

## Python3 – part2

**Chapter 10. NumPy array**

**Chapter 11. Matplotlib**

### Chapter 10. NumPy arrays

NumPy is a library designed for numerical computing in Python. It provides users with a versatile N-dimensional array object for storing data, and powerful mathematical functions for operating on those arrays of numbers.

NumPy is widely used in scientific computing, data analysis, and machine learning applications in research fields as diverse as physics, chemistry, astronomy, geoscience, biology, psychology, materials science, engineering, finance, and economics. For example, in astronomy, NumPy was an important part of the software stack used in the discovery of gravitational waves and in the first imaging of a black hole. Solving complex problems often requires working in N-dimensional spaces, which are implemented by N-dimensional arrays. Many datasets can be represented as arrays of numbers. For example, digital images can be thought of as simply two-dimensional arrays of numbers representing pixel brightness across the area. Sound clips can be thought of as one-dimensional arrays of intensity versus time.

Efficient storage and manipulation of numerical arrays is thereby very important in different research fields.

NumPy arrays are an homogeneous block of data, which means its elements are of the same type (fixed-type), and the size is fixed as well, and via a process known as vectorization, NumPy enables a degree of computational efficiency (running time and memory usage) that is otherwise unachievable by the Python language.

In this course we will focus on vectors (1D arrays) and matrices (2D arrays) of numbers, and we will go over a few fundamental concepts for organizing, exploring, and analyzing scientific numeric data.

To use the NumPy module, we need to import the module, and we can use its popular alias np.

```
import numpy as np
```

You can also keep the same name numpy instead, but the prescribed method allows us to use the abbreviation ‘np’ throughout our code, instead of having to write ‘numpy’.

The NumPy documentation is very rich, and we suggest you also read the online documentation beside these Notes.

<https://numpy.org/doc/stable/user/basics.html>

## 10.1 NumPy arrays vs lists

Here we will go over differences and similarities between a ndarray and a list type.

type	items	mutability	size
list	Heterogeneous	mutable	change
numpy. ndarray	Homogeneous	mutable	fixed

### type

First, we can convert a list in a ndarray, by using the **array()** function of the np module.

```
import numpy as np
L1=[7,2,9,10]
print(L1)
[7, 2, 9, 10]
```

```
v1=np.array(L1) #converts one list to a 1D array
print(v1)
[7 2 9 10]
```

They both use square braked, but the elements in a list type are separated by commas, whereas elements of a ndarray are separated by spaces.

```
print(type(v1))
<class 'numpy.ndarray'>
```

The data type of ndarray objects is numpy.ndarray.

### items

In addition to using the type() function, in NumPy we can also check the types of the elements of a ndarray by using the **dtype** attribute.

Remember that a ndarray is a homogeneous type, which means all the elements (items) of a ndarray are of the same type (or dtype), like all integers, or all floats.

```
print(v1.dtype) #dtype returns the data type of the elements
int64
```

In this case all the elements are int64, which means are all integer type.

In this course we will focus on arrays of integer (int64), float (float64) and Boolean (bool) dtypes.

Let's have a list of numbers, where only one element is float, and the other elements are integer. We convert this list to a ndarray.

```
L1=[7.2,2,9,10]
```

```
v1=np.array(L1)
print(v1)
print(v1.dtype)
[ 7.2  2.  9. 10. ]
float64
```

Notice that all the elements of the ndarray are now all float dtype (float64), and that 2.0 is represented as 2. Numpy does not show the 0 after the dot.

We know that a list is an heterogeneous type, and so we can have a list containing for example integers, float, and strings, like in this case. If we convert such heterogeneous list to a homogeneous ndarray, we see that all the elements of the ndarrays are now >U32, which means are all string type.

```
L1=[7.2,2,9,10,'pop']
```

If we convert such heterogeneous list to a homogeneous ndarray

```
v1=np.array(L1)
print(v1)
print(v1.dtype)
['7.2' '2' '9' '10' 'pop']
<U32  #unicode string
```

we see that all the elements of the ndarrays are now >U32, which means that all elements are string type.

The advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Fixed-type NumPy-style arrays lack this flexibility but are much more efficient for storing and manipulating data. The array essentially contains a single pointer to one contiguous block of data. The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object.

In this course, we will not talk about arrays of strings in NumPy. We will focus on integer and float.

## Mutability

Both types, lists and ndarrays, are mutable types, and so we can perform item assignments. Moreover, ndarrays can be indexed or sliced as the lists.

```
L1[0]=10      #[10, 2, 9, 10]
v1[0]=10      #[10 2 9 10]
```

We changed the first element of L1 and v1 to 10 by using the same syntax.

Now we want to change the values of a slice of L1 and v1.

```
L1[:3]=[0,0,0] #[0, 0, 0, 10]
v1[:3]=0       #[ 0  0  0 10]  #the value is propagated to the entire
selection. Broadcasting
```

Notice that to change multiple values in a list type, we would need to write down a new value for each element of the selection.

In NumPy, the new value (in this case 0) is instead propagated to the entire selection, and this feature is called broadcasting. Think about changing 100 values in a list type. By storing data in a ndarray the substitution would be easier and faster.

## size

The size of a list type can dynamically change. There are indeed several methods to add and remove elements of a list type, and it is the list object itself that dynamically can change. Indeed, if we check the id of L1 before and after, for example, we append an element, we see that the id is the same. This means that L points to the same object, which got modified.

```
id(L1) #140471496928832
L1.append(100) #[0, 0, 0, 10, 100]
id(L1) #140471496928832
#id is the same. Size can dynamically change.
```

The size of a ndarray, i.e., the number of elements, is instead fixed, and if we try to change the size by, for example, adding or removing elements of a ndarray, a new ndarray will be created. If we check the id of v1 before and after adding one element, we see that is different. This means that v1 now points to a new ndarray object containing the modified values.

```
id(v1) #140471497006512
v1=np.append(v1,[100]) #[ 0  0  0 10 100]
id(v1) #140471497048784
#id is different. Size is fixed, and a new ndarray object is created.
```

## High-performance array operation

Numeric calculations are much easier and faster with ndarray than list, because ndarrays are built as homogeneous type.

Let's store integer numbers from 0 to 9 in a list, L1 and ndarray, v1. The arange() function of NumPy for integer numbers works similarly to the range function.

```
L1=list(range(10)) #[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v1=np.arange(10)  #[0 1 2 3 4 5 6 7 8 9]
```

Now we want to calculate the cube of each number.

With a list type, we can achieve this by using comprehension or a loop.

```
L2=[i**3 for i in L1] #[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

With a ndarray type we can just do the power of 3, and this operation is applied to each element of the ndarray, without the need for looping over each value. This feature is called vectorization.

```
v2=v1**3          #[ 0  1  8 27 64 125 216 343 512 729]
```

Another example. We want to calculate the root mean square of each number. With a list type, we should import the math module, and we can use comprehension.

```
import math
L3=[math.sqrt(i) for i in L1]
```

With a ndarray, we can directly use the sqrt function on v1, and the function is applied to each element of the ndarray. There is no need to import the math module, because there are many mathematical functions available in NumPy, such as sqrt().

```
v3=np.sqrt(v1) #no need to import math
```

## type conversion lists/ndarrays

**np.array() - lists to ndarrays**

The array() built-in function of the np module converts a list and a tuple to a ndarray type.

```
L1=[7,2,9,10]
v1=np.array(L1) #converts a list to a 1D array
[ 7  2  9 10]
```

```
L2=[[5.2,3,4],[9.1,0.1,0.3]]
M=np.array(L2) #converts a list of lists to a 2D array
[[5.2 3.  4. ]
 [9.1 0.1 0.3]]
```

Notice the representation of a 2D array in NumPy. There are 2 square brackets.

There is also an optional keyword **dtype**, which specifies the type of the created array regardless of the type of the input elements.

```
T2=(-1,5,7)
v2=np.array(L2, dtype=float)
print(v2)
print(type(v2))
[-1.  5.  7.]
<class 'numpy.ndarray'>
```

Notice that here we converted a tuple of integer numbers into a 1D array of floats.

**ndarray.tolist() - ndarrays to lists**

The **ndarray.tolist() method** of the np module converts a ndarray type to a list type. Apply a method to a ndarray object.

```
L11=v1.tolist() #converts a 1D array to a list
[7, 2, 9, 10]
```

```
L22=M.tolist() #converts a 2D array to a list of lists
[[5.2, 3.0, 4.0], [9.1, 0.1, 0.3]]
```

## 10.2 Attributes of an array: shape, size, and axis

In this section we go over the difference between a 1D and a 2D array, by using the following ndarray attributes.

<code>ndarray.ndim</code>	number of axes, or dimensions, of the array
<code>ndarray.shape</code>	tuple of integers that indicate the number of elements stored along each dimension of the array.
<code>ndarray.size</code>	total number of elements of the array.
<code>ndarray.dtype</code>	the type of the elements of the array

Below is an example of a 1D array. A 1D array has only one axis, and the shape is tuple of this form (n,) where n is the number of elements. After the comma there is nothing, because a 1D array has only one dimension. We simply call a 1D array a vector.

```
print(v1)
[1 2 9 10] #vector

print(v1.ndim) # 1 axis
print(v1.size) # 4 elements
print(v1.shape) # (4,) 4 elements in one dimension
print(v1.dtype) # int64
```

1D array

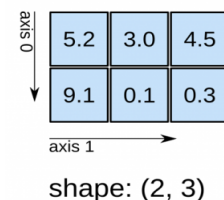


A 2D array has 2 axes, and the shape is of the form (r,c), which is tuple type, with r being the number of rows, and c the number of columns. Notice that the attribute size returns the total number of elements. Do not confuse the size attribute with the shape attribute.

```
print(M)
[[5.2 3. 4.5] #matrix
 [9.1 0.1 0.3]]

print(M.ndim) # 2 axes
print(M.size) # 6 elements
print(M.shape) # (2, 3) 2 rows and 3 columns
print(M.dtype) # float64
```

2D array



You can use the print function in Python to print ndarray objects, and you can use NumPy functions and attributes in the print function.

**Furthermore, a 1D array is represented by a pair of brackets [ ], and a 2D array by a double pair of brackets, [ [ ] ].**

## 10.3 Create arrays with built-in functions

### 10.3.1 Create 1D and 2D array of random numbers

To create ndarrays of random numbers there is no need to import the random module, since it comes within NumPy.

The built-in function [random.randint\(\)](#) of NumPy is similar to the one from the random module, but here you can also specify the dtype of the elements, which is by default integer. You can also set the size, which sets the shape of the ndarray.

