

# Scientific Programming

## Practical 9

---

### Introduction

Luca Bianco - Academic Year 2020-21  
luca.bianco@fmach.it

# Ex 1

1. The file `top_3000_words.txt` is a one-column file representing the top 3000 English words. Read the file and for each letter, count how many words start with that letter. Store this information in a dictionary. Create a pandas series from the dictionary and plot an histogram of all initials counting more than 100 words starting with them.

```
a    221
b    145
c    304
d    163
e    166
f    142
g     79
h     92
i    126
j     23
k     16
l    102
m    140
n     64
o     81
p    249
q     12
r    165
s    338
t    175
u     37
v     45
w     99
y     15
z      1
dtype: int64
```

```
import pandas as pd
import matplotlib.pyplot as plt

def readFile(file):
    fh = open(file, "r")
    initials = dict()
    for line in fh:
        line = line.strip().lower()
        init = line[0]
        if init in initials:
            initials[init] += 1
        else:
            initials[init] = 1
    fh.close()

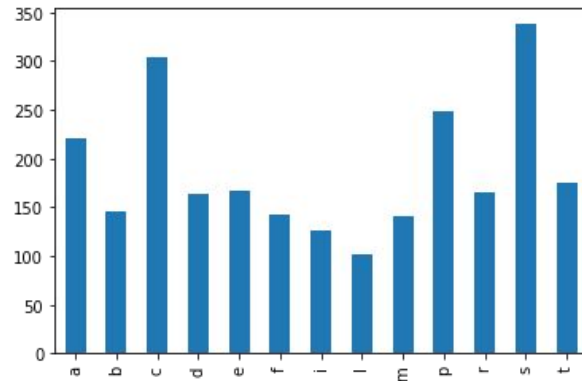
    return initials

def filter_and_plot(data, minCount = 10):
    series = pd.Series(data)

    #if you want to see the content of the series uncomment:
    #print(series)
    filt_series = series[series > minCount]
    filt_series.plot(kind = 'bar')
    plt.show()

inFile = "file_samples/top_3000_words.txt"

myDict = readFile(inFile)
filter_and_plot(myDict, 100)
```



# Ex 1

1. The file `top_3000_words.txt` is a one-column file representing the top 3000 English words. Read the file and for each letter, count how many words start with that letter. Store this information in a dictionary. Create a pandas series from the dictionary and plot an histogram of all initials counting more than 100 words starting with them.

	words	initial
0	a	a
1	abandon	a
2	ability	a
3	able	a
4	abortion	a
...	...	...
2995	your	y
2996	yours	y
2997	yourself	y
2998	youth	y
2999	zone	z

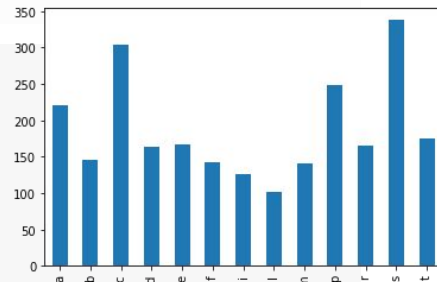
A	7
B	2
C	7
D	2
E	2
..	
a	214
b	143
c	297
d	161
e	164
f	141
g	77
h	92
i	118
j	20
k	16
l	101
..	

"""Solution 2 with DataFrame"""

```
def readFile2(file):
    data = pd.read_csv(file, header = None)
    return data

def filter_and_plot2(data, minCount = 10):
    data.columns = ['words'] #rename the column
    #Had to use lambda functions...
    #ex. lambda x : x + 1
    #check:
    #https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.apply.html
    #axis = 1 means we apply the function on all rows.
    data['initial'] = data.apply(lambda row : row['words'][0], axis = 1)
    #uncomment to see the data
    #print(data)
    d = data['initial'].value_counts()
    d = d.sort_index()
    #uncomment to see the final data
    #print(d)
    filt_data = d[d > minCount]
    filt_data.plot(kind = 'bar')
    plt.show()

print("-----")
print("----- Second solution -----")
print("-----")
d = readFile2(inFile)
filter_and_plot2(d,100)
```



