GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung

Data Literacy

Tutor : Ghassan Al-Falouji
Supervisor : Prof. Dr. Roland Mandl

AUFSTIEG DURCH BILDUNG
OFFENE HOCHSCHULEN

In [1]:
```
## CSS coloring for the dataframe tables

from IPython.core.display import HTML

css = open('../style/style-table.css').read() + open('../style/style-no
tebook.css').read()
HTML('<style>{}</style>'.format(css))
```
Out[1]:

Introduction to NumPy [[1](#ref1)]

Table of contents

Introduction[[1](ref1)]

The first step in data analysis (after obtaining correct and descriptive data) is efficiently loading, manipulating data sets. No matter what the data are (weather they are sensor measurements, images, texts, ..), the first step to make them analyzable will be to transform them into arrays of numbers. For this reason, efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science. `NumPy` (abbreviation of **Numerical python**) is a specialized Python tool (beside other tools like `Pandas`) for handling such numerical arrays.

NumPy (short for Numerical Python) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python's built-in list type, but NumPy arrays provide **much more efficient storage** and **data operations** as the arrays grow larger in size.

Installing and importing NumPy module

- To install in a command termianl/command-line, run the following shell/bash code to install (if not already installed) and update the numpy module if already installed.

```
pip install -U numpy
```

also you can do this in the jupyter notebook:

In [4]:
```
!pip install -U numpy
```

```
Collecting numpy
  Downloading https://files.pythonhosted.org/packages/cb/41/05fbf6944b09
8eb9d53e8a29a9dbfa20a7448f3254fb71499746a29a1b2d/numpy-1.17.1-cp37-cp37m
-win_amd64.whl (12.8MB)
Installing collected packages: numpy
  Found existing installation: numpy 1.17.0
    Uninstalling numpy-1.17.0:
```

```
ERROR: Could not install packages due to an EnvironmentError: [WinError
5] Access is denied: 'c:\\program files\\python37\\lib\\site-packages\\n
umpy-1.17.0.dist-info\\entry_points.txt'
Consider using the `--user` option or check the permissions.

WARNING: You are using pip version 19.2.1, however version 19.2.3 is ava
```

```
ilable.
You should consider upgrading via the 'python -m pip install --upgrade p
ip' command.
```

- Importing NumPy module and checking its version:

In [5]:
```
import numpy as np
np.__version__
```

Out[5]:
```
'1.17.0'
```

Reminder about Built-in documentation

- to display all the contents of the NumPy namespace:

```
np.<TAB>
```

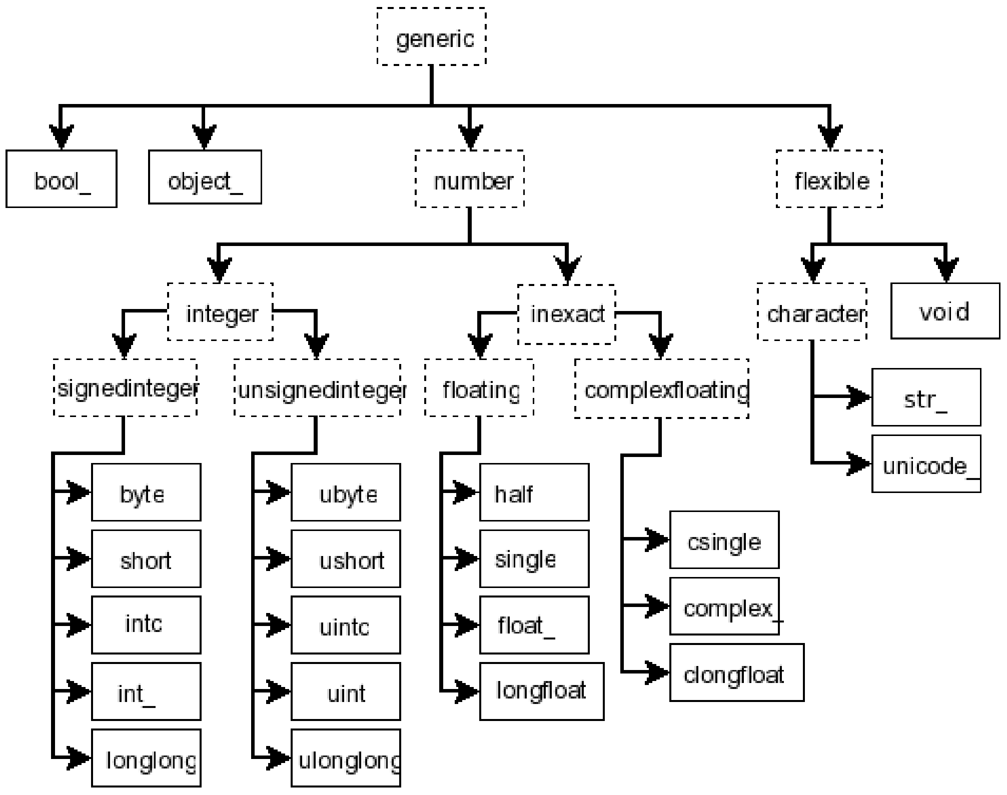
- benefiting the built-in functionality of iPython, it's possible to display the namespace/class/method documentation using `?` operator. As example:

In [6]:
```
np?
```

Scalar types in NumPy[[2](#ref2)]

Effective data-driven science and computation requires understanding how data is stored and manipulated [[1](#ref1)]. The figure below displays the hierarchy of type objects representing the array data types.

source: *https://bit.ly/30Cts38*



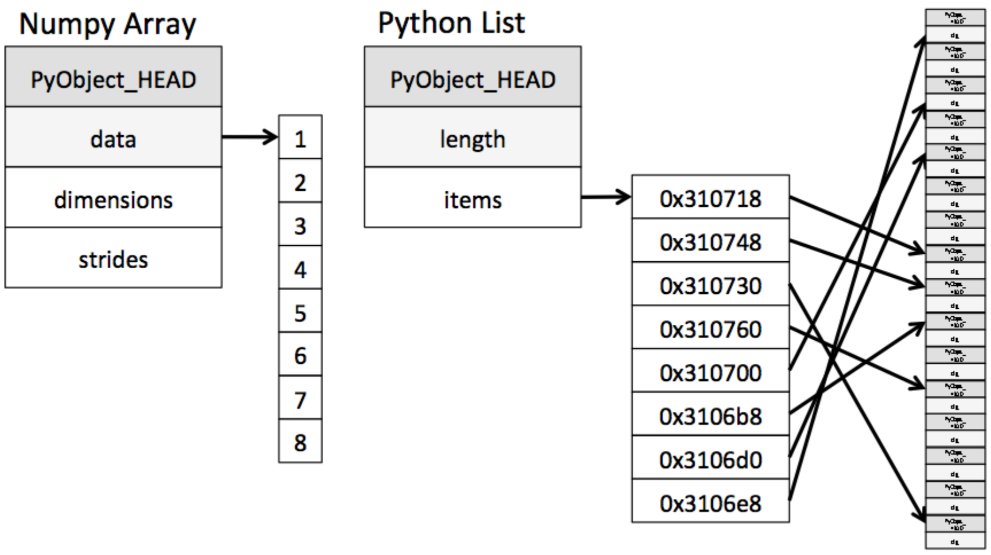
source: [1]

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

NumPy Arrays[[1](#ref1)]

An alternative to Python Lists in NumPy arrays, but not like the Python lists the NumPy array data-structure contains more properties (see next figure) which adds flexibility to them.

source: [1]



Creating NumPy array

form Python lists

- `np.array` can be used to create arrays from Python lists:

```
In [7]: # homogeneous integer array
        a = np.array([1,2,3,4,])
```

```
In [8]: print("a of type {} : {}".format(type(a), a))
```

```
a of type <class 'numpy.ndarray'> : [1 2 3 4]
```

BUT not like python lists, an array contain only homogeneous data types (called `dtype` in numpy):

```
In [9]: # Heterogenous python list
        a = [1, True, "A"]
        print("a of type {} : {}".format(type(a), a))
```

```
a of type <class 'list'> : [1, True, 'A']
```

In [10]:
```
b = np.array(a)
print("b of type {} : {}".format(type(b), b))
```
```
b of type <class 'numpy.ndarray'> : ['1' 'True' 'A']
```

Notice that the array `a` was automatically converted into string type.

In [11]:
```
[type(x) for x in b]
```
Out[11]:
```
[numpy.str_, numpy.str_, numpy.str_]
```

`from scratch`

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

In [12]:
```
# Create a length-10 integer array filled with zeros
np.zeros(10, dtype=int)
```
Out[12]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [13]:
```
# Create a 3x5 floating-point array filled with ones
np.ones((3, 5), dtype=float)
```
Out[13]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

In [14]:
```
# Create a 3x5 array filled with pi
#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------
```
Out[14]:
```
array([[3.14159265, 3.14159265, 3.14159265, 3.14159265, 3.14159265],
       [3.14159265, 3.14159265, 3.14159265, 3.14159265, 3.14159265],
       [3.14159265, 3.14159265, 3.14159265, 3.14159265, 3.14159265]])
```

In [15]:
```
# Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range() function)
#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------
```
Out[15]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [16]:
```
# Create an array of five values evenly spaced between 0 and 1
#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------
```
Out[16]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

In [17]:
```
# Create a 3x3 array of uniformly distributed
# random values between 0 and 1
np.random.random((3, 3))
```
Out[17]:
```
array([[0.36268124, 0.37637567, 0.96193164],
       [0.74630107, 0.8819328 , 0.87679024],
       [0.97830129, 0.11491992, 0.9463825 ]])
```

In [18]:
```
# Create a 3x3 array of normally distributed random values
# with mean 0 and standard deviation 1
#----------------------------------------------------- CODE HERE ---
---------------------------------------------------
```

Out[18]:
```
array([[-0.53230229, -0.52458505,  0.41418268],
       [ 0.44045286,  0.20922206, -0.69587414],
       [-0.38139806, -0.25359046,  0.76716418]])
```

In [19]:
```
# Create a 3x3 array of random integers in the interval [0, 10)
#----------------------------------------------------- CODE HERE ---
---------------------------------------------------
```

Out[19]:
```
array([[5, 8, 4],
       [5, 0, 7],
       [7, 8, 7]])
```

In [20]:
```
# Create a 3x3 identity matrix
np.eye(3)
```

Out[20]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [21]:
```
# Create an uninitialized array of three integers
# The values will be whatever happens to already exist at that memory lo
cation
np.empty(3)
```

Out[21]:
```
array([1., 1., 1.])
```

Basic array manipulators

- Creating two dimensional numpy array:

In [22]:
```
x = np.array([[0,1,2,3,4],
              [5,6,7,8,9],
              [10,11,12,13,14],
              [15,16,17,18,19]])

print(x)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
```

NumPy array attributes

NumPy's array class is called `ndarray`. It is also known by the alias array and referred as `numpy.array` . Some attributes of an `ndarray` object are:

- `ndarray.ndim` : represents the number of dimensions (axis) of an array. It is also known as **rank** of an array.
- `ndarray.shape` : represents the dimensions of the array. For $N$-rows and $M$-columns matrix, the shape is $(n,m)$, and the **rank** (`ndim`) is $2$
- `ndarray.size` : The total number of elements in an array. the `size` of an (m,n) matrix is $m*n$
- `ndarray.dtype` : displays the type of the elements of the array.
- `ndarray.itemsize` : the size in bytes used by **each** element of an array.

