém uma coleção ordenada de colunas, em que cada uma é uma Series e pode ter um tipo de dado diferente. O dataframe têm um índice tanto para as linhas quanto para as colunas. Existem diversas formas para se criar Dataframes (embora na maioria dos casos ele será criado automáticamente ao carregarmos dados externos), algumas delas são mostradas abaixo:

Criação de Dataframes

In [33]:
```
# DataFrame a partir de um dicionário de listas: as chaves são os nomes das colunas e as listas os valores
dic1 = {"peça1":[1,2,3,4],
        "peça2":[5,2,3,5],
        "peça3":[2,3,4,3]}
dt1 = pd.DataFrame(dic1)
print(dt1)

# DataFrame a partir de uma lista de tuplas
l_tuplas = [(10,20,30),(40,50,60),(70,80,90)]
dt_tuplas = pd.DataFrame(l_tuplas)
print(dt_tuplas)
```
```
   peça1  peça2  peça3
0      1      5      2
1      2      2      3
2      3      3      4
3      4      5      3
    0   1   2
0  10  20  30
1  40  50  60
2  70  80  90
```

Os indices das linhas foram criados automaticamente no dataframe, porém poderiamos criá-los também:

In [34]:
```
# DataFrame a partir de um dicionário de listas com indices criados
dic2 = {"peça1":[1,2,3,4],
        "peça2":[5,2,3,5],
        "peça3":[2,3,4,3]}
dt2 = pd.DataFrame(dic2, index = [3,4,5,6])
print(dt2)
```
```
   peça1  peça2  peça3
3      1      5      2
4      2      2      3
5      3      3      4
6      4      5      3
```

Podemos selecionar as colunas do DataFrame pela notação em colchetes com o nome da coluna:

In [35]:
```
# Selecionando uma coluna
dt1["peça1"]

# Note que o tipo de dados da coluna é:
print("Tipo da coluna :", type(dt1["peça1"]))

# Assim, sabemos que a Series tem duas partes; values e index, e que values é um NumPy array, podemos extrair a
# dataframes como arrays e usar tudo que já sabemos sobre o NumPy:
arr = dt1["peça1"].values
print(type(arr))

# Extraindo dados de um array normalmente
print("Soma :",arr.sum())
print("Maximo :",arr.max())
print("Minimo :",arr.min())
```
```
Tipo da coluna : <class 'pandas.core.series.Series'>
<class 'numpy.ndarray'>
Soma : 10
Maximo : 4
Minimo : 1
```

Também podemos acessar as linhas usando o método `loc`, que retorna uma **Series**. O método `loc` é usado para acessarmos os nomes dos indices, quando o DataTable não possuir nomes serão os índices (neste exemplo acessamos os indices):

In [36]:
```
print(dt1.loc[0])
```
```
peça1    1
peça2    5
peça3    2
Name: 0, dtype: int64
```

Neste caso o indice deve existir. Tentando acessar a linha de indice 0 de dt2, um erro ocorre, pois ao criarmos o DF alteramos os índices, e o 0 não foi incluido:

In [37]:
```
print(dt2.loc[0])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File ~\AppData\Roaming\Python\Python310\site-packages\pandas\core\indexes\base.py:3621, in Index.get_loc(self, key, method, tolerance)
   3620 try:
-> 3621     return self._engine.get_loc(casted_key)
   3622 except KeyError as err:

File ~\AppData\Roaming\Python\Python310\site-packages\pandas\_libs\index.pyx:136, in pandas._libs.index.IndexEngine.get_loc()

File ~\AppData\Roaming\Python\Python310\site-packages\pandas\_libs\index.pyx:163, in pandas._libs.index.IndexEngine.get_loc()

File pandas\_libs\hashtable_class_helper.pxi:2131, in pandas._libs.hashtable.Int64HashTable.get_item()

File pandas\_libs\hashtable_class_helper.pxi:2140, in pandas._libs.hashtable.Int64HashTable.get_item()

KeyError: 0

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Input In [37], in <cell line: 1>()
----> 1 print(dt2.loc[0])

File ~\AppData\Roaming\Python\Python310\site-packages\pandas\core\indexing.py:967, in _LocationIndexer.__getitem__(self, key)
    964 axis = self.axis or 0
    966 maybe_callable = com.apply_if_callable(key, self.obj)
--> 967 return self._getitem_axis(maybe_callable, axis=axis)

File ~\AppData\Roaming\Python\Python310\site-packages\pandas\core\indexing.py:1202, in _LocIndexer._getitem_axis(self, key, axis)
   1200 # fall thru to straight lookup
   1201 self._validate_key(key, axis)
-> 1202 return self._get_label(key, axis=axis)

File ~\AppData\Roaming\Python\Python310\site-packages\pandas\core\indexing.py:1153, in _LocIndexer._get_label(self, label, axis)
   1151 def _get_label(self, label, axis: int):
   1152     # GH#5667 this will fail if the label is not present in the axis.
-> 1153     return self.obj.xs(label, axis=axis)

File ~\AppData\Roaming\Python\Python310\site-packages\pandas\core\generic.py:3864, in NDFrame.xs(self, key, axis, level, drop_level)
   3862             new_key = index[loc]
   3863 else:
-> 3864     loc = index.get_loc(key)
   3866     if isinstance(loc, np.ndarray):
   3867         if loc.dtype == np.bool_:

File ~\AppData\Roaming\Python\Python310\site-packages\pandas\core\indexes\base.py:3623, in Index.get_loc(self, key, method, tolerance)
   3621     return self._engine.get_loc(casted_key)
   3622 except KeyError as err:
-> 3623     raise KeyError(key) from err
   3624 except TypeError:
   3625     # If we have a listlike key, _check_indexing_error will raise
   3626     #  InvalidIndexError. Otherwise we fall through and re-raise
   3627     #  the TypeError.
   3628     self._check_indexing_error(key)

KeyError: 0
```