# Numpy

Numpy is a fundamental library for **high performance scientific computations**. It provides fast and memory efficient data structures like **ndarray** with **broadcasting** capabilities, **standard mathematical functions** that can be applied on the arrays avoiding loops, **linear algebra** functions, **I/O** methods and it is well integrated with programming languages like C.

```
import numpy as np
```

# ndarray

Numpy ndarray is an N-dimensional array object designed to contain **homogeneous** data (i.e. all data must have the same type)

They have two information:

- a **shape**
- and
- a **dtype**

**np.ndarray.shape** returns a tuple with the dimensions

**np.ndarray.dtype** returns the type of the **homogeneous** data

np.array  
(list of lists)



```
import numpy as np

Aint = np.array([[1,2,3], [4,5,6]])
Afloat = np.array([[1.1,2,3], [4.2,5,6], [1,2,3]])

print(Aint)
print(type(Aint))
print(Aint.shape)
print(Aint.dtype)
print("")
print(Afloat)
print("type: {}".format(type(Afloat)))
print("shape: {}".format(Afloat.shape))
print("dtype: {}".format(Afloat.dtype))
```

```
[[1 2 3]
 [4 5 6]]
<class 'numpy.ndarray'>
(2, 3)
int64

[[1.1 2.  3. ]
 [4.2 5.  6. ]
 [1.  2.  3. ]]
type: <class 'numpy.ndarray'>
shape: (3, 3)
dtype: float64
```

# ndarray

Numpy ndarray is an **N-dimensional** array object designed to contain **homogeneous data** (i.e. all data must have the same type)

**np.ndarray.ndim** returns dimensionality (1D, 2D, 3D)

Original lists:

```
[0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0]  
[0.0, 1.0, 1.2599210498948732, 1.4422495703074083, 1.5874010519681994]
```

Numpy ndarray:

```
[0.          1.          1.41421356  1.73205081  2.          ]
```

The shape: (5,)

The dimensionality: 1

The type: float64

The 2D version of the ndarray:

```
[[0.          1.          1.41421356  1.73205081  2.          ]]
```

The shape: (1, 5)

The dimensionality: 2

The type: float64

Another 2D array:

```
[[0.          1.          1.41421356  1.73205081  2.          ]  
 [0.          1.          1.25992105  1.44224957  1.58740105]  
 [0.          1.          1.14869835  1.24573094  1.31950791]]
```

The shape: (3, 5)

The dimensionality: 2

The type: float64

```
import numpy as np  
import math
```

```
mysqrt = [math.sqrt(x) for x in range(0,5)]  
mycrt = [x**(1/3) for x in range(0,5)]  
myOtherRt = [x**(1/5) for x in range(0,5)]  
print("Original lists:")  
print(mysqrt)  
print(mycrt)  
print("")  
npData = np.array(mysqrt)  
print("Numpy ndarray:")  
print(npData)  
print("")  
print("The shape:", npData.shape)  
print("The dimensionality:", npData.ndim)  
print("The type:", npData.dtype)  
print("")
```

```
npData = np.array([mysqrt]) #NOTE: brackets!  
print("The 2D version of the ndarray:")  
print(npData)  
print("")  
print("The shape:", npData.shape)  
print("The dimensionality:", npData.ndim)  
print("The type:", npData.dtype)  
print("")
```

```
twoDarray = np.array([mysqrt, mycrt, myOtherRt])  
print("Another 2D array:")  
print(twoDarray)  
print("")  
print("The shape:", twoDarray.shape)  
print("The dimensionality:", twoDarray.ndim)  
print("The type:", twoDarray.dtype)
```

# ndarray

Zeros, ones and diagonals...

**dimensions are either a number or a tuple**

1. Array: `np.zeros(N)` or matrix: `np.zeros((N,M))`
2. Array: `np.ones(N)` or matrix: `np.ones((N,M))`
3. Matrix: `np.eye(N)`

**np.diag(values):** creates a diagonal matrix with values on the diagonal

```
Zero array (1x3)
[[ 0.  0.  0.]
```

```
Zero matrix (4x3)
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
```

```
Ones array (1x3)
[[ 1.  1.  1.]
```

```
Ones matrix (3x2)
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]
```

```
Diagonal matrix
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