```
random.randint(low, high=None, size=None, dtype=int)
It generates random integers in range [low, high).
If high is None (the default), then results are from [0, low).
If size is None returns one value.
```

```
v=np.random.randint(1,10, size=10)  #1D array, 1 axis
[7 6 3 2 3 9 9 7 4 7]
```

```
np.ndim(v)
1
```

We call just a vector a 1D array, which has only one dimension.

A matrix is a 2D array.

```
M=np.random.randint(5, size=(2,4)) #2D array, a matrix
[[3 1 4 4]
 [4 2 0 4]]
```

We call a row vector a 2D array of shape (1,c) where the number of rows r=1.

```
vr=np.random.randint(1,10, size=(1,3)) #2D array, a row vector
[[1 7 4]]
```

We call a column vector a 2D array of shape (r,1) where the number of columns c=1.

```
vc=np.random.randint(1,10, size=(3,1)) #2D array, a column vector
[[3]
 [1]
 [1]]
```

The **random.random(size)** function of NumPy is similar to the one of the random module, and it can be used to generate random floats in the half-open interval [0.0, 1.0)

```
np.random.random(size=None)
generates random floats in the half-open interval [0.0, 1.0)
see also random samples\(\)
```

```
v1=np.random.random(3) #1D array
[0.5769529  0.42219241 0.89814836]
```

```
M1=np.random.random((2,3)) #2D array - a matrix
[[0.97055086 0.31126001 0.55647421]
 [0.16726144 0.99078792 0.75163153]]
```

### 10.3.2 Create 1D array with arange()

**np.arange()** function creates arrays with regularly incrementing values. You want to use this function when you want to create a 1D array containing elements within a range, with the possibility to specify the step size between elements. The number of elements will then be calculated automatically. This built-in function returns a 1D array.

<b>arange(stop)</b>	values generated within [0, stop)
<b>arange(start, stop)</b>	values generated within [start, stop)
<b>arange(start, stop, step)</b>	values generated within [start, stop) with spacing between values given by step

To generate arrays of integer numbers, the arange() function works in the same way as the range function.

```
a=np.arange(3)      # [0 1 2]
b=np.arange(3,7)    # [3 4 5 6]
c=np.arange(1,10,2) # [1 3 5 7 9]
```

You can also create a ndarray of float numbers:

```
d=np.arange(1,3,0.3) # [1.  1.3 1.6 1.9 2.2 2.5 2.8]
e=np.arange(1,3,0.5) # [1.  1.5 2.  2.5]
```

### 10.3.3 Create 1D array with linspace()

The function **linspace()** is useful to create an array of evenly spaced numbers. You want to use this function when you want to create a 1D array within a range and want to specify the number of elements. The step size between elements is then calculated automatically. It returns a 1D array.

```
np.linspace(start, stop) creates array of 50 (default) evenly spaced numbers
over the interval [start, stop]

np.linspace(start, stop, num=N) creates array of N evenly spaced numbers over
the interval [start, stop]
```



Try this, and you will see that it creates 50 elements evenly spaced between 0 and 1.

```
a=np.linspace(0, 1)
```

You can also decide how many elements to generate, and `linspace()` will figure out the step.

```
b=np.linspace(2.0, 3.0, num=5) # create 5 float elements in range [2.0,3.0]
[2.    2.25 2.5   2.75 3.   ]
```

### 10.3.4 Create 1D and 2D array with: `zeros()`, `eye()`, `ones()`

There are many built-in functions for creating `ndarrays`. In this section we will go over the `zeros()` function.

```
M=np.zeros((2, 3)) #tuple (2,3) defines the shape: 2 rows, 3 columns
print(M)
[[0. 0. 0.] #matrix
 [0. 0. 0.]]
```

```
vc = np.zeros((3,1)) #tuple (3,1) defines the shape: 3 rows, 1 column
print(vc)
[[0.] #column vector
 [0.]
 [0.]]
```

```
vr = np.zeros((1,3)) #tuple (1,3) defines the shape: 1 row, 3 columns
print(vr)
[[0. 0. 0.]] #row vector
```

```
v = np.zeros((3,)) #tuple (3,) defines the shape: 1D array with 3 elements
print(v)
[0. 0. 0.] #1D array
```

Look at the documentation page and learn to use the functions `ones()` and `eye()`.

<https://numpy.org/doc/stable/reference/routines.array-creation.html>

### 10.3.5 Create an array from existing arrays: concatenation.

You can concatenate `ndarrays` by using built-in functions `vstack()` and `hstack()`. These are useful up to dimension 3D. For higher dimensions see [concatenate\(\)](#).

```
np.vstack(tup) Stack arrays in sequence vertically (row wise).
np.hstack(tup) Stack arrays in sequence horizontally (column wise).
tup is sequence type of ndarrays, like a tuple or a list of ndarrays
```

```
A=np.ones ( (2,3) )
```

```
[[1. 1. 1.]  
[1. 1. 1.]]
```

```
B=np.random.randint(10,size=(2,3))
```

```
[[5 7 5]  
[0 1 5]]
```

```
H = np.hstack( [ A, B ] )      # horizontal concatenation
```

```
[[1. 1. 1. 9. 1. 6.]  
[1. 1. 1. 0. 1. 4.]]
```

```
V = np.vstack( [ A, B ] )      # vertical concatenation
```

```
[[1. 1. 1.]  
[1. 1. 1.]  
[9. 1. 6.]  
[0. 1. 4.]]
```

Notice that if one ndarray is integer and the other is float, the result of the concatenation is a ndarray of float numbers.

### 10.3.6 reshape(), transpose() and flatten().

Let's create a 1D array, and see how functions reshape, transpose, and flatten work.

```
v = np.arange(1,11)
```

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

The transpose function returns the transposed array, where columns and rows are switched.

```
Mt=np.transpose(M)      #transpose a matrix
```

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

The reshape function is used to create an array with a different shape out of an existing array. The size (the number of elements) of the new array must match the size of the original array.

```
M = np.reshape(v, (2,5)) #create a new array with shape (2,5)
```

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

The flatten method returns a 1D array.

```
v1=M.flatten()          #create a 1D array
```

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

## 10.4 Indexing on ndarrays

Indexing into an array is a means of selecting a subset of elements from the array. NumPy has several indexing styles that are not only powerful and flexible, but also readable and expressive.

Ndarrays can be indexed using the standard Python syntax:

```
arr[obj]
```

where *arr* is the array and *obj* is the selection.

There are different kinds of indexing available depending on *obj*:

Basic indexing: Single element indexing  
Slicing

Advanced indexing: Integer array indexing (Fancy Indexing)  
Boolean array indexing (Masking)

### 10.4.1 Basic indexing - 1D array

Single element indexing and slicing of 1D array work exactly like for other standard Python sequences, like lists. Below the syntax

<code>arr[index]</code>	select one element at index
<code>arr[start:end]</code>	slice from index start through index end-1
<code>arr[start:]</code>	slice from index start through last index
<code>arr[:end]</code>	slice from index 0 through index end-1
<code>arr[start:end:step]</code>	slice from index start through but not past end, by step

Let's generate a 1D array, `arr1`, of integer numbers from 1 to 10.

```
arr1 = np.arange(1,11)
[ 1  2  3  4  5  6  7  8  9 10]
```

We can represent this array by using this pictorial representation.

1	2	3	4	5	6	7	8	9	10	← <b>arr1</b>
0	1	2	3	4	5	6	7	8	9	← indices

Notice that similarly to Python sequences, indices start from 0. The first element is referenced by index 0, the second by index 1 and so on.

To select a single element, we use one index. To select the first element, we use index 0.

```
arr1[0]
1
```

Notice that you can also use negative indices, as you did in a list type. Index -1 references the last element of a 1D array

```
arr1[-1]
10
```

Below we go over some examples of slicing.

```
arr1[3:8] #slice from element at index 3 to element at index 7
[4 5 6 7 8]
```

```
arr1[5:] #slice from element at index 5 to the last element
[6 7 8 9 10]
```

```
arr1[:5] #slice from 1st element to element at index 4
[1 2 3 4 5]
```

```
arr1[1:8:2] #slice from element at index 1 to element at index 7 by step of 2
[2 4 6 8]
```

## 10.4.2 Basic indexing - 2D array

### Single element Indexing - 2D array

In a 2D array, to access one element, you need two indices: one for selecting the row and another for selecting the column.

```
arr[index_row, index_col] select one element of a 2D array
```

		axis 1		
		0	1	2
axis 0	0	0,0	0,1	0,2
	1	1,0	1,1	1,2
	2	2,0	2,1	2,2

Let's generate a 2D array arr2.

```
arr2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]
```

We select the element at row 0 and column 0.

```
arr2[0,0] #same as arr2[0][0] as in nested lists  
1
```

If you only use one index, you will select a specific row.

```
arr2[0] #first row  
[1 2 3]
```

```
arr2[-1] #last row  
[7 8 9]
```

## Slicing - 2D array

The standard rules of list slicing apply to basic slicing on a per-dimension basis.

```
arr[slice_row, slice_col]    select a subset of a 2D array
```

Let's create this 2D array, arr2d.

```
arr2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])  
[[ 1  2  3  4]  
 [ 5  6  7  8]  
 [ 9 10 11 12]]
```

We can select specific columns, by writing before the comma the :, which selects all the row, and after the comma an index, to select a specific column.

In this example, we use slice notation to select the second column. This is useful to select fields, after reading in data and storing it in a 2D array.

```
arr2d[:,1] #second column  
[ 2  6 10]
```

1	2	3	4
5	6	7	8
9	10	11	12

We can also select a specific row, by writing before the comma an index to select a row, and after the comma the `:` to select all the columns.

In this example we select the first row, which is the same as writing `arr2d[0]`

```
arr2d[0,:] #first row, same as arr2d[0]
[1 2 3 4]
```

1	2	3	4
5	6	7	8
9	10	11	12

Here we select the first row, and every other column.

```
arr2d[0,::2]
[1 3]
```

1	2	3	4
5	6	7	8
9	10	11	12

And here an example of selecting a submatrix.

