

Scientific Programming

Practical 9

Introduction

Luca Bianco - Academic Year 2019-20
luca.bianco@fmach.it

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(Afloat))
print(Afloat.shape)
print(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. ]]
<class 'numpy.ndarray'>
(3, 3)
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

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 array:

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

The shape: (2, 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)]  
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("")  
twoDarray = np.array([mysqrt, mycrt])  
print("The 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).

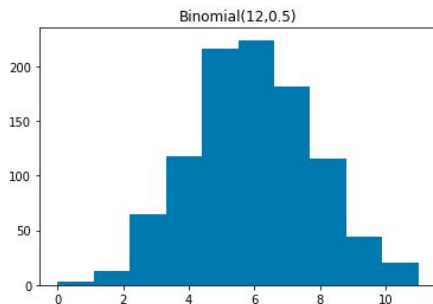
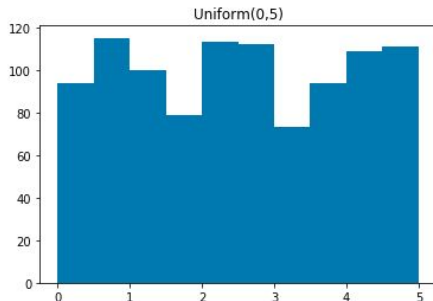
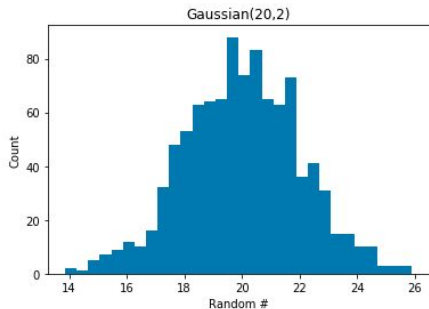
```
myRange = np.linspace(-5,2.5,num =6)
print("6 linearly spaced elements in [-5 - 2.5]:")
print(myRange)

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

```
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

%matplotlib inline

#get the gaussian random array
g = np.random.normal(20,2, 1000)
#a uniform random array with vals in [0,5]
u = np.random.uniform(0,5, 1000)
#get the binomial random array
b = np.random.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://docs.scipy.org/doc/numpy-1.16.0/reference/routines.random.html#distributions>

random seed

Random values...

... are they really random?



```
import numpy as np
|
u = np.random.uniform(0,1,size=(3,2))
u1 = np.random.uniform(0,1,size=(3,2))
u2 = np.random.uniform(0,1,size=(3,2))

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

print("")
print("With random seed reinit.")
np.random.seed(0)
u = np.random.uniform(0,1,3)
np.random.seed(0)
u1 = np.random.uniform(0,1,3)
np.random.seed(0)
u2 = np.random.uniform(0,1,3)

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

```
u: [[0.54488318 0.4236548 ]
     [0.64589411 0.43758721]
     [0.891773   0.96366276]]

u1:[[0.38344152 0.79172504]
     [0.52889492 0.56804456]
     [0.92559664 0.07103606]]

u2:[[0.0871293  0.0202184 ]
     [0.83261985 0.77815675]
     [0.87001215 0.97861834]]

With random seed reinit.
u: [0.5488135  0.71518937 0.60276338]
u1:[0.5488135  0.71518937 0.60276338]
u2:[0.5488135  0.71518937 0.60276338]
```


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(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)
```



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(10)

print("The array:")
print(myA)
print("")
myB = myA.reshape((2,5))
print("Reshaped:")
print(myB)

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

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

Reshaped:
[[0 1 2 3 4]
 [5 6 7 8 9]]

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

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

```
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:
```

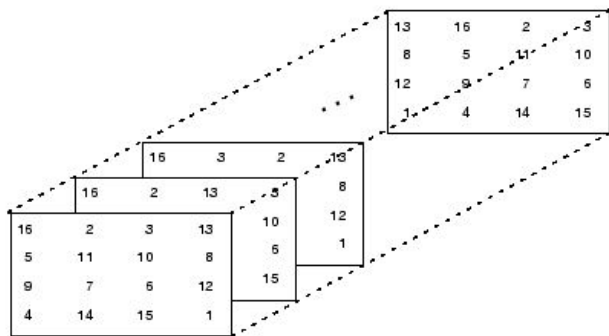
```
    print("Row: ", el, "is a", type(el))
```

```
    for j in el:
```

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

N-Dimensions

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.