```
Range 0-4
[0 1 2 3 4]
A diagonal matrix:
```

```
[[0 0 0 0 0]
 [0 1 0 0 0]
 [0 0 2 0 0]
 [0 0 0 3 0]
 [0 0 0 0 4]]
```

```
It's shape:
(5, 5)
```

```
import numpy as np
```

```
zeros = np.zeros(3)
zMat = np.zeros((4,3))
ones = np.ones(3)
oMat = np.ones((3,2))
diag = np.eye(4)
rng = np.arange(5) #5 excluded!
```

```
print("Zero array (1x3)")
print(zeros)
print("")
D = zMat.shape
print("Zero matrix ({}x{})".format(D[0],D[1]))
print(zMat)
print("")
print("Ones array (1x3)")
print(ones)
print("")
print("Ones matrix (3x2)")
print(oMat)
print("")
print("Diagonal matrix")
print(diag)
print("")
print("Range 0-4")
print(rng)
print("A diagonal matrix:")
dm = np.diag(rng)
print(dm)
print("Its shape:")
print(dm.shape)
```



# ndarray

Zeros, ones and diagonals...

Numpy has its own **range** method that is called `np.arange(N)`. Evenly spaced values in a range can be obtained also with `np.linspace(S,E, num=N, endpoint=True/False)` to obtain N linearly spaced values from S to E (included, unless endpoint = False is specified).

```
rng = np.arange(7)
print("Range 0-6")
print(rng)

myRange = np.linspace(-5,2.5,num =6)
print("6 linearly spaced elements in [-5 - 2.5]:")
print(myRange)

myRange = np.linspace(0,21,num =7, endpoint=False)
print("7 linearly spaced elements in [0 - 21):")
print(myRange)
```

```
Range 0-6
[0 1 2 3 4 5 6]
6 linearly spaced elements in [-5 - 2.5]:
[-5.  -3.5 -2.  -0.5  1.   2.5]
7 linearly spaced elements in [0 - 21):
[ 0.  3.  6.  9. 12. 15. 18.]
```



# ndarray

## Random values

Create a random array of 1000 values drawn from: 1. a gaussian distribution with  $\sigma = 20$  and  $\mu = 2$  2. a uniform distribution from 0 to 5 3. a binomial distribution with  $p = 0.5$  and  $n = 12$

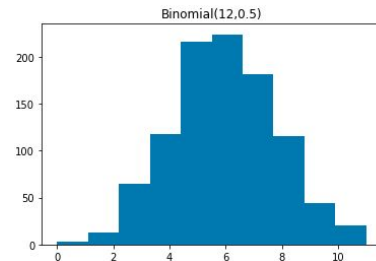
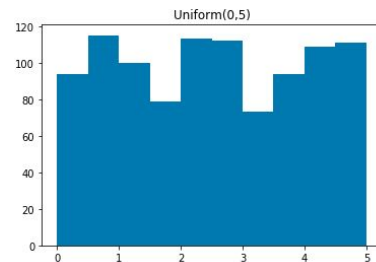
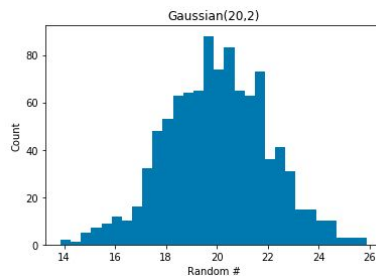
```
import numpy as np
import matplotlib.pyplot as plt

#Create the random number generator
rng = np.random.default_rng()

#get the gaussian random array
g = rng.normal(20,2, 1000)
#a uniform random array with vals in [0,5]
u = rng.uniform(0,5, 1000)
#get the binomial random array
b = rng.binomial(12,0.5, 1000)
```

```
plt.hist(g, bins = 30)
plt.title("Gaussian(20,2)")
plt.xlabel("Random #")
plt.ylabel("Count")
plt.show()
```

```
plt.hist(u, bins = 10)
plt.title("Uniform(0,5)")
plt.xlabel("Random #")
plt.ylabel("Count")
plt.show()
plt.hist(b, bins = 10)
plt.title("Binomial(12,0.5)")
plt.xlabel("Random #")
plt.ylabel("Count")
plt.show()
```



Distributions: <https://numpy.org/doc/stable/reference/random/generator.html#distributions>

# random seed

Random values...

... are they really random?

seed of PRSG  
is reinitialized to  
the same value!