In [43]:
```
print("x.ndim = {}".format(#-------------------------- CODE HERE ----
----------------------))
print("x.shape = {}".format(#-------------------------- CODE HERE ---
-----------------------))
print("x.size = {}".format(#-------------------------- CODE HERE ----
----------------------))
print("x.dtype = {}".format(#-------------------------- CODE HERE ---
-----------------------))
print("x.itemsize = {} Bytes".format(#-------------------------- CODE
HERE -------------------------))
```

```
x.ndim = 2
x.shape = (4, 5)
x.size = 20
x.dtype = int32
x.itemsize = 4 Bytes
```

- The type of the array can be specified at creation time also:

In [42]:
```
#---------------------------------------------------- CODE HERE ---
-------------------------------------------------

print(c)
```

```
[[6.+0.j 7.+0.j]
 [8.+0.j 9.+0.j]]
```

NumPy array indexing

If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. Likewise Python, NumPy indexing starts from $0$.

1D array

In [49]:
```
# Create a list
x = np.random.randint(1,10,size=10)
print(x)
```

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

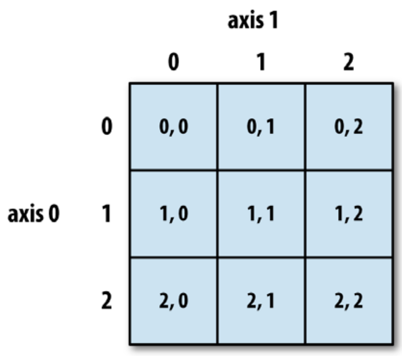
In [50]:
```
# Get the first element
#------------------------------------------------------- CODE HERE ---
-----------------------------------------------------
```
Out[50]:
```
1
```

In [51]:
```
# Get the last element
#------------------------------------------------------- CODE HERE ---
-----------------------------------------------------
```
Out[51]:
```
2
```

In [52]:
```
# Get before the last element
#------------------------------------------------------- CODE HERE ---
-----------------------------------------------------
```
Out[52]:
```
7
```

2D array

source: https://bit.ly/2LdPzGU



In [53]:
```
# Create 2D array of: row# = 5, column#= 6
#------------------------------------------------------- CODE HERE ---
-----------------------------------------------------
print(x)
```
```
[[ 4 19 10  5  3 19]
 [ 4  9 18  1 13  6]
 [ 2  9 18 16  9 11]
 [13 17 10 18 16 12]
 [ 6  6  4 16  8 13]]
```

In [54]:
```
# Get row 1
x[1]
```
Out[54]:
```
array([ 4,  9, 18,  1, 13,  6])
```

In [56]:
```
# Get element of row 1 and column 2
x[1,2]
```
Out[56]:
```
18
```

In [58]:
```
# replace row 0 and column 0 element with 0
x[0,0] = 0
```

```
print(x)
```

```
[[ 0 19 10  5  3 19]
 [ 4  9 18  1 13  6]
 [ 2  9 18 16  9 11]
 [13 17 10 18 16 12]
 [ 6  6  4 16  8 13]]
```

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

In [59]:
```
x[0,0] = np.pi
print(x)
```

```
[[ 3 19 10  5  3 19]
 [ 4  9 18  1 13  6]
 [ 2  9 18 16  9 11]
 [13 17 10 18 16 12]
 [ 6  6  4 16  8 13]]
```

3D array

In [60]:
```
# Create 3D array of: depth = 3, row# = 5, column#= 6
#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------
print(x)
```

```
[[[19 14 15 12 15  9]
  [11 12  9 14  1 19]
  [14 19  1  2  5 18]
  [18  4 19  1  5 16]
  [11 10 15  7  7  2]]

 [[ 1 13  6  1  5 14]
  [ 5  6 12 14 11 18]
  [19  1  1  9  5 10]
  [ 1 16  4  9  1  8]
  [ 8  3  6 18  4 19]]

 [[ 8  9  8 12 13 10]
  [11 19 18 18 15  3]
  [13 16 18  8 11  4]
  [14  6 17  1  9 11]
  [ 5  1  3 12  2 19]]]
```

Notice that the indexing of a 3D multidimensional array is : (*depth, row, column*)

In [61]:
```
# Getting layer 1
print(x[1])
```

```
[[ 1 13  6  1  5 14]
 [ 5  6 12 14 11 18]
 [19  1  1  9  5 10]
 [ 1 16  4  9  1  8]
 [ 8  3  6 18  4 19]]
```

In [62]:
```
# Get the element is layer 2, row 0 and column 1
print(x[2,0,1])
```

```
9
```

`numpy.where()`

Is a numpy method that enables the selection of elements based on condition.

```
pyhton
numpy.where(condition,[,x,y])
```

Arguments:

- **condition**: A Conditional expression
- *(optional)* **x,y**: Arrays from which to choose.
  - If all arguments are passed (**condition** , **x** & **y**): it will return elements selected from x & y depending on values in bool array yielded by condition. All 3 arrays must be of <u>same size</u>.
    Every True-condition will be taken from **x** otherwise from **y**
  - If x & y arguments are not passed and only condition argument is passed then it returns a tuple of arrays (one for each axis) containing the indices of the elements that are True in bool numpy array returned by condition.

As example:

In [158]:
```
# call numpy.where() with 3 arguments

#--------------------------------------------------------- CODE HERE ---
-----------------------------------------------------

# Take every true from x otherwise from y
result
```

Out[158]:
```
array([1, 8, 9])
```

Another example:

In [160]:
```
arr = np.array([11, 12, 13])

#--------------------------------------------------------- CODE HERE ---
-----------------------------------------------------

result
```

Out[160]:
```
array([1, 8, 9])
```

- When only passing the condition:

In [166]:
```
arr = np.array([11, 12, 13, 14, 15, 16, 17, 15, 11, 12, 14, 15, 16, 17]
)

#--------------------------------------------------------- CODE HERE ---
-----------------------------------------------------
```

```
print(indx)
```

```
(array([ 2,  3,  4,  7, 10, 11], dtype=int64),)
```

- If you have 2D array:

In [175]:
```
a=np.arange(1,13).reshape([3,-1])
print("a:\n",a)
#

#------------------------------------------------------- CODE HERE ---
----------------------------------------------------

print("row index: ",row_indx)
print("col index:",col_indx)
```

```
a:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
row index:  [1 1]
col index: [1 2]
```

In [179]:
```
# To get the result in 2D form indeces

#------------------------------------------------------- CODE HERE ---
----------------------------------------------------

print("index: \n", indx)
```

```
index: 
 [[1 1]
 [1 2]]
```

NumPy array slicing [[1](#ref1)]

1D array

In [63]:
```
x = np.arange(10)
x
```

Out[63]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [64]:
```
# Get the first 5 elements
x[:5]
```

Out[64]:
```
array([0, 1, 2, 3, 4])
```

In [65]:
```
# Get the last 5 elements
x[5:]
```

Out[65]:
```
array([5, 6, 7, 8, 9])
```

In [66]:
```
# Get the elements from index 1 (inclusive) to 5 (exclusive)
x[1:5] # x[start_inclusive : stop_exclusive]
```
Out[66]:
```
array([1, 2, 3, 4])
```

In [67]:
```
# Get the elements with even index
# (step 2 starting from index 0)
x[::2]
```
Out[67]:
```
array([0, 2, 4, 6, 8])
```

In [68]:
```
# Get the elements with the odd index
# (step 2 starting from index 0)
x[1::2] # x[start_index :: step_size]
```
Out[68]:
```
array([1, 3, 5, 7, 9])
```

In [69]:
```
# Get elements with step 3 starting from index 1
x[1::3]
```
Out[69]:
```
array([1, 4, 7])
```