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

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

```
[[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
```

```
myA = np.random.randint(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])
```



Operator broadcasting

```

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
```

```

A = np.random.randint(0,10, size = (3,2))
B = np.random.randint(0,10, size = (3,3,3))
C = np.random.randint(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

```
import numpy as np
from numpy import linalg

A = np.random.randint(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 7 5 5]
 [0 1 5 9]
 [3 0 5 0]
 [1 2 4 2]]
```

inv(A)

```
[[ 0.23121387  0.05780347  0.38150289 -0.83815029]
 [ 0.09537572 -0.10115607 -0.16763006  0.21676301]
 [-0.13872832 -0.03468208 -0.02890173  0.50289017]
 [ 0.06647399  0.1416185   0.03468208 -0.30346821]]
```

```
[[ 1.00000000e+00  0.00000000e+00  2.77555756e-17  0.00000000e+00]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00]
 [-2.77555756e-17  0.00000000e+00  6.93889390e-17  1.00000000e+00]]
```

```
import numpy as np
```

```
a = np.array([1, 2, 3, 1, 2, 3,1, 1 ,1])
A = a.reshape((3,3))
B = np.random.randint(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]]
```

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**.

```
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)

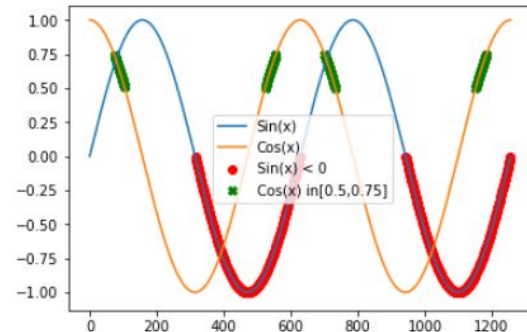
s0A_Y = sA[sA < 0]
s0A_X = np.where(sA < 0)

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')

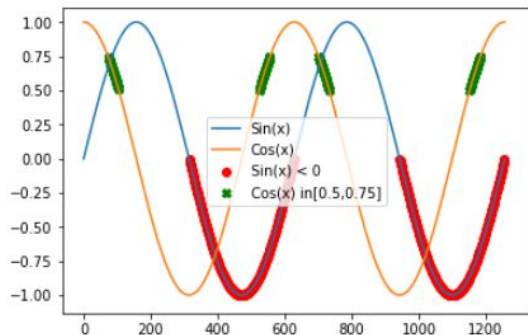
plt.legend(["Sin(x)", "Cos(x)", "Sin(x) < 0", "Cos(x) in[0.5,0.75]"])
plt.show()
```

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`.



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]
print("v1:")
print(v1)
print("v2:")
print(v2)
print("\nANY(v1,v2):")
print(np.any([v1,v2], axis=0))
print("\nALL(v1,v2):")
print(np.all([v1,v2], axis=0))
```

```
v1:
[True, False, False, True]
v2:
[False, False, True, True]
```

```
ANY(v1,v2):
[ True False  True  True]
```

```
ALL(v1,v2):
[False False False  True]
```

<https://docs.scipy.org/doc/numpy-1.16.1/reference/>

 SciPy.org  Sponsored by
ENTHOUGHT

SciPy.org Docs NumPy v1.16 Manual

index next previous

NumPy Reference

Release: 1.16
Date: January 31, 2019

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 also [NumPy User Guide](#).

- [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](#)

Table Of Contents

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

Previous topic

[Beyond the Basics](#)

Next topic

[Array objects](#)

Quick search

Exercises

1. Create the following functions:

- a. `createRadoList` : 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.

Show/Hide Solution

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

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

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