```
u: [[0.30977448 0.61185445]
     [0.14990092 0.15617448]
     [0.57900782 0.76168045]]
```

```
u1: [[7.80653928e-01 8.92726466e-01]
      [7.61916339e-06 1.60210513e-01]
      [7.62850195e-01 7.09846092e-01]]
```

```
u2: [[0.86578736 0.72540134]
      [0.2779641 0.24814519]
      [0.43204036 0.53699348]]
```

With random seed reinit.

```
u: [0.63696169 0.26978671 0.04097352]
u1: [0.63696169 0.26978671 0.04097352]
u2: [0.63696169 0.26978671 0.04097352]
u3: [0.01652764 0.81327024 0.91275558]
```

```
import numpy as np
```

```
rng = np.random.default_rng()
u = rng.uniform(0,1,size=(3,2))
u1 = rng.uniform(0,1,size=(3,2))
u2 = rng.uniform(0,1,size=(3,2))
```

```
print(" u: {}\n\n u1:{}\n\n u2:{}".format(u,u1,u2))
```

```
print("")
```

```
print("With random seed reinit.")
```

```
rng = np.random.default_rng(0) #seed init at 0
```

```
u = rng.uniform(0,1,3)
```

```
rng = np.random.default_rng(0) #seed init at 0
```

```
u1 = rng.uniform(0,1,3)
```

```
rng = np.random.default_rng(0) #seed init at 0
```

```
u2 = rng.uniform(0,1,3)
```

```
#no reinit!!!
```

```
u3 = rng.uniform(0,1,3)
```

```
print(" u: {}\n u1:{}\n u2:{}\n u3:{}".format(u,u1,u2, u3))
```

# Numpy ↔ Pandas

```
{'two': 2, 'three': 3, 'one': 1, 'four': 4}
```

```
<class 'pandas.core.series.Series'>
```

```
<class 'numpy.ndarray'>
```

Numpy matrix

```
[[0 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 2 0 0 0]
 [0 0 0 3 0 0]
 [0 0 0 0 4 0]
 [0 0 0 0 0 5]]
```

Pandas DataFrame

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	0	0	0	0
2	0	0	2	0	0	0
3	0	0	0	3	0	0
4	0	0	0	0	4	0
5	0	0	0	0	0	5

Reindexed DataFrame

	a1	b2	c3	d4	e5	f6
A	0	0	0	0	0	0
B	0	1	0	0	0	0
C	0	0	2	0	0	0
D	0	0	0	3	0	0
E	0	0	0	0	4	0
F	0	0	0	0	0	5

```
import pandas as pd
import numpy as np
```

```
myDict = {"one" : 1, "two" : 2, "three" : 3, "four" : 4}
```

```
mySeries = pd.Series(myDict)
```

```
print(myDict)
print("")
print(mySeries)
print("")
print(type(mySeries))
print("")
print(type(mySeries.values))
print("")
```

```
myMat = np.diag(np.arange(6))
myDF = pd.DataFrame(myMat)
print("Numpy matrix")
print(myMat)
print("")
print("Pandas DataFrame")
print(myDF)
print("")
print("Reindexed DataFrame")
myDF = pd.DataFrame(myMat, index = list("ABCDEF"),
                    columns = ['a1', 'b2', 'c3', 'd4', 'e5', 'f6'])
print(myDF)
```



The values of  
the pd.Series  
are actually  
np. ndarray

# Reshaping

ndarrays can be reshaped...

**`np.ndarray.reshape(tuple)`** : returns a reshaped ndarray according to the **integer dimensions** in the tuple

**`np.ndarray.ravel()`** : flattens the matrix down to a 1D array

```
import numpy as np

myA = np.arange(10)

myB = myA.reshape(2,7)
print(myB)
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-9-2936ce504f51> in <module>()
      3 myA = np.arange(10)
      4
----> 5 myB = myA.reshape(2,7)
      6 print(myB)