- Reversed slicing

When the *step_size* is negative, this means swap (reverse) the array

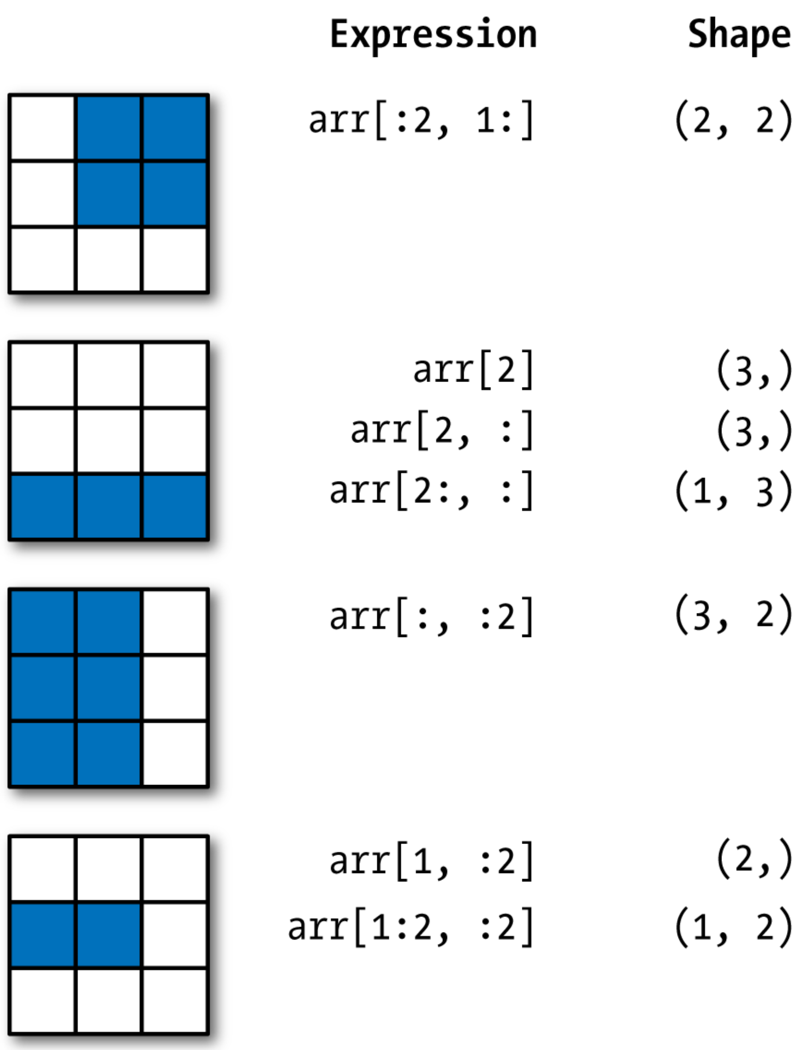
In [70]:
```
# Get all elements or the array reversed
x[::-1]
```
Out[70]:
```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

In [71]:
```
# Get the elements reversed starting from element index=5 with step 1
x[5::-1]
```
Out[71]:
```
array([5, 4, 3, 2, 1, 0])
```

In [72]:
```
# Get the elements reversed starting from element index=5 with step 2
x[5::-2]
```
Out[72]:
```
array([5, 3, 1])
```

Multi-dimensional array

source: https://bit.ly/2UeB3Tb



In [76]:
```python
# Create a two dimensional array
x = np.random.randint(1,10,size=[4,6])
x
```

Out[76]:
```
array([[2, 2, 7, 2, 9, 5],
       [9, 3, 9, 6, 4, 3],
       [1, 9, 5, 1, 6, 7],
       [3, 1, 6, 5, 2, 5]])
```

In [77]:
```python
# To get row index 1
x[1]
```

Out[77]:
```
array([9, 3, 9, 6, 4, 3])
```

In [80]:
```python
# To get column index 1
print(x[:,1])
print((x[:,1]).shape)
```

```
[2 3 9 1]
(4,)
```

> Notice that the returned sliced column was returned as a horizontal list (let's call it **vector**)

If we attempted to transpose a vector:

In [84]:
```
x_1 = x[:,1].T
print(x_1)
print(x_1.shape)
```

```
[2 3 9 1]
(4,)
```

> Notice that NOTING happend !!

Python consider a 1D array is always a vector with one dimension, therefore you see in `.shape` that the returned value a tuple that contains only one element. Therefore to make operations -which is 2D specific- like transpose on 1D array, you have to convert the 1D array into 2D using `[np.newaxis]` or using `np.reshape`, as follows:

In [87]:
```
#-------------------------------------------------------- CODE HERE ---
----------------------------------------------------

print(temp)
print(temp.shape)
```

```
[[2 3 9 1]]
(1, 4)
```

In [91]:
```
#-------------------------------------------------------- CODE HERE ---
----------------------------------------------------

print(temp)
print(temp.shape)
```

```
[[2 3 9 1]]
(1, 4)
```

Also, using reshape you can give one of the desired dimension, while the other dimension is automatically calculated, then use $-1$ for the axis where it should be automatically calculated:

In [92]:
```
#-------------------------------------------------------- CODE HERE ---
----------------------------------------------------

print(temp)
print(temp.shape)
```

```
[[2 3 9 1]]
(1, 4)
```

In [95]:
```
# Get the sub-matrix fo the first 3 rows and the first 3 columns
x[:3, :3] # or x[0:3, 0:3]
```

Out[95]:
```
array([[2, 2, 7],
       [9, 3, 9],
       [1, 9, 5]])
```

```
In [94]: # Get the last 3 rows and the first 3 columns
         x[-3::, :3]
```

```
Out[94]: array([[9, 3, 9],
                [1, 9, 5],
                [3, 1, 6]])
```

Exercise

1. Create a two dimensional array (size=[5,6]) with uniformly random elements.

```
In [ ]: ### YOUR CODE HERE
        #
        #
        ###
```

1. Slice the matrix returning the rows with even index of the last 3 columns

```
In [ ]: ### YOUR CODE HERE
        #
        #
        ###
```

1. Slice the last 3 rows, and the first 2 odd columns index, and return the columns in reversed (column 3 then column 1)

```
In [ ]: ### YOUR CODE HERE
        #
        #
        ###
```

Deleting a subarray

This can be done using two ways:

1. Boolean masking $\leftarrow $ is a preferred method
2. `numpy.delete()`

1. Using Boolean masking

In [151]:
```
x = np.random.randint(0,10,size=(5,6))
x
```

Out[151]:
```
array([[9, 7, 5, 4, 0, 1],
       [5, 9, 6, 7, 7, 2],
       [7, 7, 7, 3, 7, 6],
       [7, 5, 2, 0, 2, 2],
       [2, 3, 7, 2, 6, 5]])
```

In [154]:
```
# Create a mask that delete (mask the first and last rows)

#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------


print(row_mask)
# replace the frist and last will false
row_mask[[0,-1]] = False
print(row_mask)
```

```
[ True  True  True  True  True]
[False  True  True  True False]
```

In [155]:
```
result = x[row_mask]
print(result)
```

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

Exercise

1. Create a (5,6) matrix with integer contents or your choice

In [2]:
```
import numpy as np
```

In [ ]:
```
### YOUR SOLUTION
#
#
###
```

1. Using boolean masking, delete all the elements of row_index ($1$) as well as column_index ($4$)

hint : you may need to using the numpy function ``unique``

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

2. Using `numpy.delete()`

In [143]:
```
x = np.random.randint(0,10,size=(5,6))
print(x)
print("size= ",x.shape)
```

```
[[3 6 7 1 9 8]
 [6 3 7 4 4 4]
 [9 0 1 7 2 7]
 [2 3 9 3 9 9]
 [3 7 0 8 1 0]]
size=  (5, 6)
```

In [144]:
```
# Delete the last row
np.delete(x,-1, axis=0) # axis(0) will represent the row since
                        # it is a two dimensional array
```

Out[144]:
```
array([[3, 6, 7, 1, 9, 8],
       [6, 3, 7, 4, 4, 4],
       [9, 0, 1, 7, 2, 7],
       [2, 3, 9, 3, 9, 9]])
```

In [145]:
```
print(x)
print("size= ",x.shape)
```

```
[[3 6 7 1 9 8]
 [6 3 7 4 4 4]
 [9 0 1 7 2 7]
 [2 3 9 3 9 9]
 [3 7 0 8 1 0]]
size=  (5, 6)
```

> Notice that the original matrix is not affected, and only a copy was created and operated.
>
> > Compare with Shallow and deep copy

In [146]:
```
# Delete the last column
np.delete(x, -1, axis=1) # axis (1) will represent the column in
                        # two dimensional array
```

Out[146]:
```
array([[3, 6, 7, 1, 9],
       [6, 3, 7, 4, 4],
       [9, 0, 1, 7, 2],
       [2, 3, 9, 3, 9],
       [3, 7, 0, 8, 1]])
```

Exercise

1. Create a two dimensional array of size $(5,6)$ containing random integers having values between $[0,10]$.

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

1. Delete row 2 and columns (2, 3)

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

Shallow- and deep copy

One important–and extremely useful–thing to know about array slices is that they return *views* rather than *copies* of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Consider our two-dimensional array from before:

In [117]:
```
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x2
```

Out[117]:
```
array([[9, 8, 4, 6],
       [5, 7, 4, 9],
       [4, 6, 2, 6]])
```

Let's extract $2*2$ subarray:

In [118]:
```
#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------

print(x2_sub)
```

```
[[9 8]
 [5 7]]
```

Now if we modify this subarray, we'll see that the original array is changed! Observe:

In [119]:
```
#---------------------------------------------------------- CODE HERE ---
```

```
-------------------------------------------------------

print(x2_sub)
```

```
[[99  8]
 [ 5  7]]
```

In [120]:
```
print(x2)
```

```
[[99  8  4  6]
 [ 5  7  4  9]
 [ 4  6  2  6]]
```

> This *shallow copy* (default behavior) is actually quite **useful**: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

You can create a _deep__ copy using `.copy()` :

In [123]:
```
#  Create a deep copy

#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------

x2_sub
```

Out[123]:
```
array([[99,  8],
       [ 5,  7]])
```

In [124]:
```
x2_sub[0,0] = 0
print(x2_sub)
print()
print(x2)
```

```
[[0 8]
 [5 7]]

[[99  8  4  6]
 [ 5  7  4  9]
 [ 4  6  2  6]]
```

Exercise

Remember that an image is just a matrix with three layers:

- layer 1 : red
- layer 2: green
- layer 3: blue

You can convert an image to a numpy matrix using:

```
from skimage import io
img_dir = "./images/vegitables.jpg"
# Convert
img_mat = io.imread(img_dir)
```

To show your 3D matrix as an image:

```
import matplotlib.pylot as plt
plt.imshow(img_mat)
```

Use the previous tips to import the `vegitables.jpg` image, and show it again after making the following processes:

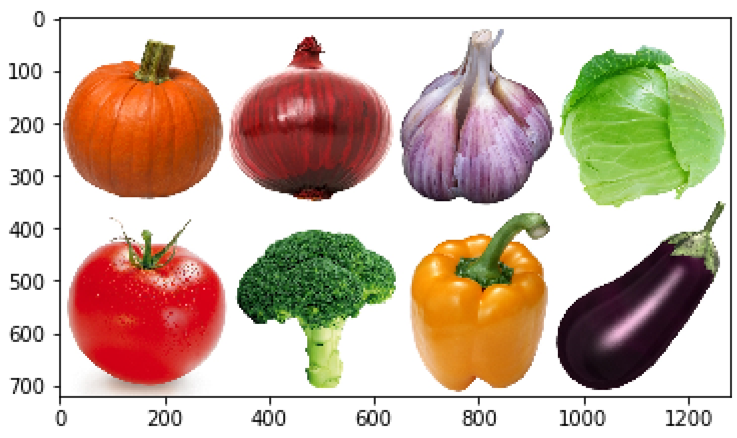
> NOTE : You may need to install `scikit-image` package

```
In [ ]: !pip install -U scikit-image
```

```
In [187]: from skimage import io # you must install the package: scikit-image
          import matplotlib.pyplot as plt
```

```
In [185]: img_dir = "./images/vegitables.jpg"
          # Convert
          img_mat = io.imread(img_dir)
          img_mat.shape

Out[185]: (720, 1280, 3)
```

```
In [191]: plt.imshow(img_mat);
```



1. flip the image up-side down

```
In [ ]: ### YOUR CODE HERE
        #
        #
        ###
```

1. Get only the tomato image

hint: Notice how the image dimensions are presented.

```
In [ ]: ### YOUR CODE HERE
        #
        #
        ###
```

Reshaping

`numpy.reshape()` is a flexible way for changing the shape of a numpy array.

As example:

```
In [3]: import numpy as np
```

```
In [4]: a=np.arange(1,10)
        print(a)
        print('---')

        #---------------------------------------------------------- CODE HERE ---
        ------------------------------------------------------
```

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

Also, reshape can be used to changing 1D array into two dimensional row or column array. As example:

```
In [8]: a = np.arange(6)
        print(a)
        print(a.shape) # One dimensional array
```

```
[0 1 2 3 4 5]
(6,)
```

- Converting to 2D array (row or column):
  - using `reshape`
  - using `newaxis`

```
In [11]: ## Converting to row 2D array:
         # using reshape
```

```
#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------

print(temp)
print(temp.shape)
#
print()
# using newaxis

#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------

print(temp)
print(temp.shape)
```

```
[[0 1 2 3 4 5]]
(1, 6)

[[0 1 2 3 4 5]]
(1, 6)
```

In [12]:
```
## Converting to column 2D array:
# using reshape

#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------

print(temp)
print(temp.shape)
#
print()
# using newaxis

#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------

print(temp)
print(temp.shape)
```

```
[[0]
 [1]
 [2]
 [3]
 [4]
 [5]]
(6, 1)

[[0]
 [1]
 [2]
 [3]
 [4]
 [5]]
(6, 1)
```

Concatenation

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines:

- `np.concatenate` : Concatenate on any of the 3-axis
- `np.vstack` : Vertical Concatenation
- `np.hstack` : Horizontal Concatenation
- `np.dstack` : Depth Concatenation

`np.concatenate`

In [27]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])

#----------------------------------------------------- CODE HERE ---
---------------------------------------------------
```
Out[27]:
```
array([1, 2, 3, 3, 2, 1])
```

Also it's possible to concatenate more that two arrays at once:

In [26]:
```
z = [99, 99, 99]

#----------------------------------------------------- CODE HERE ---
---------------------------------------------------
```
```
[ 1  2  3  3  2  1 99 99 99]
```

It can also be used for two-dimensional arrays:

In [28]:
```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
```

In [29]:
```
# Concatenating along the axis-1 = hstack

#----------------------------------------------------- CODE HERE ---
---------------------------------------------------
```
Out[29]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

In [30]:
```
# Concatenating along the axis-0 = vstack

#----------------------------------------------------- CODE HERE ---
---------------------------------------------------
```
Out[30]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

Exercise

Give the two 1D arrays below. Concatenate them vertically.

In [ ]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
### YOUR CODE HERE
#
#
###
```

Another more readable and clearer method for concatenating is by using `np.hstack` and `np.vstack` :

In [33]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])

#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------
```

Out[33]:
```
array([[1, 2, 3],
       [3, 2, 1]])
```

> NOTICE: There is no need to reshape to two dimensional array compared to `concatenate`

In [34]:
```
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])
y = np.array([[99],
              [99]])

#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------
```

Out[34]:
```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

In [35]:
```
#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------
```

Out[35]:
```
array([[99,  9,  8,  7],
       [99,  6,  5,  4]])
```

Splitting

Splitting is the opposite of **Concatenating**, and can be implemented using the methods:

- `numpy.split` : For splitting on any of the 3 axis

- `numpy.hsplit` : Horizontal splitting
- `numpy.vsplit` : Vertical splitting
- `numpy.dsplit` : Depth splitting

For example:

In [37]:
```
np.split?
```

In [38]:
```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5]) # will result: x1=x[:3], x2=[3:5], x3=
[5:]
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

also,

In [44]:
```
x = np.array(x).reshape([-1,2])
x
```

Out[44]:
```
array([[ 1,  2],
       [ 3, 99],
       [99,  3],
       [ 2,  1]])
```

In [49]:
```
# Vertical split (on axis 0) the first, last and the in between
x1,x2,x3 = np.split(x,[1,-1],axis=0)
print(x1)
print("-----")
print(x2)
print("-----")
print(x3)
```

```
[[1 2]]
-----
[[ 3 99]
 [99  3]]
-----
[[2 1]]
```

`hsplit`, `vsplit` and `dsplit` provide more readable tool that `split`:

In [36]:
```
x = np.arange(16).reshape((4, 4))
x
```

Out[36]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [51]:
```
# Split the first, last and rest rows vertically

#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------

print(x1)
print("-----")
print(x2)
print("-----")
print(x3)
```

```
[[1 2]]
-----
[[ 3 99]
 [99  3]]
-----
[[2 1]]
```

Computation of NumPy Arrays[[1](#ref1)]

Repeated computations on NumPy arrays can be slow by using the conventional loops (like `for`). instead NumPy provides what is called *universal functions* (ufunc), which are optimized for repeated calculations on NumPy array elements.

Python's default implementation (known as CPython) does some operations slowly. This is due to the *dynamic*, interpreted nature of the language.

> The fact that types are flexible, so that sequences of operations cannot be compiled down to efficient machine code as in languages like C and Fortran.

Recently various attempts tried to address and solve this weakness, such as:

- PyPy project: a just-in-time compiled implementation of Python;
- Cython project: which converts Python code to compilable C code
- Numba project: which converts snippets of Python code to fast LLVM bytecode.

Each of these has its strengths and weaknesses, but it is *safe* to say that none of the three approaches has yet surpassed the **reach** and **popularity** of the standard CPython engine.

The slowness of CPython manifests itself in situations on which iterative (repetitive) operations are required, as example looping over the array elements to perform an operation.

As example, if we want to find the reciprocal of array elements in the conventional way:

In [8]:
```
import numpy as np
np.random.seed(56)
```

Exercise

1. Define a function that accepts a list and computer the reciprocal of every element in the list by **looping** over the list element.

In [ ]:
```
# Define a function that takes a list and returns the reciprocal of that
```

```
list
def get_reciprocal(values):
    result = np.empty(shape=np.array(values).shape)
    ### YOUR CODE HERE
    #
    #
    ###
    return result
    pass
```

1. Check the integrity of the defined function

In [45]:
```
a = [2]*10 # Create a list containing 10 elements of 2
print("a = ",a)
print("1/a = ",get_reciprocal(a))
```

```
a =  [2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
1/a =  [0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5]
```

1. Create an array of random (`size=1000000`) integers. Compute their reciprocal using `get_reciprocal()` and **time the execution time**:

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

Too slow !!
The bottleneck here is not the operations themselves, but the type-checking and function dispatches that CPython must do at each cycle of the loop. Each time the reciprocal is computed, Python first examines the object's type and does a dynamic lookup of the correct function to use for that type.

Keep this experiment and its result in your head. We will compare it with the speed on numpy ufuncs.

Vectorized operations

NumPy provides interface for *statically typed* arrays, called **vectorized operation**. This can be accomplished by simply performing an operation on the array, which will then be applied to each element.

This vectorized approach will lead to much faster execution.