```
arr2d[1:,1:]
[[ 6  7  8]
 [10 11 12]]
```

1	2	3	4
5	6	7	8
9	10	11	12

### 10.4.3 Integer array indexing

Indexing with an integer array or list of integers allows selection of arbitrary items in the array.

This method is also called **fancy indexing**. It is like the simple indexing we've already seen, but we pass arrays of indices in place of single scalars. We will see some examples for Indexing a 1D and a 2D array.

```
arr[arr_indices] selection of multiple arbitrary elements of a 1D array
```

```
arr1 = np.arange(1,11)
[ 1  2  3  4  5  6  7  8  9 10]
```

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	10

**arr1**  
indices

```
arr1[[0,4,6]] # [0,4,6] is a list of indices
[1 5 7]
```

```
indarr=np.array([0,4,6]) #1D array of indices
arr1[indarr]
[1 5 7]
```

## Integer array Indexing - 2D array

For 2D array, two integer 1D arrays (or two lists) are needed, one for each dimension.

**arr[arr\_row, arr\_col]** selection of multiple arbitrary elements of a 2Darray

arr\_row is an array of indices for the rows, and arr\_col an array of indices for the columns.

```
arr2d
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

In this example, we want to select the element 2 referenced by indices [0,1], the element 8 referenced by indices [1,3] and the element 10 referenced by indices [2,1].

We make a list of indices for the rows [0,1,2] and one list of the columns [1,3,1].

```
arr2d[[0,1,2],[1,3,1]] #provide two lists, one for the
rows and another for the columns
[2 8 10]
```

	[0,1]		
1	2	3	4
5	6	7	8
9	10	11	12
	[2,1]		

[1,3]

We can also use a 1D array containing the indices for the rows, and a 1D array containing the indices for the columns.

```
arr2d[np.array([0,1,2]),np.array([1,3,1])]
[2 8 10]
```

## 10.4.5 Replacing values in 1D and 2D array

By using an indexing method on the left side of the equal sign, you can replace selected elements of an array.

```
arr[obj]=value
```

The value being assigned to the indexed array must be shape consistent (the same shape or broadcastable to the shape the index produces).

Here some examples:

```
arr1=np.arange(1,11)
```

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

```
arr1[0]=100
```

```
[ 100  2  3  4  5  6  7  8  9 10]
```

```
arr1[3:7]=12 #a scaler is broadcasted to the entire selection
```

```
[ 1  2  3 12 12 12 12  8  9 10]
```

```
arr1[5:]=arr1[5:]**2
```

```
[ 1  2  3  4  5 36 49 64 81 100]
```

```
arr2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

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

```
[4 5 6]
```

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

```
arr2[[0,1],[0,1]]=100,200] #shape consistent
```

```
[[100  2  3]
```

```
[ 4 200  6]
```

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

```
arr2[[0,1],[0,1]]=0 #a scaler is broadcasted to the entire selection
```

```
[[0 2 3]
```

```
[4 0 6]
```

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

To learn more about broadcasting read

<https://numpy.org/doc/stable/user/basics.broadcasting.html>



## 10.5. Vectorization

NumPy is optimized for operations involving N-dimensional arrays, including vectors and matrices. The process of revising loop-based, scalar-oriented code to use NumPy matrix and vector operations is called vectorization. Vectorization is worthwhile for several reasons:

- Appearance: Vectorized mathematical code appears more like the mathematical expressions found in textbooks, making the code easier to understand.
- Less Error Prone: Without loops, vectorized code is often shorter. Fewer lines of code means fewer opportunities to introduce programming errors.
- Performance: Vectorized code often runs much faster than the corresponding code containing loops.

**Vectorization is one of the core concepts of NumPy.** With one command it lets you process all elements of an array, avoiding loops and making your code more readable and efficient. Fortran, R and MATLAB have similar vectorization capabilities.

We will go over several vectorized features:

- Array Operations
- Vectorized functions or ufunc
- Boolean arrays and Indexing

### 10.5.1 Array Operations

Any arithmetic operations between equal-size arrays applies the operation elementwise.

<b>A+B</b>	<b>+</b>	<b>element by element addition</b>
<b>A-B</b>	<b>-</b>	<b>element by element subtraction</b>
<b>A*B</b>	<b>*</b>	<b>element by element multiplication</b>
<b>A/B</b>	<b>/</b>	<b>element by element division</b>
<b>A**B</b>	<b>**</b>	<b>element by element power</b>
<b>If is B a scalar it will be broadcast to all elements of A</b>		

We highlight that the `*` performs element by element multiplication, and not mathematical matrix multiplication.

```
A=np.ones((2,3))  
[[1. 1. 1.]  
 [1. 1. 1.]]
```

```
B=np.random.randint(10,size=(2,3))  
[[5 4 1]  
 [4 1 6]]
```

If we multiple an array by a scalar, the operation is applied to all the elements of the array:

```
B*2  
[[10 8 2]  
 [ 8 2 12]]
```

Below we see an example that the + sign performs element by element addition, and that if we add a float array and integer array, the result is a float array.

```
A+B  
[[6. 5. 2.]  
 [5. 2. 7.]]
```

**Array operations are useful for repetitive calculations.** For example, suppose you collect the temperature of 100 objects in Fahrenheit in a list called Tf, and you want to convert them in Celsius by using this formula:

$$T_c = (T_f - 32) / 1.8$$

For simplicity, we use this variable which contains less values:

```
Tf=[10.0, 30, 4, 100.7, 89, 89, 87.6, 60.7, 45.3, 100]
```

You can easily use NumPy elementwise operations. First, we convert the list into a 1D array Tv, and then use elementwise operations to implement the formula.

```
import numpy as np  
Tf=[10.0, 30, 4, 100.7, 89, 89, 87.6, 60.7, 45.3, 100]  
  
Tv=np.array(Tf)  
  
Tc=(Tv-32)/1.8  
  
print(Tc)  
[-12.22222222 -1.11111111 -15.55555556  38.16666667  31.66666667  
 31.66666667  30.88888889  15.94444444   7.38888889  37.77777778]
```

## 10.5.2 Universal Functions

Universal Functions perform fast element-wise array functions. A universal function, or ufunc, is a function that performs elementwise operations on data in ndarrays, thus removing the need for flow control loops. Example of ufuncs are: sqrt, exp, sin, cos, log, log10. Available ufunc for math operations can be found here: <https://numpy.org/doc/stable/reference/ufuncs.html#math-operations>

```
a = np.arange(1,4)  
[1 2 3]
```

In this example, the function `sqrt` is applied to each elements of `a`.

```
np.sqrt(a)
[1.          1.41421356  1.73205081]
```

Same here for the power function:

```
np.power(a,2)
[1  4  9]
```

### 10.5.3 Boolean arrays and Indexing

Boolean arrays can be used to index `ndarrays`. This method is also called **Boolean masking**. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

#### Creating Boolean arrays.

A logical extension of the bulk processing of arrays is to vectorize comparisons and decision making. NumPy comparison operators (logical and relational operations) accept array inputs and return Boolean arrays outputs. You can filter (or select) the elements of an array by applying conditions to the array.

```
Comparison operators
>
>=
<
<=
==
!=

logical_and, logical_or equivalent to operators & |
```

Let's make a 1D array.

```
arr1=np.arange(1,11)
[ 1  2  3  4  5  6  7  8  9 10]
```

Conditional expressions applied to an array return a Boolean array of the same size.

We apply the `>=` to the array, and the result is a Boolean array. A Boolean array is an array whose elements are `True` and `False`. The elements `True` are the ones satisfying the condition.

```
bool1=arr1 >= 6
[False False False False False True True True True True]
```

This is arr1:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

And this is the corresponding Boolean array bool1, where elements satisfying the conditions are True, and the rest False. For example, 6 in arr1 satisfies the condition, and the corresponding element in bool1 is True.

False	False	False	False	False	True	True	True	True	True
-------	-------	-------	-------	-------	------	------	------	------	------

## Use Boolean array for Indexing

A Boolean array provides a different type of array indexing in NumPy.

While most indices are numeric, indicating a certain row or column number, Boolean indices are positional. That is, it is the position of each True in the Boolean array that determines which array element is being referred to.

By using Boolean indices, we can extract the elements of arr1 that satisfy the condition (the True elements). In Boolean indexing, you use a single Boolean array for the subscript. NumPy extracts the array elements corresponding to the True values of the logical array.

```
v=arr1[bool1] #extract True elements
print(v)
print(v.ndim) #notice v is returned as 1D array
[ 6 7  8 9 10 ]
1
```

And you can also use them to replace those elements.

```
arr1[bool1]=arr1[bool1]*10 #replace True elements
[ 1 2 3 4 5 60 70 80 90 100 ]
```

Below some examples showing the use of Boolean arrays.

### ○ Extract elements satisfying a condition.

For example, suppose you collect data from 7 cones, which you store in variable D.

However, you record several negative values for the diameter, which are invalid. You can determine which elements in vector D are valid with the > operator (i.e., which cones have diameter greater than 0):

```
import numpy as np
V=np.array([0.0004,0.2618,1.3254,21.2058, 0.2618, 81.4640, 25.4500])
D=np.array([-0.2, 1.0, 1.5, 3.0, -1.0, 4.2, 3.14])
b1=D > 0
print(b1)
[False  True  True  True False  True  True]
```

The result is a Boolean array, `b1`. Each value in `b1` represents a logical True or logical False and indicates whether the corresponding element of `D` fulfills the condition `D > 9`.

Although `b1` contains information about which elements in `D` are greater than 0, it doesn't tell you what their values are. Rather than comparing the two vectors element by element, you can use `b1` to index into `D`.

```
D1=D[b1]
print(D1)
[1.  1.5  3.  4.2  3.14]
```

The result is an array of the elements in `D` that are greater than 0. Since `b1` is a Boolean array, this operation is indeed called **Boolean indexing**.

- **Count elements satisfying a condition.**

You can also answer the questions how many cones are valid, by using the `size` attribute, which returns the number of elements in an array.

```
print('There are', D1.size, 'valid cones')
There are 5 valid cones
```

More, you can directly exploit the Boolean indexing power of NumPy to select the valid cone volumes, `V1`, for which the corresponding elements of `D` are non-negative:

```
V1 = V[b1]
print(V1)
[ 0.2618  1.3254 21.2058 81.464  25.45  ]
```

- **Replace values satisfying a condition.**

Boolean indexing can also be used to replace the values in an array that meet a condition. For example, we want to replace the values of dimeters that are negative to positive.