ValueError: cannot reshape array of size 10 into shape (2,7)
```

```
import numpy as np

myA = np.arange(12)

print("The array:")
print(myA)
print("")
myB = myA.reshape((2,6))
print("Reshaped (2x6):")
print(myB)
print("")
myC = myA.reshape((3,4))
print("Reshaped: (3x4)")
print(myC)

myD = myB.ravel()
print("")
print("Back to array:")
print(myD)
```

The array:  
[ 0 1 2 3 4 5 6 7 8 9 10 11]

Reshaped (2x6):  
[[ 0 1 2 3 4 5]  
 [ 6 7 8 9 10 11]]

Reshaped: (3x4)  
[[ 0 1 2 3]  
 [ 4 5 6 7]  
 [ 8 9 10 11]]

Back to array:  
[ 0 1 2 3 4 5 6 7 8 9 10 11]

# Looping through arrays

`np.ndarray.flat`

returns an iterator,  
allowing to loop  
through the elements  
as if they were 1D.

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Matrix:

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

Looping through elements:

```
Element: 0
Element: 1
Element: 2
Element: 3
Element: 4
Element: 5
Element: 6
Element: 7
Element: 8
Element: 9
Element: 10
Element: 11
```

Looping row by row:

```
Row: [0 1 2] is a <class 'numpy.ndarray'>
      el: 0
      el: 1
      el: 2
Row: [3 4 5] is a <class 'numpy.ndarray'>
      el: 3
      el: 4
      el: 5
Row: [6 7 8] is a <class 'numpy.ndarray'>
      el: 6
      el: 7
      el: 8
Row: [ 9 10 11] is a <class 'numpy.ndarray'>
      el: 9
      el: 10
      el: 11
```

**NOTE:** although it is possible to iterate through the elements, numpy gives its best when we use vector/matrix operations.

```
import numpy as np
```

```
myA = np.arange(12)
```

```
print(myA)
```

```
print("")
```

```
print("Matrix:")
```

```
myA = myA.reshape((4,3))
```

```
print(myA)
```

```
print("Looping through elements:")# equivalent to:
```

```
for el in myA.flat:                # for el in myA.ravel():
    print("Element:", el)          #     print("Element:",el)
```

```
print("Looping row by row:")
```

```
for el in myA:
```

```
    print("Row: ", el, "is a", type(el))
```

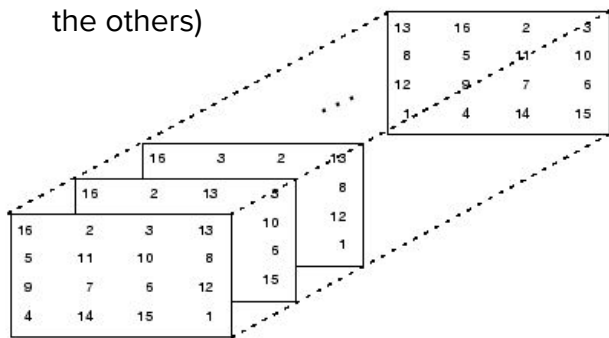
```
    for j in el:
```

```
        print("\tel:", j)
```



# N-Dimensions

numpy allows  
N-dimensional  
matrices (one  
stacked on top of  
the others)



Note that `np.ndarray[0,:,:]` is the whole first matrix. `np.ndarray[:,0,:]` is all the first rows, while `np.ndarray[:, :, 0]` is all the first columns. Regarding slicing and indexing, the same reasoning applies to n-dimensional matrices. For example, `myB` below is a 3x3x3 matrix.

```
[[3 7 9 3]
 [5 2 4 7]
 [6 8 8 1]]
```

```
myA[2,2] = 8
myA[1,3] = 7
myA[0,3] = 3
second row: [5 2 4 7]
```

```
3D matrix
- shape: (3, 3, 3)
[[[6 7 7]
  [8 1 5]
  [9 8 9]]
```

```
[[4 3 0]
 [3 5 0]
 [2 3 8]]
```

```
[[1 3 3]
 [3 7 0]
 [1 9 9]]]
```

```
myB[0,2,2] = 9
Second matrix:
[[4 3 0]
 [3 5 0]
 [2 3 8]]
Third row of second matrix:
[2 3 8]
Second column of second matrix:
[3 5 3]
```