In [14]:
```
# Check that the vectorized reciprocal operation works well
a = np.array(a)
```

```
print("a = ",a)
print("1/a = ",1.0/a)
```

```
a =  [2 2 2 2 2 2 2 2 2 2]
1/a =  [0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5]
```

In [16]:
```
# Compare the performance with ``get_reciprocal()``

#----------------------------------------------------------- CODE HERE ---
-----------------------------------------------------
```

```
3.71 ms ± 75.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

> **Much Faster !!**

Numpy's UFuncs can be categorized into:

- *unary ufuncs* : Which operate on a singe input
- *birany ufuncs*: Which oeprate on two inputs

Unary Universal Functions

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
`ceil`	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as a separate array
`isnan`	Return boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions

`logical_not`	Compute truth value of `not x` element-wise (equivalent to `~arr`).

source: https://bit.ly/2UeB3Tb

Binary Universal Functions

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum; `fmax` ignores `NaN`
`minimum, fmin`	Element-wise minimum; `fmin` ignores `NaN`
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array (equivalent to infix operators `>, >=, <, <=, ==, !=`)
`logical_and, logical_or, logical_xor`	Compute element-wise truth value of logical operation (equivalent to infix operators `& |, ^`)

source: https://bit.ly/2UeB3Tb

See the following examples:

Array arithmetics

Includes basic element-operations on arrays, like: *addition, subtraction, multiplication and devision*:

In [29]:
```
x = np.arange(1,5).reshape([2,-1])
# x = np.repeat(x,2,axis=0)

#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------

x
```

Out[29]:
```
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

In [33]:
```
print("x + 5 =\n", #------------------------- CODE HERE ---------------
------------------)
print("\nx - 5 =\n", #------------------------- CODE HERE ------------
--------------------)
print("\nx * 2 =\n", #------------------------- CODE HERE ------------
--------------------)
```

```
print("\nx / 2 =\n", #------------------------ CODE HERE ------------
--------------------)
print("\nx // 2 =\n", #------------------------ CODE HERE -----------
---------------------)  # floor division


#------------------------------------------------------- CODE HERE ---
-----------------------------------------------------
```

```
x + 5 =
 [[6 7]
 [8 9]
 [6 7]
 [8 9]]

x - 5 =
 [[-4 -3]
 [-2 -1]
 [-4 -3]
 [-2 -1]]

x * 2 =
 [[2 4]
 [6 8]
 [2 4]
 [6 8]]

x / 2 =
 [[0.5 1. ]
 [1.5 2. ]
 [0.5 1. ]
 [1.5 2. ]]

x // 2 =
 [[0 1]
 [1 2]
 [0 1]
 [1 2]]
```

There is also a unary ufunc for negation, and a `**` operator for exponentiation, and a `%` operator for modulus:

In [35]:
```
print("\n-x     = \n", #------------------------ CODE HERE ----------
----------------------)
print("\nx ** 2 = \n", #------------------------ CODE HERE ----------
----------------------)
print("\nx % 2  = \n", #------------------------ CODE HERE ----------
----------------------)
```

```
-x     =
 [[-1 -2]
 [-3 -4]
 [-1 -2]
 [-3 -4]]

x ** 2 =
 [[ 1  4]
 [ 9 16]
 [ 1  4]
 [ 9 16]]
```

```
x % 2  =
 [[1 0]
 [1 0]
 [1 0]
 [1 0]]
```

or making a combination of vectorized operations:

In [36]:
```
-(0.5*x + 1) ** 2
```

Out[36]:
```
array([[-2.25, -4.  ],
       [-6.25, -9.  ],
       [-2.25, -4.  ],
       [-6.25, -9.  ]])
```

Each of the vectorized operations are **overloaded operators** (in terms of `C++` language) of the following NumPy arithmetic operations:

The following table lists the arithmetic operators implemented in NumPy:

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

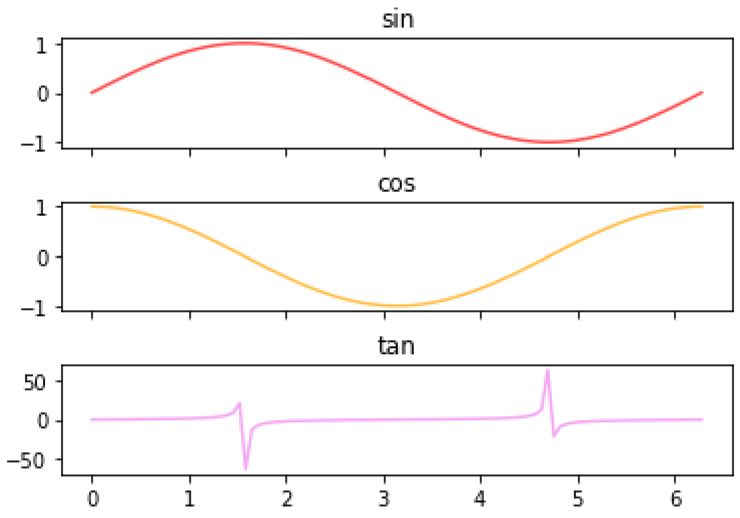
Trigonometric functions

Example of the NumPy's trigonometric functions.

In [47]:
```
import matplotlib.pyplot as plt

%matplotlib inline
```

In [75]:
```
theta = np.linspace(0,2*np.pi,100)
y_sin = np.sin(theta)
y_cos = np.cos(theta)
y_tan = np.tan(theta)

## Plotting

#-------------------------------------------------------- CODE HERE ---
-----------------------------------------------------
```

```
# add more space between subplots
fig.subplots_adjust(hspace=0.5)


#------------------------------------------------------------ CODE HERE ---
---------------------------------------------------------


ax1.set_title("sin");

#------------------------------------------------------------ CODE HERE ---
---------------------------------------------------------


ax2.set_title("cos");

#------------------------------------------------------------ CODE HERE ---
---------------------------------------------------------


ax3.set_title("tan");
```



```
Advanced Ufunc features
```

1. Specifying output

For all ufuncs, you can specify the output using the input argument `out` . As example:

In [76]:
```
x = np.arange(5)
y = np.empty(5)

#------------------------------------------------------------ CODE HERE ---
---------------------------------------------------------


print(y)
```

```
[ 0. 10. 20. 30. 40.]
```

This can even be used with array views. For example, we can write the results of a computation to every other element of a specified array:

In [77]:
```
y = np.zeros(10)
```

```
#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------

print(y)
```

```
[ 1.  0.  2.  0.  4.  0.  8.  0. 16.  0.]
```

2. Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed directly from the object.

For example, if we'd like to *reduce* an array with a particular operation, we can use the `reduce` method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

In [79]:
```
x = np.arange(1, 6)

#---------------------------------------------------------- CODE HERE ---
------------------------------------------------------
```

Out[79]:
```
15
```

Similarly, calling `reduce` on the `multiply` ufunc results in the product of all array elements:

In [80]:
```
np.multiply.reduce(x)
```

Out[80]:
```
120
```

If we'd like to store all the intermediate results of the computation, we can instead use `accumulate`:

In [81]:
```
np.add.accumulate(x)
```

Out[81]:
```
array([ 1,  3,  6, 10, 15], dtype=int32)
```

In [82]:
```
np.multiply.accumulate(x)
```

Out[82]:
```
array([  1,   2,   6,  24, 120], dtype=int32)
```

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`), which we'll explore in Aggregations.

Outer products

Any ufunc can compute the output of all pairs of two different inputs using the `outer` method. This allows you, in one line, to do things like create a multiplication table:

In [83]:
```
x = np.arange(1, 6)

#---------------------------------------------------------- CODE HERE ---
```

```
----------------------------------------------------
```

Out[83]:
```
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

`learning more about UFuncs:`

More information on universal functions (including the full list of available functions) can be found on the NumPy and SciPy documentation websites.

Aggregations [[1](ref1)]

One of the first steps when you work with a given dataset, is to explore it.
The most common summary statistics are `mean`, `standard deviation`, `min`, `max`, etc. .
NumPy has fast built-in aggregation functions for working on arrays. Here are some of them:

In [87]:
```
x = np.random.random(100)
```

`Summation`

Summing the values of an array:

In [88]:
```
# Summing all elements

#------------------------------------------------------- CODE HERE ---
----------------------------------------------------
```

Out[88]:
```
53.843769226570565
```

Even that Python has a built in `sum` function, the `numpy.sum` is more optimized:

In [90]:
```
my_big_arr = np.random.random(1000000)

#------------------------------------------------------- CODE HERE ---
----------------------------------------------------


#------------------------------------------------------- CODE HERE ---
----------------------------------------------------
```

```
131 ms ± 4.47 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
674 µs ± 4.75 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Summing for multi-dimensional array

In [91]:
```
x2 = np.random.randint(1,10,size=[5,6])
x2
```

Out[91]:
```
array([[3, 1, 4, 8, 2, 6],
       [5, 6, 5, 5, 7, 7],
       [8, 7, 5, 3, 5, 6],
       [7, 2, 8, 6, 7, 7],
       [2, 2, 9, 2, 4, 7]])
```

In [95]:
```
# sum over the rows (axis=0)

#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------


#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------

print(result)
```

```
[25 18 31 24 25 33]
```

In [96]:
```
# sum over the rows (axis=1)

#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------


#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------

print(result)
```

```
[24 35 34 37 26]
```

Min, Max

NumPy is still quicker than native python `min`, `max` functions !

In [97]:
```
#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------


#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------
```

```
85.9 ms ± 3.29 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
361 µs ± 19 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [98]:
```
#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------


#----------------------------------------------------------- CODE HERE ---
-------------------------------------------------------
```

```
82.9 ms ± 1.52 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
337 µs ± 4.29 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

min/max for multi-dimensional array

In [99]:
```
x2
```

Out[99]:
```
array([[3, 1, 4, 8, 2, 6],
       [5, 6, 5, 7, 7],
       [8, 7, 5, 3, 5, 6],
       [7, 2, 8, 6, 7, 7],
       [2, 2, 9, 2, 4, 7]])
```

Exercise

For the previous $x_2$:

1. Get the minimum of every **column**, save the result in the variable `result`

In [ ]:
```
result = []
### YOUR CODE HERE
#
###
assert x2.shape[1]==len(result), "Something is wrong !"
print(result)
```

1. Get the maximum of every **row**, save the result in the variable `result`

In [ ]:
```
result = []
### YOUR CODE HERE
#
###
assert x2.shape[1]==len(result), "Something is wrong !"
print(result)
```

`Other aggregation functions`

The following table provides a list of useful aggregation functions available in NumPy:

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements

`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

Exercise

As you will learn later, Pandas and numpy work well together, Pandas is used usually for manipulating tabulated data.

Here we will work on the `president_heights.csv` dataset. So let's first explore the first 3 lines of the dataset (using bash command):

In [103]:
```
!head -3 ../assets/president_heights.csv
```

```
order,name,height(cm)
1,George Washington,189
2,John Adams,170
```

So you can see that the dataset, contains three columns (order, name, height(cm)), and the delimiter is `,`.