```
D=np.array([-0.2, 1.0, 1.5, 3.0, -1.0, 4.2, 3.14])
vn=D < 0
print(vn)
[ True False False False  True False False]
```

Now, we use the expression we used for replacing values with other indexing method.

```
arr[obj]=value
```

Here, before the equal sign, we select the elements of `D` by using Boolean Indexing, and after the equal sign we select the same elements and make them positive by using the `abs()` function.

```
D[vn]=np.abs(D[vn]) # abs returns the absolute value of a number
print(D)
[0.2  1.   1.5  3.   1.   4.2  3.14]
```

## Comparing different arrays

You want to know if the elements of a matrix A are less than the corresponding elements of another matrix B. The less-than operator returns a logical array whose elements are True when an element in A is smaller than the corresponding element in B.

```
import numpy as np

A = np.array([[1,2,6],[ 4,3,6]])
print(A)
[[1 2 6]
 [4 3 6]]
```

```
B = np.array([[0,3,7],[ 3,7,5]])
print(B)
[[0 3 7]
 [3 7 5]]
```

```
ind = A < B
print(ind)
[[False  True  True]
 [False  True False]]
```

Now that you know the locations of the elements that meet the condition, you can inspect the individual values using ind as the index array. NumPy matches the locations of the value True in ind to the corresponding elements of A and B and lists their values in a 1D array.

```
Avals = A[ind]
print(Avals)
print(Avals.ndim). # Avals is a 1D array
[2 6 3]
1
```

## Use logical and, logical or

```
arr1=np.arange(1,11)
[ 1  2  3  4  5  6  7  8  9 10]
```

We will use the function `logical_and` to implement the logical and the function `logical_or` to implement the or.

Below we want to select the values that are greater than 8 and less or equal to 15. We apply the function and write the conditions inside the function. We store the Boolean array in bool2, and we use bool2 to extract the elements of arr1 that satisfy the conditions.

```
bool2 = np.logical_and( arr1 < 8 , arr1 >= 5 )  
[False False False False  True  True  True False False]
```

```
arr1[bool2]  
[5 6 7]
```

This is an example of logical or.

```
bool3 = np.logical_or( arr1 <=3 , arr1 > 9 )  
arr1[bool3]  
[ 1  2  3 10]
```

Usually, internal values of an interval are implemented by logical and, and external values of an interval are implemented by logical or.

## 10.6. Built-in Function useful for data analysis

In this section we will go over several built-in function useful for data analysis. The general form of these function is the following:

```
B=np.function(A, axis=n)  
  
axis=0: apply operation column-wise, across all rows for each column.  
axis=1: apply operation row-wise, across all columns for each row
```

Where A in the ndarray we want to apply the function to, and axis=n defines the dimension to which we want to apply the function by specifying a number n.

For a 2D array, axis=0 means we want to apply the function on each column and axis=1 on each row.

For a 1D array we can apply the functions without including the axis specification.

## Finding maximum and minimum - max, min, argmax, argmin

```
M=np.max(A, axis=n)      returns the maximum along a given axis.

m=np.min(A, axis=n)      returns the minimum along a given axis.

ind=np.argmax(A, axis=n) returns the indices of the maximum values along
an axis.

ind=np.argmin(A, axis=n) returns the indices of the minimum values along
an axis.
```

```
M = np.array([[1, 2, 3,4], [8, 6, 0,9], [9, 2, 3,12]])
[[ 1  2  3  4]
 [ 8  6  0  9]
 [ 9  2  3 12]]
```

We want to calculate the minimum value in each row, and we use the np.min() function, and set the axis=1

```
vm=np.min(M, axis=1)      #returns minimum of each row
[1 0 2]
```

The minimum function returns then a 1D array, which we store in vm, containing the minimum value of each row.

In conjunction with the min() function, the argmin() function is useful if we want to obtain the indices of the minimum values, and we can choose to obtain the indices of the minimum values of each row (axis=1) or column (axis=0)

```
indm=np.argmin(M, axis=1) #returns the indices of the minimum values of each
row
[0, 2, 1]
```

In this case we store in indm the indices of M corresponding to the minimum values in each row.

Notice that each row is indexed starting from 0.

The minimum value of the first row is 1, and its index is 0.

The minimum value of the second row is 0 and its index is 2.

The minimum value of the 3<sup>rd</sup> row is 2 and its index is 1.

If we apply the function to a 1D array, we do not include the axis.

```
v1=np.random.randint(1,50, size=6)
[22, 38, 30, 40, 18, 1]
```

Here we calculate the minimum value in v1, which is a scalar number, and its index, which is a scalar as well.

```
m=np.min(v1) #1 returns the minimum value
i=np.argmin(v1) #5 returns the index of the min value
```

The max() function works same way as the min() function.

Notice these are max() and min() functions of the NumPy module.



## Sorting - sort, argsort

```
S=np.sort(A, axis=n)  returns a sorted copy S of an array. Sort along axis.  
  
ind=np.argsort(A, axis=n) returns an array of indices of the same shape as  
A that index elements along the given axis in sorted order.
```

The sort function works similarly to the max and min function, because you can choose to sort an array along a particular dimension you can specify by axis.

```
M = np.array([[1, 2, 3,4], [8, 6, 0,9], [9, 2, 3,12]])  
[[ 1  2  3  4]  
 [ 8  6  0  9]  
 [ 9  2  3 12]]
```

Here, we sort each column of the 2D array M:

```
Ms=np.sort(M, axis=0)  #axis=0 sort by column  
[[ 1  2  0  4]  
 [ 8  2  3  9]  
 [ 9  6  3 12]]
```

Notice it sorts each column in ascending order, the largest element is the last.

With the function argsort() we obtain a 2Darray of the indices of M, which are rearranged to match the sorted matrix. It tells how the elements (the indices) of M should be rearranged in order to be sorted.

```
Mind=np.argsort(M, axis=0) #Returns indices that would sort M  
[[0 2 1 0]  
 [1 0 0 1]  
 [2 1 2 2]]
```

If you sort a 1D array, there is no need to write the axis.

```
vs=np.sort(v1)  #returns the sorted array  
[ 1, 18, 22, 30, 38, 40]
```

```
ind=np.argsort(v1) #returns the indices that would sort array  
[5, 4, 0, 2, 1, 3]
```

## Statistical functions – mean, std

```
m=np.mean(A, axis=n)  computes and returns the arithmetic mean along the
                        specified axis.

s=np.std(A, axis=n)   computes and returns the standard deviation along the
                        specified axis.
```

```
M = np.array([[1, 2, 3,4], [8, 6, 0,9], [9, 2, 3,12]])
[[ 1  2  3  4]
 [ 8  6  0  9]
 [ 9  2  3 12]]
```

```
m=np.mean(M, axis=0) #mean of each column
[6.      3.33333333 2.      8.33333333]
```

```
s=np.std(M, axis=1) #standard deviation of each row
[1.11803399 3.49106001 4.15331193]
```

Here you can find more statistical functions.

<https://numpy.org/doc/stable/reference/routines.statistics.html>

## Some math functions – sum, cumsum

```
s=sum(A, axis=n)  calculates and returns the sum of array elements over a
                  given axis.

c=cumsum(A, axis=n) returns the cumulative sum of the elements along a
                  given axis.
```

You can calculate the sum and the cumulative sum for each row or column by setting the axis.

```
M = np.array([[1, 2, 3,4], [8, 6, 0,9], [9, 2, 3,12]])
[[ 1  2  3  4]
 [ 8  6  0  9]
 [ 9  2  3 12]]
s=np.sum(M, axis=0) #sum of each column
[18 10  6 25]
```

```
c=np.cumsum(M, axis=1) #cumulative sum of each row
[[ 1  3  6 10]
 [ 8 14 14 23]
 [ 9 11 14 26]]
```

More math functions can be found here.

<https://numpy.org/doc/stable/reference/routines.math.html>

## 10.7. Import and Export a ndarray

### 10.7.1 Import data with loadtxt

```
A=np.loadtxt(filename, delimiter='sep') stores data in a 2D array
```

- We will use np.loadtxt() on files that contain numbers, and can have some comment lines.
- If the field separator is not a space, you should specify a delimiter character.
- 

Example: the file ph.dat is on Canvas.

```
dat = np.loadtxt('ph.dat') # the field separator of ph.dat is spaces
```

You can use slicing to store the first column in variable ph, and second column in variable p:

```
ph = dat[:,0]  
p = dat[:,1]
```

You can also define the dtype of the resulting array and skip comment lines.

<https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html>

### 10. 7.2 Exporting array with savetxt()

```
np.savetxt(filename, X, fmt='format', delimiter='sep') saves an array to a  
text file.
```

We use savetxt() to write numeric arrays in file. If you do not specify the fmt and delimiter, it will format as %.18e with a space as delimiter.

The format is applied to all numbers, and the syntax to format is as usual.

```
M = np.array([[1, 2, 3,4], [8,6, 0,9], [9, 2, 3,12]])  
[[ 1  2  3  4]  
 [ 8  6  0  9]  
 [ 9  2  3 12]]
```

```
np.savetxt('file1.txt', M, fmt='%d', delimiter=':') #save in file1.txt,  
format each number as integer, and set delimiter to colon.
```

```
np.savetxt('file1.csv', M, fmt='%.1f', delimiter=',') #save in file1.csv,  
format each number as float with 1 decimal precision, and set delimiter to  
comma.
```

You can also write the header, and comment lines. Look at the documentation page

<https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html>

## 10.8 Looping over ndarrays

Numpy arrays are iterable objects and so you can iterate over its elements with the usual python syntax.

**Iterate over the elements of a 1-D array:**

```
arr = np.array([1, 2, 3])

for x in arr:
    print(x)
```

```
1
2
3
```

**Iterate over the elements of a 2-D array:**

In a 2-D array the standard for loop will go over all the rows.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
    print(x)
```

```
[1 2 3]
[4 5 6]
```

To iterate over each scalar element of the 2-D array, we need a nested for loop (but see **np.nditer()**)

```
arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
    for y in x:
        print(y)
```

```
1
2
3
4
5
6
```

**Looping over elements of a ndarray by using nditer()**

To iterate over each scalar (element) of an array we need to use *n* for loops which can be difficult to write for arrays with very high dimensionality. To avoid nested loops, especially when the dimension of an array is large, we can use the **nditer()** function of the NumPy module. **nditer()** iterates over each scalar element of a ndarray (one by-one). Look at the documentation page of **nditer()** to learn more about it. (<https://numpy.org/doc/stable/reference/arrays.nditer.html>)

```
for x in np.nditer(arr):
    print(x)
```

```
1
2
3
4
5
6
```

Iterate over the scalar elements of a 3-D array by using **nditer()**

```
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

for x in np.nditer(arr):
    print(x)
```

```
1
2
3
4
5
6
7
8
```

You can utilize the Python function `enumerate()` to enumerate a NumPy ndarray, operating similarly to lists for 1D arrays. However, for arrays with higher dimensions, you should use NumPy's `ndenumerate()` function. Refer to: <https://numpy.org/doc/stable/reference/generated/numpy.ndenumerate.html>

## Chapter 11. Matplotlib: Visualizing data and Errors

Visualization tools are a very important part of scientific analysis because they provide ways to observe relationships between variables, see how data are distributed, understand trends, reveal outliers, clusters, and patterns in data, and to compare data.