```
import numpy as np

rng = np.random.default_rng()

myA = rng.integers(0,10, size = (3,4))
print(myA)
print("")
print("myA[2,2] = ", myA[2,2])
print("myA[1,3] = ", myA[1,3])
print("myA[0,3] = ", myA[0,3])
print("second row:", myA[1,:])
print("")
print("3D matrix ",)
myB = np.random.randint(0,10, size = (3,3,3))
print(" - shape:", myB.shape)
print(myB)
print("")
print("myB[0,2,2] = ", myB[0,2,2])
print("Second matrix:")
print(myB[1,:,:])
print("Third row of second matrix:")
print(myB[1,2,:])
print("Second column of second matrix:")
print(myB[1,:,1])
```



`np.ndarray[M, R, C]`

matrix (can be : for all)  
row (can be : for all)  
column (can be : for all)



# Operator broadcasting

Operator  
broadcasting  
works pretty  
much like with  
pandas  
DataFrames

```
B + C
[[[ 6  4  4]
  [10 11 11]
  [10 12  8]]

  [[ 5  8 13]
   [14  7  7]
   [11 13 13]]

  [[ 7 10 11]
   [ 8  6  9]
   [ 9 13  8]]]

B
[[[2 0 0]
  [4 5 5]
  [6 8 4]]

  [[1 4 9]
   [8 1 1]
   [7 9 9]]

  [[3 6 7]
   [2 0 3]
   [5 9 4]]]
Sub array B - 20
[[[-18 -20  0]
  [-16 -15  5]
  [ 6  8  4]]

  [[-19 -16  9]
   [-12 -19  1]
   [ 7  9  9]]

  [[-17 -14  7]
   [-18 -20  3]
   [ 5  9  4]]]
```

```
Matrix A 3x2
[[0 4]
 [7 3]
 [2 7]]
```

```
Matrix B 3x3x3
[[[2 0 0]
  [4 5 5]
  [6 8 4]]

  [[1 4 9]
   [8 1 1]
   [7 9 9]]

  [[3 6 7]
   [2 0 3]
   [5 9 4]]]
```

```
Matrix C 3x1
[[4]
 [6]
 [4]]
```

```
A squared
[[ 0 16]
 [49  9]
 [ 4 49]]
```

```
A square-rooted
[[ 0.          2.          ]
 [ 2.64575131  1.73205081]
 [ 1.41421356  2.64575131]]
```

```
B square-rooted
[[[ 1.41421356  0.          0.          ]
  [ 2.          2.23606798  2.23606798]
  [ 2.44948974  2.82842712  2.          ]]
```

```
[[[ 1.          2.          3.          ]
  [ 2.82842712  1.          1.          ]
  [ 2.64575131  3.          3.          ]]
```

```
[[[ 1.73205081  2.44948974  2.64575131]
  [ 1.41421356  0.          1.73205081]
  [ 2.23606798  3.          2.          ]]
```

```
A + C
[[ 4  8]
 [13  9]
 [ 6 11]]
```

```
import numpy as np
```

```
rng = np.random.default_rng()
A = rng.integers(0,10, size = (3,2))
B = rng.integers(0,10, size = (3,3,3))
C = rng.integers(0,10, size = (3,1))
print("Matrix A 3x2")
print(A)
print("")
print("Matrix B 3x3x3")
print(B)
print("")
print("Matrix C 3x1")
print(C)
print("")
print("A squared")
print(A**2)
print("")
print("A square-rooted")
print(np.sqrt(A))
```

```
print("")
print("B square-rooted")
print(np.sqrt(B))
```

```
print("A + C ")
print(A + C)
print("")
```

```
print("B + C ")
print(B + C)
```

```
print("")
print("B")
print(B)
print("Sub array B - 20")
B[:, 0:2 , 0:2 ] -= 20
print(B)
```

# Linear algebra

Linear algebra operations (e.g. matrix/vector multiplication, decompositions, inversions...) are implemented by the linalg module

NOTE:  
for some  
operations:

```
import numpy as np
from numpy import linalg
rng = np.random.default_rng()
A = rng.integers(0,10, size = (4,4))
print("Matrix A:")
print(A)
print("")
print("inv(A)")
A_1 = linalg.inv(A)
print(A_1)
print("")
print(np.dot(A,A_1))
```

Matrix A:  
[[3 2 1 7]  
[6 1 4 8]  
[9 0 0 8]  
[3 8 2 6]]

inv(A)  
[[-0.27777778 0.03703704 0.15740741 0.06481481]  
[-0.08333333 -0.05555556 0.01388889 0.15277778]  
[-0.1875 0.29166667 -0.13541667 0.01041667]  
[ 0.3125 -0.04166667 -0.05208333 -0.07291667]]