Now let's import the csv file in Pandas, then convert the columns in focus into numpy:

In [115]:
```
import pandas as pd             # For tabulated data
import numpy as np              # For array manipulation
import matplotlib.pyplot as plt # For plotting
import seaborn as sns           # Advanced Functions for plotting

%matplotlib inline
```

In [106]:
```
data = pd.read_csv('../assets/president_heights.csv')
# Display the first 5 rows of data
data.head()
```

Out[106]:

	order	name	height(cm)
0	1	George Washington	189

1	2	John Adams	170
2	3	Thomas Jefferson	189
3	4	James Madison	163
4	5	James Monroe	183

Create the heights and names arrays:

```
In [107]: heights = np.array(data['height(cm)'])
          names = np.array(data.name)
```

1. Compute:
   - Mean of the heights
   - Standard deviation of the heights
   - Minimum height
   - Maximum height
   - Heights range

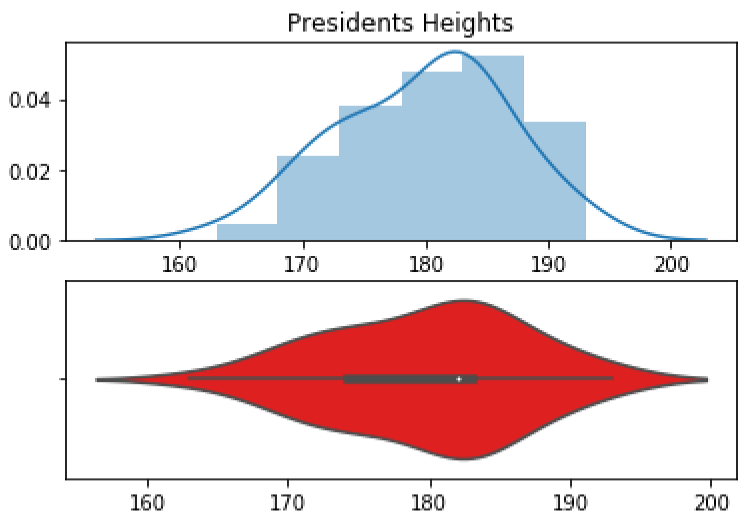
```
In [ ]: ### YOUR CODE HERE
        #
        #
        ###
```

1. Compute:
   - 25^{th} Percentile
   - Median
   - 75^{th} Percentile

```
In [ ]: ### YOUR CODE HERE
        #
        #
        ###
```

Sometimes it's more useful to visualize the data:

```
In [127]: fig, (ax1,ax2)=plt.subplots(2,1);
          sns.distplot(heights, ax=ax1);
          ax1.set_title("Presidents Heights")
          #
          sns.violinplot(heights, gamma=0.4, split=True, color="red", ax=ax2)

Out[127]: <matplotlib.axes._subplots.AxesSubplot at 0x2a6f9608748>
```



1. Get the name of all presidents whom are shorter than the $25^{th}$ percentile.

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

1. Get the name of the tallest and shortest presidents

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

Broadcasting [[1](ref1)]

Another means to avoid the slow Python iterative loops beside the previously introduced **vectorized operations**, is using NumPy **broadcasting**.

Broadcasting is simply a set of rules for applying binary ufuncs (e.g., addition, subtraction, multiplication, etc.) on arrays of different sizes.

Recall that for arrays of the **same size**, binary operations are performed on an element-by-element basis:

In [3]:
```
import numpy as np
```

In [4]:
```
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
```

```
#------------------------------------------------------- CODE HERE ---
-----------------------------------------------------
```

Out[4]:
```
array([5, 6, 7])
```

Broadcasting allows ufunc-binary operations to be performed on arrays of different sizes, for example:

In [5]:
```
#------------------------------------------------------- CODE HERE ---
-----------------------------------------------------
```

Out[5]:
```
array([5, 6, 7])
```

> Remember that this is not allowed in conventional python

In [7]:
```
import sys # To print the error/exception name
```

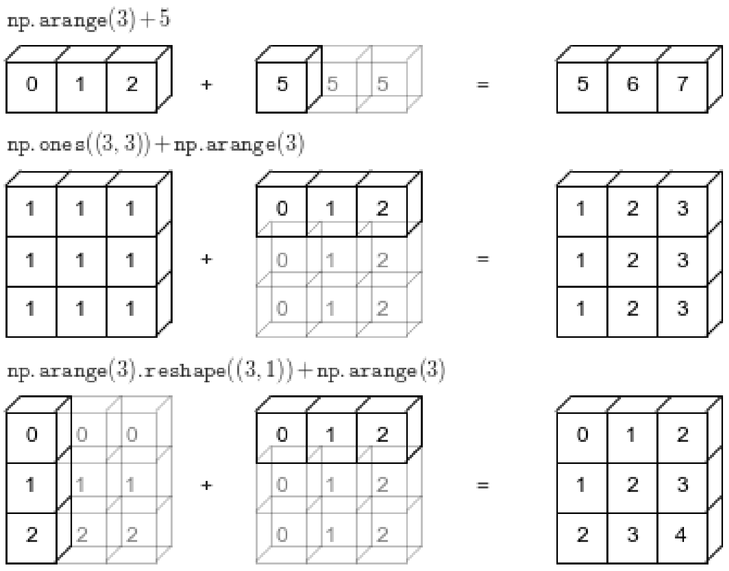
In [20]:
```
a = [1,2,3]
try:
    a+5
    pass
except:
    e_type,e_value,_=sys.exc_info()
    print("ERROR type: {}\n{}".format(e_type.__name__, e_value))
    pass
```

```
ERROR type: TypeError
can only concatenate list (not "int") to list
```

Broadcasting rules

To illustrate how the broadcasting work, see the following figure:

source: [1]


np.arange(3)+5
np.ones((3,3))+np.arange(3)
np.arange(3).reshape((3,1))+np.arange(3)

- **Rule 1**: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (the one with more dimensions) side.
- **Rule 2**: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- **Rule 3**: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

To make these rules clear, let's consider a few examples in detail:

```
Example 1:
```

```
x1=np.ones((2,3))
x2=np.arange(3)
```

and required is:

```
x1+x2
```

Let's consider an operation on these two arrays. The shape of the arrays are

- `x1.shape = (2, 3)`
- `x2.shape = (3,)`

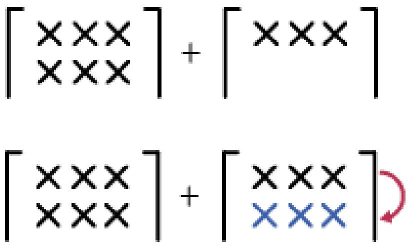
We see by rule 1 that the array `x2` has fewer dimensions, so we pad it on the left with ones:

- `x1.shape -> (2, 3)`
- `x2.shape -> (1, 3)`

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- `x1.shape -> (2, 3)`
- `x2.shape -> (2, 3)`

The shapes match, and we see that the final shape will be `(2, 3)`:



In [22]:
```
x1=np.ones((2,3))
x2=np.arange(3)

print("x1.shape={}".format(x1.shape))
print("x2.shape={}".format(x2.shape))
```

```
x1.shape=(2, 3)
x2.shape=(3,)
```

Since x_2 has fewer dimensions that x_1, x_2 will be padded to the left one:

In [28]:
```
print("{}\n+\n{}\n =\n {}".format(x1,x2,x1+x2))
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
+
[0 1 2]
 =
 [[1. 2. 3.]
 [1. 2. 3.]]
```

In [29]:
```
print("{}\n+\n{}\n =\n {}".format(x2,x1,x2+x1))
```

```
[0 1 2]
+
[[1. 1. 1.]
 [1. 1. 1.]]
 =
 [[1. 2. 3.]
 [1. 2. 3.]]
```

```
    Example 2:
```

```
x1=np.ones((2,3))
x2=np.arange(2)
```

and required is:

```
x1+x2
```

In [35]:
```
x1=np.ones((2,3))
x2=np.arange(2)
```

In [36]:
```
try:
    print("{}\n+\n{}\n =\n {}".format(x1,x2,x1+x2))
    pass
except:
    e_type,e_value,_=sys.exc_info()
    print("ERROR type: {}\n{}".format(e_type.__name__, e_value))
    pass
```

```
ERROR type: ValueError
operands could not be broadcast together with shapes (2,3) (2,)
```

> The default attempt of summation is applying each of elements of the lower dimension to the columns of the higher dimension array. Since in the previous example there is no match, an error will be generated.

The solution is to match the number of dimensions (reshaping and transposing x_1 $\rightarrow$ we have to make **Rule1** manually). Then NumPy will pad both arrays on the matching dimension (which is the rows of x_1):

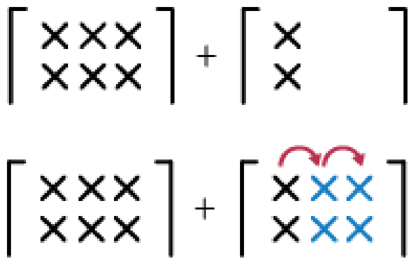
Again, we'll start by writing out the shape of the arrays:

- `x1.shape = (2, 3)`
- `x2.shape = (2,1)`

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

- `a.shape -> (2, 3)`
- `b.shape -> (2, 3)`

Because the result matches, these shapes are compatible. We can see this here:



In [39]:
```
x2=x2.reshape(2,-1)
print("{}\n+\n{}\n =\n {}".format(x1,x2,x1+x2))
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
+
[[0]
 [1]]
 =
 [[1. 1. 1.]
 [2. 2. 2.]]
```

```
    Example 3: Both arrays need to be broadcased
```

```
x1=np.arange(3).reshape((3, 1))
```

```
x2=np.arange(3)
```

and required is:

```
x1+x2
```

Again, we'll start by writing out the shape of the arrays:

- `x1.shape = (3, 1)`
- `x2.shape = (3,)`

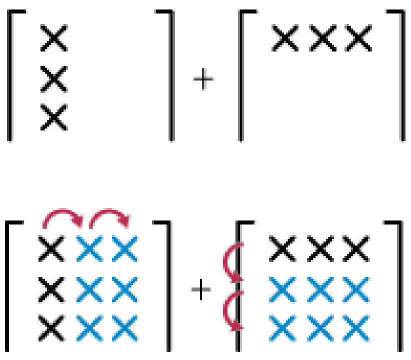
Rule 1 says we must pad the shape of `x2` with ones:

- `x1.shape -> (3, 1)`
- `x2.shape -> (1, 3)`

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

- `x1.shape -> (3, 3)`
- `x2.shape -> (3, 3)`

Because the result matches, these shapes are compatible. We can see this here:



In [40]:
```
x1=np.arange(3).reshape((3, 1))
x2=np.arange(3)
```

In [41]:
```
print("{}\n+\n{}\n =\n {}".format(x1,x2,x1+x2))
```

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

Example 4:

```
x1=np.ones((3, 2))
x2=np.arange(3)
```

and required is:

```
x1+x2
```

This is just a slightly different situation than in the first example: the matrix `M` is transposed. How does this affect the calculation? The shape of the arrays are

- `M.shape = (3, 2)`
- `a.shape = (3,)`

Again, rule 1 tells us that we must pad the shape of `a` with ones:

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

By rule 2, the first dimension of `a` is stretched to match that of `M` :

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

Now we hit rule 3–the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

In [43]:
```
x1=np.ones((3, 2))
x2=np.arange(3)
```

In [44]:
```
try:
    print("{}\n+\n{}\n =\n {}".format(x1,x2,x1+x2))
    pass
except:
    e_type,e_value,_=sys.exc_info()
    print("ERROR type: {}\n{}".format(e_type.__name__, e_value))
    pass
```

```
ERROR type: ValueError
operands could not be broadcast together with shapes (3,2) (3,)
```

> Note the potential confusion here: you could imagine making `x2` and `x1` compatible by, say, padding `x2` 's shape with ones on the right rather than the left. But this is not how the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side padding is what you'd like, you can do this explicitly by reshaping the array (we'll use the `np.newaxis` keyword or using `np.reshape`)

In [45]:
```
x1 + x2[:, np.newaxis]
```

Out[45]:
```
array([[1., 1.],
       [2., 2.],
       [3., 3.]])
```

Also note that while we've been focusing on the `+` operator here, these broadcasting rules apply to *any* binary `ufunc` . For example, here is the `logaddexp(a, b)` function, which computes `log(exp(a) + exp(b))` with more precision than the naive approach:

In [47]:
```
try:
    print("log(exp(x1) + logexp(x2)) =\n {}".format(np.logaddexp(x1,x2)
))
    pass
except:
    e_type,e_value,_=sys.exc_info()
    print("ERROR type: {}\n{}".format(e_type.__name__, e_value))
    pass
```

```
ERROR type: ValueError
operands could not be broadcast together with shapes (3,2) (3,)
```

In [48]:
```
np.logaddexp(x1, x2[:, np.newaxis])
```

Out[48]:
```
array([[1.31326169, 1.31326169],
       [1.69314718, 1.69314718],
       [2.31326169, 2.31326169]])
```

Exercise

In this exercise, it is required to convert a color image into a gray scale.

You can find the mathematics behind this image manipulation HERE.

To do this conversion you have to:

1. Scale the values to your image to a range $\in [0,1]$
2. Convert the scaled [R,G,B] channels into [$R_l$, $G_l$, b_l], using this function: \begin{equation*} C_{l} =\left\{ \begin{array}{@{}ll@{}} \frac{C}{12.92}, & \text{if}\ C \leq 0.04045 \\ \left(\frac{C+0.055}{1.055} \right)^{2.4}, & \text{otherwise} \end{array}\right. \end{equation*}

    > Where $C_l$ is the linearly converted channel (i.e. $R_l$, $G_l$, $B_l$) and $C$ is the original channel (i.e. $R$, $G$, $B$)

3. Compute \begin{equation*} Y_{l} = 0.2126 R_l + 0.7152 G_l + 0.0722 B_l \end{equation*}
4. The final gray scale image is done by stacking on depth-axis three $Y_{l}$ (the final image dimensions should be: $(lengh,\ width,\ 3)$)

In [303]:
```
from skimage import io # you must install the package: scikit-image
import matplotlib.pyplot as plt
```

In [304]:
```
img_dir = "./images/vegitables.jpg"
# Convert
img_mat = io.imread(img_dir)
img_mat.shape
```

Out[304]:
```
(720, 1280, 3)
```

In [ ]:
```
### YOUR CODE HERE
#
#
###
```

Boolean arrays [[1](#ref1)]

In this part some of the boolean operations will be displayed.

As example:

In [3]:
```
import numpy as np
```

In [4]:
```
x = np.random.randint(1,10,[3,4])
print(x)
```

```
[[6 1 3 9]
 [9 6 5 6]
 [6 3 2 6]]
```

```
    Boolean operations
```

Python's *bitwise logic operators*, `&`, `|`, `^`, and `~`. Like with the standard arithmetic operators, NumPy overloads these as ufuncs which work element-wise on (usually Boolean) arrays.

As example, let us indicate the elements of x which are bigger that $4$ but less than $8$:

In [12]:
```
#-------------------------------------------------------- CODE HERE ---
-----------------------------------------------------
```

Out[12]:
```
array([[ True, False, False, False],
       [False,  True,  True,  True],
       [ True, False, False,  True]])
```

> Note : that one common point of confusion is the difference between the keywords `and` and `or` on one hand, and the operators `&` and `|` on the other hand.
>
> The difference is this: `and` and `or` check the truth or falsehood of *entire object*, while `&` and `|` refer to *bits within each object*. When you use `and` or `or`, it's equivalent to asking Python to treat the object as a single Boolean

> entity.

```
In [13]: bool(42), bool(0)
Out[13]: (True, False)
```

```
In [14]: bool(42 and 0)
Out[14]: False
```

```
In [15]: bool(42 or 0)
Out[15]: True
```

When you use `&` and `|` on integers, the expression operates on the bits of the element, applying the *and* or the *or* to the individual bits making up the number:

```
In [16]: # binary representation of the int:42
         bin(42)
Out[16]: '0b101010'
```

```
In [17]: bin(59)
Out[17]: '0b111011'
```

```
In [18]: bin(42 & 59)
Out[18]: '0b101010'
```

```
In [19]: bin(42 | 59)
Out[19]: '0b111011'
```

When doing a Boolean expression on a given array, you should use `|` or `&` rather than `or` or `and`:

```
In [22]: x = np.arange(10)

         #------------------------------------------------------------ CODE HERE ---
         ------------------------------------------------------
Out[22]: array([False, False, False, False, False,  True,  True,  True, False,
                False])
```

Trying to evaluate the truth or falsehood of the entire array will give the same `ValueError` we saw previously:

```
In [26]: import sys
         try:
             (x > 4) and (x < 8)
             pass
```

```
except ValueError as e:
    print ("ValueError: \n",e)
```

```
ValueError: 
 The truth value of an array with more than one element is ambiguous. Us
e a.any() or a.all()
```

Counting entries

- Count the number of entries that are less than $6$:

In [5]:
```
#------------------------------------------------------------ CODE HERE ---
---------------------------------------------------------
```

Out[5]:
```
5
```

Another way to get at this information is to use `np.sum`; in this case, `False` is interpreted as `0`, and `True` is interpreted as `1`:

In [6]:
```
#------------------------------------------------------------ CODE HERE ---
---------------------------------------------------------
```

Out[6]:
```
5
```

The advantage of using the `sum()` method, is the ability to use its aggregation properties. As example, we can use it to compute how many values are less than $6$ in each row:

In [7]:
```
np.sum(x<6, axis=1)
```

Out[7]:
```
array([2, 1, 2])
```

Also we can use it to compute number of elements less than $6$ in every column:

In [8]:
```
np.sum(x<6, axis=0)
```

Out[8]:
```
array([0, 2, 3, 0])
```

`numpy.any()` and **`numpy.all()`**

- Check if any on the x elements is bigger than 8:

In [9]:
```
np.any(x>8)
```

Out[9]:
```
True
```

- Check if all the elements are less than $10$:

In [10]:
```
np.all(x<10)
```

Out[10]:
```
True
```

`np.all` and `np.any` can be used along particular axes as well. For example: Are all values in each row less than 8?

```
In [11]: np.all(x < 8, axis=1)
```

```
Out[11]: array([False, False,  True])
```

> Notice that `any` , `all` and `sum` methods as well as other methods have common method-names between numpy and conventional python, but the computation speed and the syntax are usually different in both groups.

[[1](#ref1)] Fancy Indexing

In the previous sections, we saw how to access and modify portions of arrays using simple indices (e.g., `arr[0]`), slices (e.g., `arr[:5]`), and Boolean masks (e.g., `arr[arr > 0]`). Here another style of array indexing will be introduced, known as *fancy indexing*. Fancy indexing is like the simple indexing we've already seen, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

Fancy means: passing an array of indices to access multiple array elements at once.