Matplotlib is one of the most successful and commonly used libraries, providing various tools for data visualization in Python. It is one of the most powerful plotting libraries in Python.

**Matplotlib is based on NumPy, and numerical data should be stored in a ndarray type, but Matplotlib also works with numerical data stored in a list and tuple types, and a pandas Series.**

You can import Matplotlib and its submodule pyplot by using generic import and their common aliases:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

There is a variety of plots available in Matplotlib (plot, scatterplots, histograms, bar, errorbar, etc.) and we will go over some of the most used 2D plots for scientific visualization and describe when it is best to use one rather than another.

### 11.1 Object Oriented Interface

When plotting with Matplotlib, it is useful to know that there are **two interfaces**: the more traditional interface, the *pyplot interface (functional interface)* inspired by and modeled on MATLAB, *and* the more modern *Object-Oriented* interface (OO). In the Matplotlib documentation and on internet you will find examples of both. **In this Note we will use the OO interface.**

When using the OO interface we create **Figure** and **Axes** objects and call their methods to add content and modify the appearance.

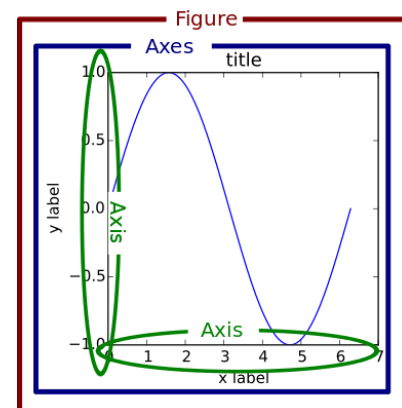
- **matplotlib.Figure**: create Axes, and handle figure-level content
- **matplotlib.Axes**: it is the Axes object to which we apply plotting methods and define plot elements like axis labels, axis styling, etc.

A Figure object is the outermost container for a matplotlib graphic, which can contain multiple Axes objects. One source of confusion is the name: Axes translates into what we think of as an individual plot or graph (rather than the plural of “axis,” as we might expect).

You can think of the Figure object as a box-like container holding one or more Axes (actual plots). Below the Axes in the hierarchy are smaller objects such as tick marks, individual lines, legends, and text boxes. Almost every “element” of a chart is a Python object, all the way down to the ticks and labels.

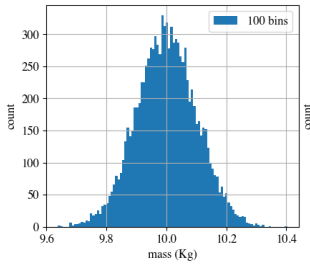
It helpful to look at the [anatomy of a Figure in Matplotlib](#).

There are very helpful [Matplotlib Cheatsheets and Handouts](#) you can use. One useful handout for beginners can be found [here](#).



Matplotlib offers different plot types (which are methods of the Axes Object), and here you find a summary of several plot types we will use in this course, and their use.

Axes method	plot type	use	figure plot
<code>ax.plot(x, y, '-b')</code>	Line plot	track changes over time, visualize mathematical functions	
<code>ax.plot(x, y, 'dg')</code>	plot with markers - scatterplot	Visualize possible relationships between two parameters, Visualize experimental data	
<code>ax.bar(x,energy)</code>	bar graph	compare data between different groups	
<code>ax.bar(x,mean_values, yerr=stderr)</code>	bar graph with error bar	compare data and errors between different groups	
<code>ax.errorbar(x,y,yerr)</code>	error bar	Visualize data and errors	

<code>ax.hist(data, 100)</code>	Histogram	show distribution of data	
---------------------------------	-----------	---------------------------------	---

### 11.1.1 Figure and Axes Objects

The basic idea when you plot with the OO interface is that you create a Figure Object, which is a container of Axes Objects. One Axes Object is one individual graph. Do not confuse the axes (x axis, and y axis) with the Axes Object. The x axis and y axis are elements of the Axes Object.

Here is a summary of the functions and methods we use in this course.

- **Use `plt.subplots()` function to create a Figure and Axes Objects**

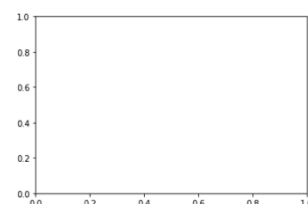
The first function to use is the [plt.subplots\(\)](#), which returns a tuple containing a Figure and Axes object(s).

```
plt.subplots(nrows, ncols) # default nrows=1 and ncols=1
```

In the example below, we create one Figure Object, and one Axes Object.

```
fig, ax = plt.subplots() #creates one Figure and one Axes - default
```

When using `fig, ax = plt.subplots()` we unpack the tuple into the variables `fig` and `ax`.



Let's check the type of `fig` and `ax`:

```
print(type(fig))
<class 'matplotlib.figure.Figure'>
```

```
print(type(ax))
<class 'matplotlib.axes._axes.Axes'>
```

You can also create an array of Axes. This code below will create a 2x2 matrix of Axes, where 2,2 is the shape of the array of Axes. This is a 2D array whose elements are Axes Objects.



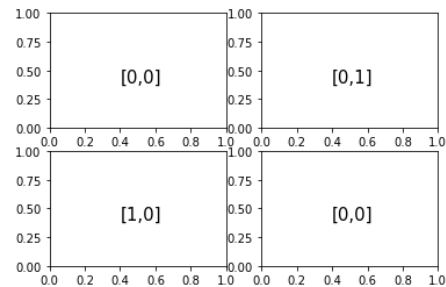
```
fig, axs = plt.subplots(2,2) #creates one Figure and a 2x2 array of Axes
```

When creating an array of Axes, it is common to reference it by using variable `axs`.

An array of Axes is used to make subplots.

Notice that each Axes Object can be accessed by using Basic Indexing (like in a 2D array in Numpy).

`axs[0,0]` references the Axes Object at row 0 and column 0.



The Figure and Axes Objects have methods.

- **Apply methods to the Axes Object**

The **Axes** Object represents one (sub-)plot in a figure. It contains the plotted data, axis ticks, labels, title, legend, etc. Its methods are the main interface for manipulating the plot.

Each plot type is a method that you apply to one Axes object. There are methods to label the axes, to make the legend, etc. Here we report several methods we will use in this course. In the next sections you will find examples of these methods.

- Choose the plotting method

```
ax.plot()  
ax.bar()  
ax.hist()  
ax.errorbar()
```

- Label the axis:

```
ax.set_xlabel()  
ax.set_ylabel()
```

- Show the grid:

```
ax.grid()
```

- Setting x and y limits:

```
ax.set_xlim()  
ax.set_ylim()
```

- Setting x and y ticks:

```
ax.set_yticks()  
ax.set_xticks()
```

- Show the legend:

```
ax.legend()
```

- Show the graph:

```
plt.show() # notice that show is a method you apply to plt
```

- **Apply methods to the fig object.**

Having the variable `fig`, which references the Figure Object, is useful for changing the figure-level attributes or for saving the figure as an image file, as shown below.

- Save the figure

```
fig.savefig('figurename.png') #saves figure as .png file
```

## Visually recognize the Object-Oriented Interface vs the Functional Interface:

### Object Oriented Interface

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

x = np.arange(0, 10, 0.1)
y = np.sin(np.pi * x) + x

fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_xlabel("x")
ax.set_ylabel("y")
```

### Functional Interface

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

x = np.arange(0,10,0.1)
y = np.sin(np.pi*x) + x

plt.figure()
plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("y")
```

## 11.2 plot(x,y)

We use the [plot method](#) to make a line plot (we set a line type) and scatterplots (we set a marker).

### 11.2.1 line plot

The `plot` method is used to draw lines between consecutive points, generating a line plot. Line plots can be used to draw functions, or to track changes over short and long periods of time. When smaller changes exist, line plots are better to use than bar plots.

We create a Figure `fig` and Axes `ax`. Then we call methods on them to plot data, add axis labels, and show the grid. Here we will plot the sin function.

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

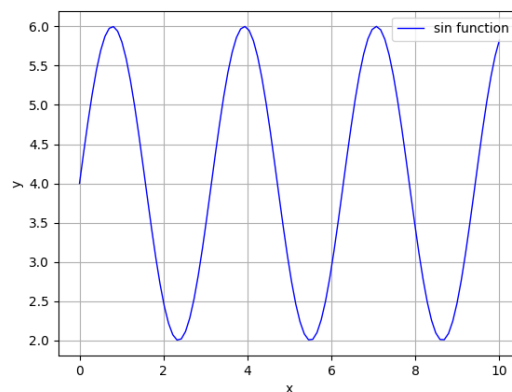
# make data
x = np.linspace(0, 10, 100)
y = 4 + 2 * np.sin(2 * x)
```

```
#Create one Figure object fig and one Axes object ax
fig, ax = plt.subplots()