```
[[ 1.00000000e+00  3.46944695e-17 -2.77555756e-17 -6.93889390e-17]
 [ 0.00000000e+00  1.00000000e+00 -1.11022302e-16 -1.11022302e-16]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00 -1.11022302e-16]
 [ 0.00000000e+00  1.38777878e-17  0.00000000e+00  1.00000000e+00]]
```

```
import numpy as np

a = np.array([1, 2, 3, 1, 2, 3, 1, 1, 1])
A = a.reshape((3,3))
rng = np.random.default_rng()
B = rng.integers(0,10, size = (3,2))

print("A (3x3)")
print(A)
print("")
print("B (3x2)")
print(B)
print("")
print("AxB (3x2)")
print(A.dot(B))
print("")
print("A transposed:")
print(A.T)
```

A (3x3)  
[[1 2 3]  
[1 2 3]  
[1 1 1]]

B (3x2)  
[[4 3]  
[4 4]  
[8 4]]

AxB (3x2)  
[[36 23]  
[36 23]  
[16 11]]

A transposed:  
[[1 1 1]  
[2 2 1]  
[3 3 1]]

```
from numpy import linalg
```

# Filtering

It is possible to **filter** `np.ndarrays` to **retrieve the indexes** (or the values) meeting specific conditions. The method `where` **provides the index** of those values.

If the `np.ndarray` is a **matrix**, `where` **returns a tuple of indexes** that are respectively the **i and j coordinates of the elements fulfilling the condition**.

Note that in the code above, `np.all` tests if the two conditions are True at the same time (i.e. AND). If we want to test if at least one is True we use `np.any`.

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

```
A = np.arange(-2* np.pi, 2*np.pi, 0.01)
```

```
sA = np.sin(A)
cA = np.cos(A)
```

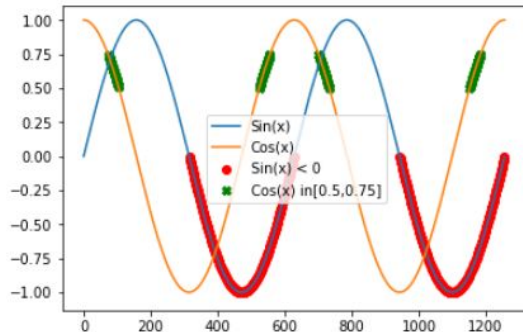
```
plt.plot(sA)
plt.plot(cA)
```

```
s0A_Y = sA[sA < 0]
s0A_X = np.where(sA < 0)
```

```
c0A_Y = cA[np.all( [cA > 0.5 , cA < 0.75], axis = 0)]
c0A_X = np.where(np.all( [cA > 0.5 , cA < 0.75], axis = 0))
```

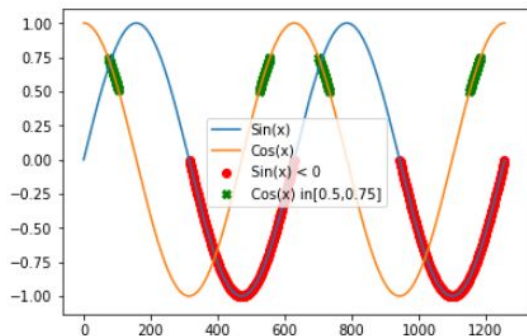
```
plt.scatter(s0A_X,s0A_Y, marker='o', c = 'red')
plt.scatter(c0A_X,c0A_Y, marker='x', c = 'green')
```

```
plt.legend(["Sin(x)", "Cos(x)", "Sin(x) < 0", "Cos(x) in[0.5,0.75]"])
plt.show()
```



# Any / All

Note that in the code above, `np.all` tests if the two conditions are True at the same time (i.e. AND). If we want to test if at least one is True we use `np.any`.