Ainda, podemos usar índices sequenciais inteiros para acessar as linhas (mesmo que as mesmas tenham outros nomes em seus índices) usando o método `iloc` (que também retorna uma **Series**):

```
dt2.iloc[0]
```

Usando o método `columns` obtemos um objeto do tipo Index com as colunas do DataFrame.

```
type(dt1.columns)
```

Podemos adicionar uma nova coluna no DataFrame usando as chaves com o nome da coluna:

```
# Todos os elementos da nova coluna são preenchidas com o valor 10
dt1["Nova coluna"] = 10
dt1
```

Da mesma forma podemos remover colunas usando o método `del`.

```
del dt1["peça1"]
dt1
```

Ordenando

Podemos ordenar todo um dataframe com base nos dados de uma coluna usando a função `.sort_values()`, passando o argumento `by=` com o nome da coluna que queremos ordenar. Considerando o banco de dados das peças, o código abaixo ordena a dataframe de Peças pelos valores de peça2 (note que os índices das linhas foram alterados):

```
dt1.sort_values(by="peça2")
```

Carregando dados em um DataFrame

A maior utilidade dos DataFrames é a manipulação de dados. Dessa forma, o Pandas contém inúmeras maneiras para se carregar dados externos, e a estrutura de dados padrão gerada é um Dataframe. Inicialmente, faremos a leitura de dados no formato .csv do próprio computador. Para isso usamos o método `pd.read_csv()`. Esse método possui diversos parametros (mais de 50!), porém os dois principais são: o caminho do arquivo a ser lido e o delimitador dos dados. Considere o exemplo abaixo que carrega os dados "e-shop clothing 2008.csv", contido na pasta Data.

```
caminho = "G:\\Meu Drive\\Arquivos\\UFPR\\Disciplinas\\2 - Intro  Mineração de Dados\\Python\\Datasets\\e-shop
dt = pd.read_csv(caminho,sep = ";")
dt
```

Alguns repositórios de dados disponibilizam os mesmos diretamente na internet, de forma que podemos carregar os dados sem mesmo baixá-los no computador. Para isso só precisamos do URL dos dados. Por exemplo: https://raw.githubusercontent.com/cs109/2014_data/master/countries.csv. Lendo esses dados em um DataFrame temos:

```
caminho_url = "https://raw.githubusercontent.com/cs109/2014_data/master/countries.csv"
dt_url = pd.read_csv(caminho_url, sep = ",")
dt_url
```

Também podemos ler dados tabelares direto de uma planilha de excel com o método `read_excel`. OBS: Para isso o pandas requer a instalação do pacote `openpyxl`. Assim, abra um terminal e instale o pacote pelo *pip install*:

`pip install openpyxl`

```
# Podemos usar a string pura do caminho (sem barras invertidas), usando a letra 'r' antes de começar o caminho
caminho_excel = r"G:\Meu Drive\Arquivos\UFPR\Disciplinas\2 - Intro  Mineração de Dados\Python\Datasets\db_addre
dt_excel = pd.read_excel(caminho_excel)
dt_excel
```

Exportando dados de um DataFrame

Podemos exportar os dados de um DataFrame usando o método `.to_csv()`, em seu modo mais simples com o único argumento do caminho do arquivo.

```
caminho = "G:\Meu Drive\Arquivos\UFPR\Disciplinas\Arquivo_exportado.csv"
dt1.to_csv(caminho)
```

```
dt1
```

Exportando dessa forma surgem 3 problemas (ou melhoras possíveis):

1. As índices das linhas foram exportados também.
2. O arquivo não fica tabulado ao abri-lo com o Excel.
3. Os nomes não estão com a acentuação correta.

Para melhorar a exportação usamos os seguintes argumentos:

1. `index = False`: Não exporta o índice das linhas.
2. `sep = ";"`: Adicionando o separador ';' para os dados ficarem tabulares no Excel.
3. `encoding = "utf-8-sig"`: Permite exportar acentos.