#apply the plot method to the Axes object ax
ax.plot(x, y, '-b', linewidth=1, label='sin function') #blue solid line is
defined by the string '-b'

ax.set_xlabel('x') #label x axis
ax.set_ylabel('y') #label y axis

ax.grid() #show the grid
ax.legend() #show the legend. Text is reported in the label of plot
plt.show() #show the plot. Notice show is applied to plt.
```



### 11.2.2 Use the plot(x,y) method to make a scatterplot

You can use the plot method to make scatterplots when markers are identical in size and color. In the plot method you can omit the line type and add a marker instead to generate a plot a scatter plot, where points are labeled with markers, and not connected with lines. Usually, experimental data is reported by using markers, and not solid lines, since the data set does not have continuous values.

A scatterplot is useful to determine relationships between the two different variables. The x-axis is used to measure one event (or variable) and the y-axis is used to measure the other. If both variables increase at the same time, they have a positive relationship. If one variable decreases, while the other increases, they have a negative relationship. Sometimes the variables don't follow any pattern and have no relationship.

However, to make a scatter plot, you can also use the [scatter\(\)](#) method.

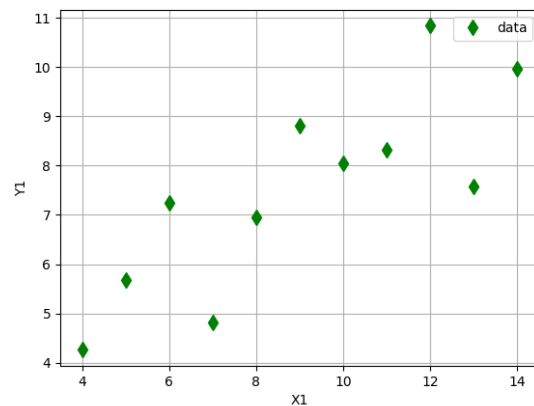
```
import matplotlib.pyplot as plt
import numpy as np

#dataset from
#https://en.wikipedia.org/wiki/Anscombe%27s quartet
```

```
x=np.array([10.0, 8.0, 13.0, 9.0, 11.0, 14.0, 6.0, 4.0, 12.0, 7.0, 5.0])
y=np.array([8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68])

fig, ax = plt.subplots()
ax.plot(x, y, 'dg', markersize=8, label='data') #string 'dg' sets a green
diamond marker
ax.set_xlabel("X1")
ax.set_ylabel("Y1")

ax.grid()
ax.legend()
plt.show()
```



### 11.2.3 Specify line, marker color and type in plot()

```
ax.plot(x, y, 'ob-', linewidth=1, markersize=8) #example
```

A format string is a string type containing characters that specify markers, lines and colors.

A format string is given by '**marker line color**' #each of them is optional

Example format strings:

```
'og'    # green circles
'-b'    # blue solid line
'--'    # dashed line with default color
'^k:'   # black triangle up markers connected by a dotted line
```

You can find more information [here](#)

Here we report some characters for the format string

Markers		Line Styles		color	
.	point marker	-	solid line style	b	blue
o	circle marker	--	dashed line style	g	green
v	triangle down marker	-.	dash-dot line style	r	red
s	square marker	:	dotted line style	k	black
*	star marker				
D	diamond marker				

## 11.3 Drawing multiple plots in a single Axes Object

Sometimes it is useful to compare different curves, and you want them on the same Axes Object.

To add another plot to the same Axes (window) just apply to the same Axes object another plotting method. Here, we show how to add multiple line plots.

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

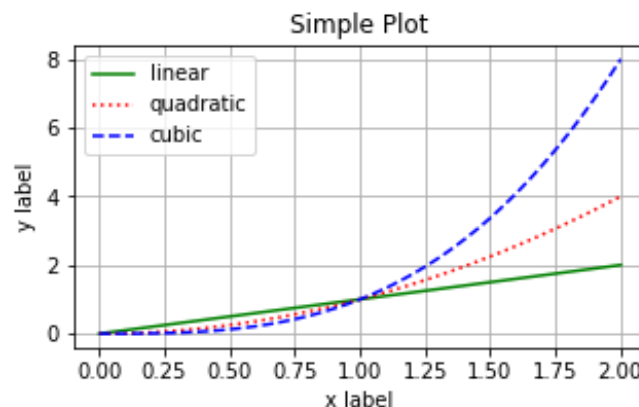
x = np.linspace(0, 2, 100) # Sample data.

fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(x, x, '-g', label='linear')      #Plot some data on the axes.
ax.plot(x, x**2, ':r', label='quadratic') #Plot more data on the axes.

ax.plot(x, x**3, '--b', label='cubic')    # ... and some more.

ax.set_xlabel('x label')      #Add an x-label to the axes.
ax.set_ylabel('y label')     #Add a y-label to the axes.

ax.set_title("Simple Plot") #Add a title to the axes.
ax.legend() #Add a legend
ax.grid()
```



## 11.4 Graph customization

Here we will show how we can customize a plot. These methods apply also to a bar, scatter etc.

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

```
#Set font size and font name with rcParams
plt.rcParams['font.size']=12
plt.rcParams['font.family']='Ariel'

# make data
x = np.linspace(0, 12, 100)
y = 4 + 2 * np.sin(2 * x)
y1= 4 + 4 * np.sin(2 * x)

#Create one figure object fig and one Axes object ax
fig, ax = plt.subplots()

#apply methods to the Axes object ax

ax.plot(x, y, '-b', linewidth=1, label=r'sin($\Theta$) function')
ax.plot(x, y1, '--r', linewidth=1, label=r'cos($\Theta$) function')

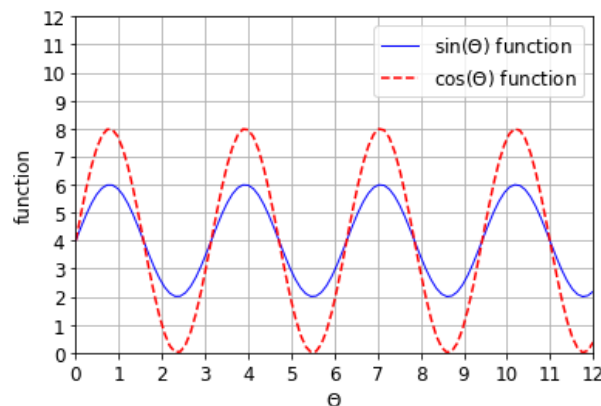
ax.set_xlim(0, 12) #sets x axis limit
ax.set_ylim(0, 12)

ax.set_yticks(np.arange(0,13)) #setting x ticks
ax.set_xticks(np.arange(0,13))
ax.legend() #shows in legend the strings reported in the label in plot
#you can label the axis with Greek symbols by following the Tex syntax

ax.set_xlabel(r'$\Theta$')
ax.set_ylabel(r'sin($\Theta$) ')

ax.grid()
plt.show() #shows the plot

#apply method to the figure object
fig.savefig('test', dpi=300) #saves figure with 300 dpi by default as png
```



### 11.4.2 Write Mathematical Expressions and Greek Symbols

You can use a subset of TeX markup in any Matplotlib text string by placing it inside a pair of dollar signs (\$).

Any text element can use math text. You should use raw strings (precede the quotes with an 'r'), and surround the math text with dollar signs (\$), as in TeX

```
ax.set_title(r'$\alpha > \beta$')
```

$$\alpha > \beta$$

More information [here](#)

### 11.4.3 Use rcParams to change default settings – e.g. change font size, font name

Here we will show how we can change the visual appearance of plots by changing default settings via rcParams. All rc settings are stored in a dictionary-like variable called rcParams, which is global to the matplotlib package.

rcParams is a like-dictionary type, storing different settings.

To see the default settings type:

```
import matplotlib.pyplot as plt
print(plt.rcParams)
```

You can change the default rc (runtime configuration) settings, by using the same syntax you would use to replace values in a dictionary type.

```
import matplotlib.pyplot as plt

#the rcParams must be before the plt.subplots()

plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.linestyle'] = '--'
plt.rcParams['font.size']=12
plt.rcParams['font.family']='Ariel'
plt.rcParams['figure.subplot.wspace']= 0.4 #set width of the padding between
subplots

plt.rcParams['figure.subplot.hspace']= 0.3 #set the height of the padding
between subplots
```

In a subplot, these commands will change the settings of all the Axes Objects in the array of Axes, i.e., for all the subplots. See the [Matplotlib documentation page of rcParams](#) for a full list of configurable rcParams.

### 11.4.4 Set the legend

There are [different ways to make a legend](#), and here we will go over one of them, which is the automatic detection of elements to be shown in the legend taken from the label parameter of the plotting methods.

Each plotting method has a parameter called label which references a string type. That string will be automatically detected by the `ax.legend()` method, which places a legend on the Axes.

In this example we only have one plot method

```
ax.plot(x, x, '-g', label='linear')      #Plot some data on the axes.
ax.legend()    # detect the string of the label parameter of the plot method
               and construct the legend.
```

In this example, we only have one bar method:

```
ax.bar(x, energy, label="Energy data")
ax.legend() #shows the string in the label
```

In this example, we have multiple plotting methods on the same Axes.

```
ax.plot(x, x, '-g', label='linear')      #Plot some data on the axes.
ax.plot(x, x**2, ':r', label='quadratic') #Plot more data on the axes.
ax.plot(x, x**3, '--b', label='cubic')    # ... and some more.
ax.legend()    # detect each string of each label in the plot methods and
               construct the legend.
```

The same procedure can be used with `errorbar()` and `hist()` methods.

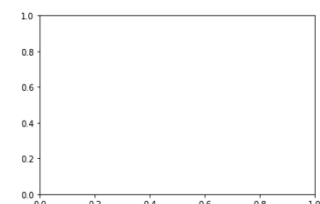
## 11.5 Subplots

Sometimes it is useful to compare multiple plots in one figure. To make subplots, instead of making a figure with a single Axes object, you can make a figure with an array of Axes objects.

The shape of the Axes array is set by a tuple and the resulting array can be stored in a variable.

The variables `ax` for a single Axes and pluralized `axs` for an array of Axes are usually used.