```
import numpy as np

v1 = [True, False, False, True]
v2 = [False, False, True, True]
v3 = [False, False, False, True]
print("vals:")
vals = np.array([v1,v2,v3])
print(vals)
print("\nANY(vals):")
print(np.any(vals, axis=0))
print("\nANY(vals) on rows:")
print(np.any(vals, axis=1))
print("\nALL(vals):")
print(np.all(vals, axis=0))

print("\nALL(vals) on rows:")
print(np.all(vals, axis=1))
```

```
vals:
[[ True False False  True]
 [False False  True  True]
 [False False False  True]]
```


```
ANY(vals):
[ True False  True  True]
```

```
ANY(vals) on rows:
[ True  True  True]
```

```
ALL(vals):
[False False False  True]
```

```
ALL(vals) on rows:
[False False False]
```

# <https://numpy.org/doc/stable/reference/index.html>

 NumPy

[NumPy.org](#) [Docs](#) [NumPy v1.19 Manual](#)

[index](#) [next](#) [previous](#)

## Table of Contents

- [NumPy Reference](#)
  - [Acknowledgements](#)

## Previous topic

[How to read and write data using NumPy](#)

## Next topic

[Array objects](#)

## Quick search

## NumPy Reference

**Release:** 1.19  
**Date:** June 29, 2020

This reference manual details functions, modules, and objects included in NumPy, describing what they are and what they do. For learning how to use NumPy, see the [complete documentation](#).

- [Array objects](#)
  - [The N-dimensional array \(ndarray\)](#)
  - [Scalars](#)
  - [Data type objects \(dtype\)](#)
  - [Indexing](#)
  - [Iterating Over Arrays](#)
  - [Standard array subclasses](#)
  - [Masked arrays](#)
  - [The Array Interface](#)
  - [Datetimes and Timedeltas](#)
- [Constants](#)
- [Universal functions \(ufunc\)](#)
  - [Broadcasting](#)
  - [Output type determination](#)
  - [Use of internal buffers](#)
  - [Error handling](#)
  - [Casting Rules](#)
  - [Overriding Ufunc behavior](#)
  - [ufunc](#)
  - [Available ufuncs](#)
- [Routines](#)
  - [Array creation routines](#)
  - [Array manipulation routines](#)
  - [Binary operations](#)
  - [String operations](#)
  - [C-Types Foreign Function Interface \(numpy.ctypeslib\)](#)
  - [Datetime Support Functions](#)
  - [Data type routines](#)
  - [Optionally Scipy-accelerated routines \(numpy.dual\)](#)
  - [Mathematical functions with automatic domain \(numpy.emath\)](#)
  - [Floating point error handling](#)
  - [Discrete Fourier Transform \(numpy.fft\)](#)

## Exercises

1. Write a function that converts a numpy ndarray of temperatures expressed in Degrees Celsius into Degrees Fahrenheit. The formula to convert a temperature C in Celsius into F in Fahrenheit is the following:

$$F = C * 9/5 + 32$$

Write then a function that converts a numpy ndarray of temperatures in Fahrenheit into Celsius.

Finally:

1. apply the Celsius to Fahrenheit conversion on an ndarray containing the following October's minimum and maximum temperatures in Trento: `tmin = [12, 11, 11, 8, 9, 10, 3, 8, 4, 5, 10, 9, 8, 9, 8, 7, 6, 4, 5, 6, 9, 9, 3, 3, 5]` and `tmax = [15, 22, 18, 20, 22, 22, 20, 21, 21, 21, 21, 23, 24, 24, 24, 25, 22, 22, 20, 20, 19, 15, 20, 23, 19]`;
2. check that both functions work correctly by converting the values from Celsius to Fahrenheit and back to Celsius;
3. plot the minimum and maximum temperatures in celsius on the same graph. Since the temperatures refer to the first 25 days of the month of October 2017, the x coordinate can be a `range(1,26)`;

Show/Hide Solution

2. Create the following functions:

- a. `createRandomList` : with parameters, N, min, max. Creates a list of N random integers ranging from min to max;
- b. `getIdential` : with parameters two lists of integers L1 and L2 having the same size. It returns the list of indexes I where `L1[I] == L2[I]`
- c. `check` : gets lists L1, L2, identities (as computed by `getIdential`) and a number N and prints if the first N and last N values in identities correspond to indexes of identical values in L1 and L2;
- d. implement `getIdential` using `numpy.ndarrays`. Call it `getIdentialNpy` (hint: subtract the two arrays and find zeros).

Test the software creating two lists of 100,000 random numbers from 0 to 10. You should get