Assim, o código melhorado fica:

```
caminho = "G:\Meu Drive\Arquivos\UFPR\Disciplinas\Arquivo_exportado.csv"
dt1.to_csv(caminho, sep = ";", index = False, encoding = "utf-8-sig")
```

Assim que carregamos um conjunto de dados, podemos obter algumas informações superficiais e rápidas sobre eles, por exemplo:

1. `.shape`: Retorna uma tupla com o número de linhas e colunas do DataFrame.
2. `.info()`: Mostra o nome das colunas e seus tipos de dados associados.
3. `.describe()`: Retorna um *DataFrame* com várias estatísticas descritivas sobre as colunas.

```
#dt1.shape
#dt1.info()
dt1.describe()
```

Filtros e indexação booleana

Os DataFrames são muito utilizados para realizarmos filtros no banco de dados. A mesma lógica da indexação booleana e do fatiamento usadas nas listas e ndarrays pode ser utilizada aqui. Considere o conjunto de dados `Production_Data.csv`:

```
dt_production = pd.read_csv("G:\Meu Drive\Arquivos\UFPR\Disciplinas\2 - Intro  Mineração de Dados\Python\Datase
```

Podemos aplicar uma condição booleana em alguma das colunas, para obtermos um array de True/False. Por exemplo, todas as linhas em que a coluna "Activity" é igual a "Turning & Milling - Machine 4":

```
cond = dt_production["Activity"] == "Turning & Milling - Machine 4"
cond
```

Se atribuirmos esse vetor ao dataframe, teremos somente as linhas em que a cond. é verdadeira:

```
dt_production[cond]
```

Podemos escrever a mesma coisa de forma direta, ou seja, colocamos a condição diretamente no dataframe:

```
dt_production[ dt_production["Activity"] == "Turning & Milling - Machine 4"]
```

Usando o método *unique()* (das Series) em um determinado atributo (coluna), conseguimos encontrar os valores sem repetição que ocorrem nos registros dessa coluna. Por exemplo, quais são os tipos de atividade desempenhadas nesse banco de dados?

```
dt_production["Activity"].unique()
```

Exercícios II

1. Considerando o exercício dos pães da seção anterior: salve os dados em um dataframe adequado para se realizar operações. Crie 2 dataframes iguais usando métodos diferentes: um a partir de um dicionário e um a partir de uma lista de tuplas.
   A. Usando o dataframe, encontre as somas de pesos por dias da semana e por tipo de pão.
   B. Usando o dataframe, encontre a média de pesos por dias da semana e por tipos de pão.
2. Leia os dados 'clientes_shopping.csv', exclua a coluna "Genre" e exporte os dados novamente em um arquivo *.csv*.
3. Considerando o conjunto de dados *MateriaisConstrução.xlsx*. Este conjunto contém dados referente a compra em uma loja de construção. Cada linha representa um pedido, sendo que as colunas contém os itens comprados e as células as quantidades adquiridas. Responda às seguintes questões:
   A. Quantos registros de compra existem?
   B. Quantos e quais os itens vendidos pela loja?
   C. Qual a média de vendas dos itens?
   D. Qual é o item com a maior média de vendas?
   E. Qual é o item que está presente na maioria das compras? Em quantas?
4. Considere o conjunto de dados *Production_Data.csv*. Este conjunto contém dados de produção, a coluna `Case ID` indica as ordens de produção, uma ordem de produção passa por diversas atividades (`Activity`), portanto existem diversas linhas para cada `Case ID`. A coluna `Worker ID` indica o número de identificação do funcionário que realizou a atividade. `Qty Rejected` indica quantas peças foram perdidas na atividade executada naquela linha. `Start Timestamp` e `Complete Timestamp` indicam as datas e horas de início e fim de processamento das atividades. Responda às seguintes questões:
   A. Quantos trabalhadores existem nesse db?
   B. Qual o total de peças rejeitadas no total?
   C. Quantas ordens de produção foram processados no total?
   D. Quais são as datas mais cedo e mais tarde de início de processamento de OPs?
   E. O db compreende um período de quantos dias de produção?
   F. Quais as médias de peças rejeitadas/dia e ordens de produção/dia no período todo?
   G. Quais as médias de peças rejeitadas/dia e ordens de produção/dia nos seguintes períodos:
      a. (2012/01/02,2012/02/01) -> janeiro
      b. (2012/02/01,2012/03/01) -> fevereiro
      c. (2012/03/01,2012/03/30) -> março
   H. Qual é o tempo médio, em minutos, de processamento da atividade *"Turning & Milling - Machine 4"*?
5. Considere o conjunto de dados 'ProducaoGrega.csv', com os dados de uma produção de cerâmica na Grécia, incluindo as informações do dia da semana, temperatura, medida do diametro e se houve defeitos ou não. Quais informações você pode extrair dos dados? Faça uma análise com o que já aprendeu.