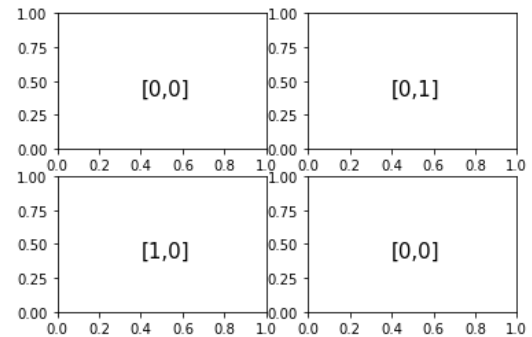
```
#make a single Axes object, called ax
fig, ax = plt.subplots()
ax.plot()
ax.set_xlabel()
```





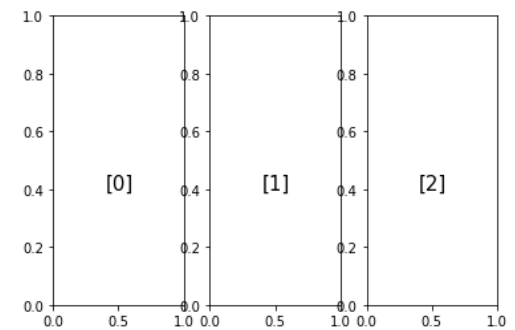
```
#make a 2x2 matrix of Axes Objects, called axs
fig, axs = plt.subplots(2, 2)
axs[0,0].plot() #apply to Axes [0,0]
axs[0,0].set_xlabel()

axs[0,1].plot() #apply to Axes [0,1]
```



Do not confuse Axes with Axis. The Axes is the object containing the x Axis and y Axis.

```
#make a row vector of 3 Axes Objects, called axs
fig, axs= plt.subplots(1, 3)
axs[0].bar() # apply to Axes [0]
axs[1].bar() # apply to Axes [1]
```



```
#You can use tuple unpacking to assign a variable to each Axes object
fig, (ax1, ax2) = plt.subplots(1, 2)
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2)
```

## Example

Here we will go over a subplot example. Notice that now each Axes is referenced by two indices (one for the row and another for the column, same as for a 2D array), and we apply methods on that Axes element. We will use the Latex syntax to write [mathematical symbols](#), the [rcParams](#) to change default settings, and the [figsize](#) parameter in the subplots function, to set the size of each Axes.

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

#rcParams should be placed before plotting
plt.rcParams['font.size']=15
plt.rcParams['font.family']='Times'
plt.rcParams['figure.subplot.wspace']=0.3 #set width of the padding between
subplots
```

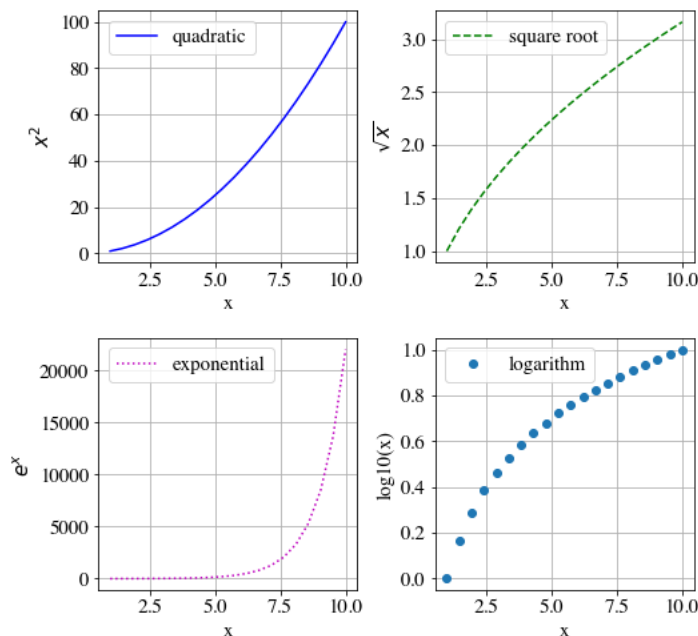
```
plt.rcParams['figure.subplot.hspace']=0.3 #set the height of the padding
between subplots

fig,axs = plt.subplots(2,2, figsize=(8, 8)) #set the size of the entire
figure to 8x8 inches
x = np.linspace(1,10,num=20)

axs[0,0].plot(x,x*x, '-b', label='quadratic')
axs[0,0].set_xlabel('x')
axs[0,0].set_ylabel(r'$x^2$')
axs[0,0].grid()
axs[0,0].legend()
axs[0,1].plot(x,np.sqrt(x), '--g', label='square root')
axs[0,1].set_xlabel('x')
axs[0,1].set_ylabel(r'$\sqrt{x}$')
axs[0,1].grid()
axs[0,1].legend()

axs[1,0].plot(x,np.exp(x), ':m', label='exponential')
axs[1,0].set_xlabel('x')
axs[1,0].set_ylabel(r'$e^x$')
axs[1,0].grid()
axs[1,0].legend()

axs[1,1].plot(x,np.log10(x), 'o', label='logarithm')
axs[1,1].set_xlabel('x')
axs[1,1].set_ylabel('log10(x)')
axs[1,1].grid()
axs[1,1].legend()
plt.show()
```



[Here](#) find more examples of subplotting.

In the next section, we show how we can plot different functions in different subplots by using a for loop.

### 11.5.1 Using loops to make subplots.

Sometimes is possible to make a repeated task and generate subplots within a for loop. We can use enumerate, and the flatten method to avoid nested loops over a matrix of Axes.

For a row or column vectors of Axes, there is no need to use flatten. The Axes Object is iterable, so you can use the usual python for loop syntax to loop over it, and keep in mind that now the Objects are the Axes.

```
import matplotlib.pyplot as plt
import numpy as np
plt.rcParams['font.size']=15
plt.rcParams['font.family']='Times'
plt.rcParams['figure.subplot.wspace']=0.4 #set width between subplots
plt.rcParams['figure.subplot.hspace']=0.3 #set the height

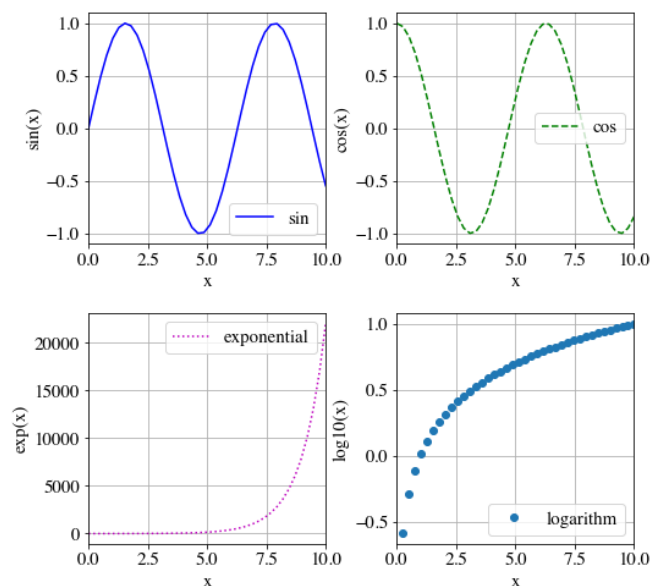
#We create a 2x2 matrix of Axes.
fig,axs = plt.subplots(2,2, figsize=(8,7)) #set each Axes 8x7 inches

x = np.linspace(1,10,num=40)

#make lists that contain data we use within the loop
formats=['b-', 'g--', 'm:', 'o']
labels=['sin', 'cos', 'exponential', 'logarithm']
func=[np.sin,np.cos,np.exp,np.log10] # ufunc
ylabels=['sin(x)', 'cos(x)', 'exp(x)', 'log10(x)']

#we use flatten to make one for loop and avoid a nested loop over a matrix.
for i,j in enumerate(axs.flatten()): # we flat the matrix of Axes
    j.plot(x,func[i](x),formats[i],label=labels[i])
    j.set_xlabel('x')
    j.set_ylabel(ylabels[i])
    j.set_xlim(1,10)
    j.set_xticks(np.arange(1,10))
    j.grid()
    j.legend()

plt.show()
```



## 11.6 Why using the Object-Oriented approach.

To understand the difference between the OO and Pyplot approaches read this [here](#).

The main reasons to learn the OO approach:

- It can be accompanied with other Python libraries, such as pandas.  
For data analysis, you can use pandas library, that allows to easily deal with large amount of non-homogeneous data by using the data frame type. You may want to draw a relationship between two variables, aggregate result by groups, show a trend in time-series or a correlation among parameters. Pandas provides you with a plot method and you can handle the plot by using the Axes Object. That's the object-oriented way of using it.
- **It can be utilized to make custom plotting function.**  
One of the main advantages of using OO style codes is that it can utilize the plotting functions and object concepts well. You can make your own functions, which for example can take one Axes, labels, title as parameters, and return the modified Axes. That increases re-usability of codes.

Here we go over an example, where we make a function called **myplot**, which returns a customized Axes.

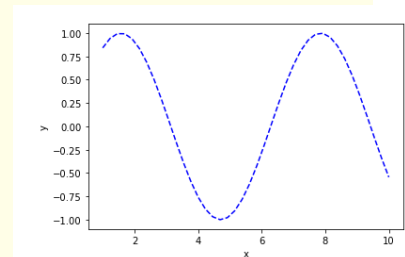
```
import matplotlib.pyplot as plt
import numpy as np

#This function takes parameters: the Axes, etc., and returns the
customized Axes Object

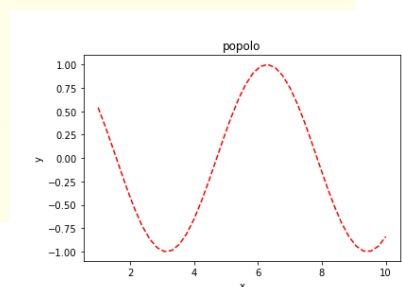
def myplot(ax0,xvalue,yvalue,formatstr, labelx, labely):
    ax0.plot(xvalue, yvalue,formatstr)
    ax0.set_xlabel(labelx)
    ax0.set_ylabel(labely)
    return ax0
```

```
x = np.linspace(1,10,num=40) y=np.sin(x)
```

```
fig, ax = plt.subplots()
out=myplot(ax,x,y,'--b', 'x','y') #call the function
plt.show()
```



```
fig, ax1 = plt.subplots() y1=np.cos(x)
out1=myplot(ax1,x,y1,'--r', 'x','y') # call the function
out1.set_title("popolo") # you can apply methods to
the returned object
plt.show()
```



## 11.7 bar(x,y) and barh(y,x)

The [bar\(\) method](#) makes a vertical bar graph, and the [barh\(\)](#) a horizontal bar.

Bar graphs are used to compare between different groups. We show an example of a vertical bar graph, where we compare the energy production for various energy sources.

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

x = ['Nuclear', 'Hydro', 'Gas', 'Oil', 'Coal', 'Biofuel']
y = np.array([5, 6, 15, 22, 24, 8])

#We can create a pandas DataFrame
```

```
energy_df = pd.DataFrame({'Source': x, 'Energy Production': y})
print(energy_df)
```

	Source	Energy Production
0	Nuclear	5
1	Hydro	6
2	Gas	15
3	Oil	22
4	Coal	24
5	Biofuel	8

We can make two variables, one storing the source and another the energy production:

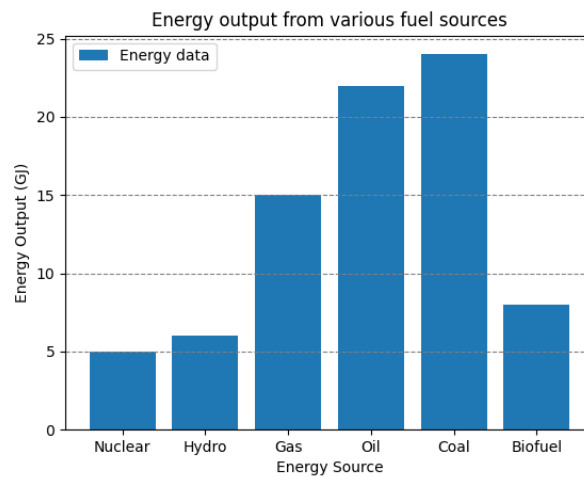
```
source = energy_df['Source']
energy = energy_df.loc['Energy Production']
```

```
fig, ax = plt.subplots()
ax.bar(source, energy, label="Energy data")

ax.set_xlabel("Energy Source")
ax.set_ylabel("Energy Output (GJ)")

ax.set_title("Energy output from various fuel sources") #Add title

# we set the grid to be only horizontal
ax.grid(visible=True, which='major', axis='y', color='grey', linestyle='--')
ax.legend() #shows the string in the label
plt.show()
```



The grid can be set in different ways. [Here](#) how to set the grid.