For example:

```
In [29]: import numpy as np
         rand=np.random.RandomState(42)
         x = rand.randint(100, size=10)
         print(x)
```

```
[51 92 14 71 60 20 82 86 74 74]
```

Suppose we want to access three different elements. We could do it like this:

```
In [30]: [x[3], x[7], x[2]]
```

```
Out[30]: [71, 86, 14]
```

Alternatively, we can pass a single list or array of indices to obtain the same result:

```
In [31]: ind = [3, 7, 4]
         x[ind]
```

```
Out[31]: array([71, 86, 60])
```

When using fancy indexing, the shape of the result reflects the shape of the *index arrays* rather than the shape of the *array being indexed*:

In [32]:
```
ind = np.array([[3, 7],
                [4, 5]])
x[ind]
```

Out[32]:
```
array([[71, 86],
       [60, 20]])
```

Fancy indexing also works in multiple dimensions. Consider the following array:

In [33]:
```
X = np.arange(12).reshape((3, 4))
X
```

Out[33]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Like with standard indexing, the first index refers to the row, and the second to the column:

In [34]:
```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
```

Out[34]:
```
array([ 2,  5, 11])
```

The pairing of indices in fancy indexing follows all the broadcasting rules:

In [35]:
```
X[row[:, np.newaxis], col]
```

Out[35]:
```
array([[ 2,  1,  3],
       [ 6,  5,  7],
       [10,  9, 11]])
```

Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen:

In [36]:
```
print(X)
```

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

We can combine fancy and simple indices:

In [37]:
```
X[2, [2, 0, 1]]
```

Out[37]:
```
array([10,  8,  9])
```

We can also combine fancy indexing with slicing:

In [38]:
```
X[1:, [2, 0, 1]]
```

Out[38]:
```
array([[ 6,  4,  5],
       [10,  8,  9]])
```

And we can combine fancy indexing with masking:

In [39]:
```
mask = np.array([1, 0, 1, 0], dtype=bool)

#--------------------------------------------------------- CODE HERE ---
------------------------------------------------------
```

Out[39]:
```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

Exercise

One common use of fancy indexing is the selection of subsets of rows from a matrix. For example, we might have an $N$ by $D$ matrix representing $N$ points in $D$ dimensions, such as the following points drawn from a two-dimensional normal distribution:

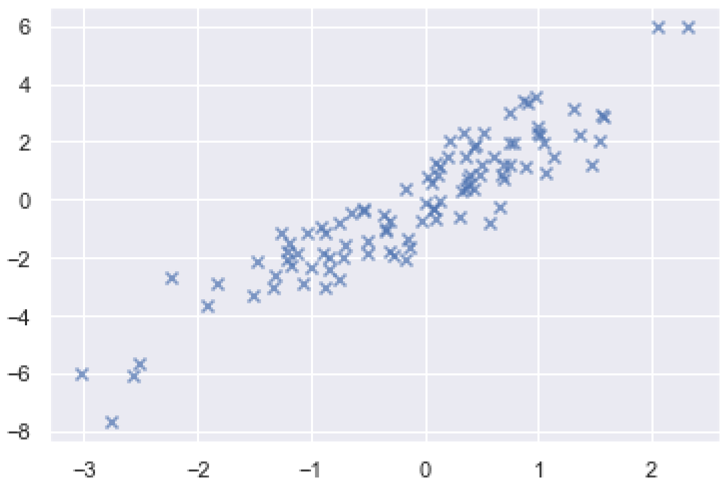
In [40]:
```
mean = [0, 0]
cov = [[1, 2],
       [2, 5]]
X = rand.multivariate_normal(mean, cov, 100)
X.shape
```

Out[40]:
```
(100, 2)
```

These points features can be visualized as below:

In [46]:
```
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

In [68]:
```
plt.scatter(X[:, 0], X[:, 1], marker = "x", alpha=0.7);
```

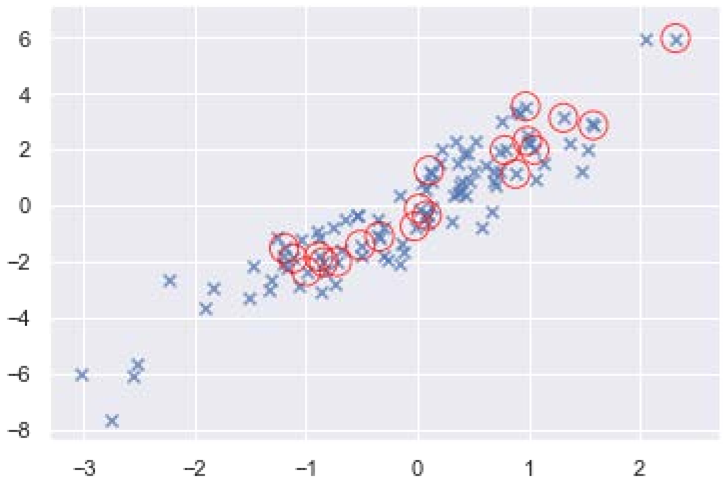


1. Choose 20 random indices (rows from both features) with no repeats, and use these indices to create a new subset from the original dataset (save it in a variable called `selection`):

```
In [ ]: ### YOUR CODE HERE
        #
        #
        ###
```

Now to see which points were selected, let's over-plot large circles at the locations of the selected points:

```
In [69]: plt.scatter(X[:, 0], X[:, 1], marker="x", alpha=0.7)
         plt.scatter(selection[:, 0], selection[:, 1],
                     edgecolors="red",facecolor='none',
                     alpha=0.7, s=200);
```



This sort of strategy is often used to quickly partition datasets, as is often needed in **train/validation/test** splitting for *validation of statistical models*, and in *sampling* approaches to answer statistical questions.

Linear Algebra [[3](#ref3)]

Dot product

Linear algebra, like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library. Unlike some languages like MATLAB, multiplying two two-dimensional arrays with `*` is an element-wise product instead of a matrix dot product. Thus, there is a function `dot`, both an array method and a function in the `numpy` namespace, for matrix multiplication:

In [3]:
```
import numpy as np
```

In [5]:
```
x = np.array([[1., 2., 3.], [4., 5., 6.]])
y = np.array([[6., 23.], [-1, 7], [8, 9]])
print("x = \n", x)
print("\ny = \n",y)
```

```
x =
 [[1. 2. 3.]
 [4. 5. 6.]]

y =
 [[ 6. 23.]
 [-1.  7.]
 [ 8.  9.]]
```

In [7]:
```
#---------------------------------------------------- CODE HERE ---
-----------------------------------------------

print("result = \n",result)
print("\n",result.shape)
```

```
result =
 [[ 28.  64.]
 [ 67. 181.]]

 (2, 2)
```

- A matrix product between a two-dimensional array and a suitably sized one-dimensional array results in a one-dimensional array:

In [9]:
```
#---------------------------------------------------- CODE HERE ---
-----------------------------------------------
```

Out[9]:
```
array([ 6., 15.])
```

- The `@` symbol (as of Python 3.5) also works as an operator that performs matrix multiplication:

In [10]:
```
#---------------------------------------------------- CODE HERE ---
```

```
--------------------------------------------------------
```

Out[10]:
```
array([ 6., 15.])
```

In [11]:
```
#------------------------------------------------------------ CODE HERE ---
--------------------------------------------------------
```

Out[11]:
```
array([[ 28.,  64.],
       [ 67., 181.]])
```

Matrix Inverse and decomposition

`numpy.linalg` has a standard set of matrix decompositions and things like inverse and determinant. These are implemented under the hood via the same industry-standard linear algebra libraries used in other languages like _MATLAB _and *R*:

In [13]:
```
from numpy.linalg import inv, qr
```

In [14]:
```
x = np.random.randn(5, 5)
x
```

Out[14]:
```
array([[-0.80062008,  0.47244462, -0.71563444,  0.44292808, -0.29847464]
,
       [-0.91803117, -2.05234005,  0.44555587, -0.64427851, -0.05289097]
,
       [ 0.24932707, -0.29746057,  0.47392311, -0.35229938,  0.14833523],
       [-0.2754943 , -0.66228763,  0.53128445,  0.30689697,  1.57484043],
       [-1.78567704, -0.23798097, -1.58696847,  0.9391031 ,  0.79243442]
])
```

In [24]:
```
#------------------------------------------------------------ CODE HERE ---
--------------------------------------------------------


print("x^2 = \n", y)
```

```
x^2 =
 [[ 4.81047734  2.03911214  2.96952631 -1.61247035 -1.52438717]
 [ 2.03911214  5.01904623 -1.36769806  1.20959001 -1.30816801]
 [ 2.96952631 -1.36769806  3.73598788 -2.0982769  -0.16054786]
 [-1.61247035  1.20959001 -2.0982769   1.71149532  1.07710668]
 [-1.52438717 -1.30816801 -0.16054786  1.07710668  3.22196259]]
```

- Matrix inverse:

In [25]:
```
#------------------------------------------------------------ CODE HERE ---
--------------------------------------------------------
```

Out[25]:
```
array([[ 10.19048513, -10.27613459,  -2.52608587,  17.01753001,
         -5.16577032],
       [-10.27613459,  11.76215006,   0.21222455, -22.39908929,
          7.41235845],
       [ -2.52608587,   0.21222455,   6.09763589,   6.90520506,
         -3.11356427],
       [ 17.01753001, -22.39908929,   6.90520506,  52.37155191,
        -18.20679122],
       [ -5.16577032,   7.41235845,  -3.11356427, -18.20679122,
        6.80726606]])
```

- qr-decomposition

In [26]:
```
#------------------------------------------------ CODE HERE ---
------------------------------------------------

print("q = \n",q)
print("\nr = \n",r)
```

```
q =
 [[-0.75089889 -0.03029281 -0.00466239 -0.61508654 -0.23849532]
 [-0.31829836 -0.79610385 -0.24416327  0.29694587  0.34221669]
 [-0.46353279  0.47645197 -0.419364    0.60133281 -0.14374826]
 [ 0.25170105 -0.34674275 -0.32921591  0.0382234  -0.84057831]
 [ 0.23795157  0.13440232 -0.8100082  -0.41283111  0.31428054]]

r =
 [[-6.40629175 -2.50157386 -4.09257196  2.48549622  2.6732475 ]
 [ 0.         -5.30433229  3.4848775  -2.36252414  1.07068254]
 [ 0.          0.         -0.42581102 -0.84379444 -2.57057492]
 [ 0.          0.          0.         -0.29001515 -0.83632347]
 [ 0.          0.          0.          0.          0.04616839]]
```

Commonly used `numpy.linalg` functions

In the next table, you can find a list of some of the most commonly used linear algebra functions.

Function	Description
`diag`	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
`dot`	Matrix multiplication
`trace`	Compute the sum of the diagonal elements
`det`	Compute the matrix determinant
`eig`	Compute the eigenvalues and eigenvectors of a square matrix
`inv`	Compute the inverse of a square matrix
`pinv`	Compute the Moore-Penrose pseudo-inverse of a matrix
`qr`	Compute the QR decomposition
`svd`	Compute the singular value decomposition (SVD)
`solve`	Solve the linear system Ax = b for x, where A is a square matrix
`lstsq`	Compute the least-squares solution to `Ax = b`

source: [3]

References

[1] **[BOOK]** Python Data Science Handbook, J.VanderPlas

[2] **[BLOG]** NumPy Scalars

[3] **[BOOK]** Python for Data Analysis, 2nd Edition

[4] **[ARTICLE]** Linear Algebra Review and Reference