### More on Bar Graphs

[Here](#) find examples on how to set colors, make horizontal bar graphs and stick and group bars together

You can also make a horizontal bar graph by using the `barh(y,x)` method.

We use the same variables, and change `bar()` in `barh()`. In `barh(y,x)` the first argument (`y`) is plotted on the `y` axis, and the second (`x`) on the `x` axis.

In this example, we use in the `bar()` function with list and a ndarray, without converting into a DataFrame.

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

source = ['Nuclear', 'Hydro', 'Gas', 'Oil', 'Coal', 'Biofuel']
energy = np.array([5, 6, 15, 22, 24, 8])

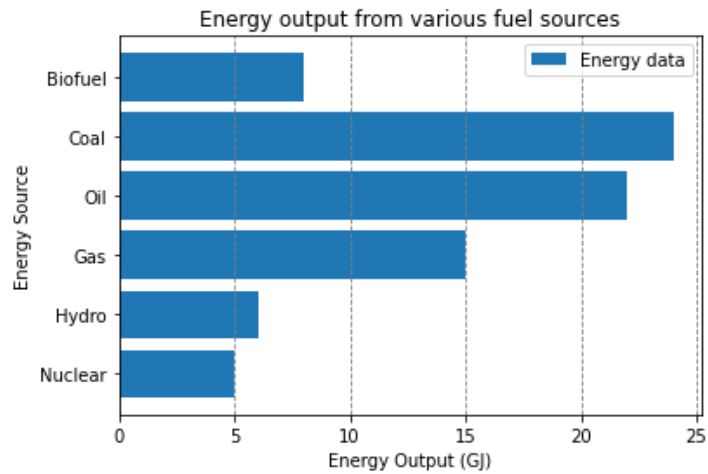
fig, ax = plt.subplots()

ax.barh(source, energy, label="Energy data")
ax.set_ylabel("Energy Source")
ax.set_xlabel("Energy Output (GJ)")

ax.set_title("Energy output from various fuel sources") #Add title

# we set the grid to be only vertical
ax.grid(visible=True, which='major', axis='x', color='grey', linestyle='--')
ax.legend() #shows the string in the label
plt.show()
```

The source is now plotted on the `y` axis, and the energy on the `x` axis. We have changed the `x` and `y` labels to match this. We also change the grid to be only vertical, by setting the `axis='x'`.



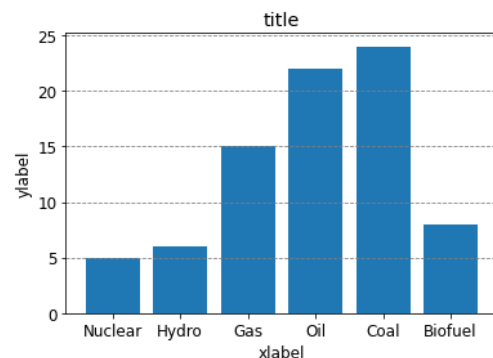
Here we go over an example, where we make a function called **mybar**, which returns a customized bar graph.

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

def mybar(ax0,xvalue,yvalue, labelx, labely, mytitle):
    ax0.bar(xvalue, yvalue)
    ax0.set_xlabel(labelx)
    ax0.set_ylabel(labely)
    ax0.set_title(mytitle)
    ax0.grid(visible=True, which='major', axis='y', color='grey',
            linestyle='--')
    return ax0
```

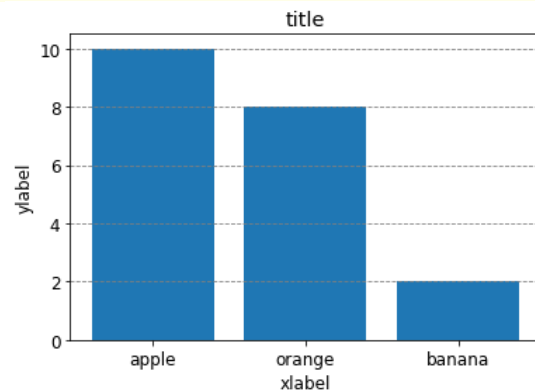
Call the function on a set of data.

```
x = ['Nuclear', 'Hydro', 'Gas', 'Oil', 'Coal', 'Biofuel']
energy = np.array([5, 6, 15, 22, 24, 8])
fig, ax = plt.subplots()
outax=mybar(ax,x,energy,'xlabel','ylabel','title')
plt.show()
```



Call the function on another set of data.

```
fruits = ['apple', 'orange', 'banana']
numbers = np.array([10, 8, 2])
fig1, ax1 = plt.subplots()
outax1=mybar(ax1, fruits,numbers,'xlabel','ylabel','title')
plt.show()
```



## 11.8 Visualizing Errors

Variability is an inherent part of the results of measurements and of the measurement process. Measurement errors can be divided into two components: *random* and *systematic*.

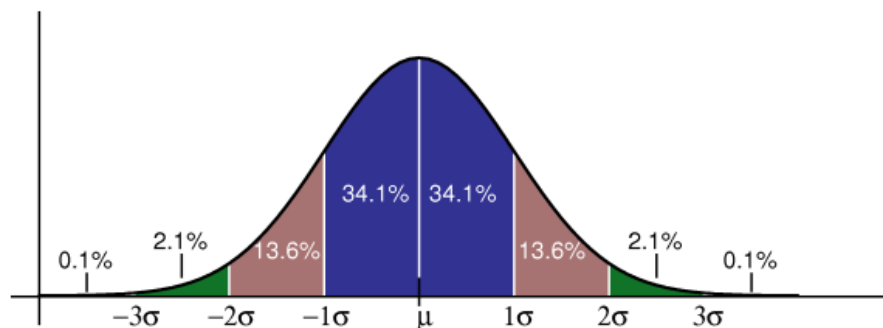
- **Random error** is always present in a measurement. It is caused by inherently unpredictable fluctuations in the readings of a measurement apparatus or in the experimenter's interpretation of the instrumental reading. Random errors show up as different results for ostensibly the same repeated measurement. They can be estimated by comparing multiple measurements and reduced by averaging multiple measurements. Random error can be caused by unpredictable fluctuations in the readings of a measurement apparatus, or in the experimenter's interpretation of the instrumental reading; these fluctuations may be in part due to interference of the environment with the measurement process. The concept of random error is closely related to the concept of precision. The higher the precision of a measurement instrument, the smaller the variability (standard deviation) of the fluctuations in its readings.
- **Systematic error** is predictable and typically constant or proportional to the true value. If the cause of the systematic error can be identified, then it usually can be eliminated. Systematic errors are caused by imperfect calibration of measurement instruments or imperfect methods of observation, or interference of the environment with the measurement process, and always affect the results of an experiment in a predictable direction. Incorrect zeroing of an instrument leading to a zero error is an example of systematic error in instrumentation.



In the analysis of statistical errors, the key concept is the **standard deviation** that we were calculating throughout the semester. Standard deviation is used to quantify the amount of variation or **dispersion** of a set of data values. A standard deviation close to 0 indicates that the data points tend to be very close to the **mean** of the set, while a high standard deviation indicates that the data points are spread out over a wider range of values.

### The standard deviation

Let's say a measurement of the mass of an object was repeated several times, and the mean value was  $\mu=10.0$  kg and standard deviation was  $\sigma=0.1$  kg. This means that 68.2% of measured values will be in the interval 9.9-10.1 kg, 95.4% of values will be in the interval 9.8-10.2 kg, and 99.6% of values will be in the interval 9.7-10.3 kg.

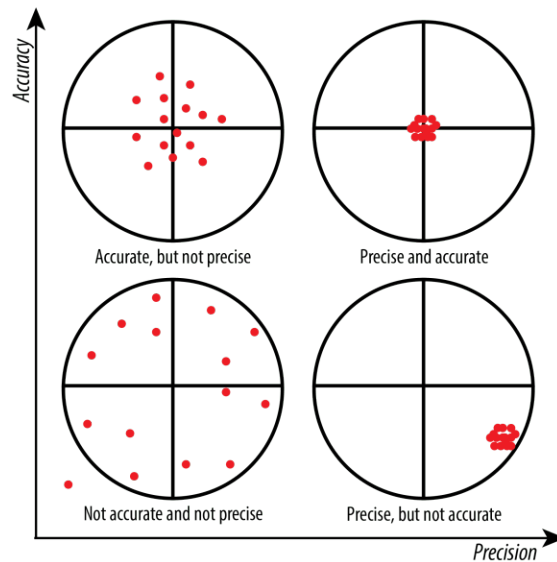


The probability distribution for a random error represented above is called a Gaussian distribution. When many independent random factors act in an additive manner to create variability, data will follow a bell-shaped distribution called the Gaussian distribution. This distribution is sometimes also called a normal distribution. Although no data follows this ideal mathematical distribution, many kinds of data follow a distribution that is approximately Gaussian. Such a distribution is characterized by two parameters,  $\mu$  the mean or average value, and  $\sigma$  the standard deviation. In data analysis the Gaussian distribution is called the limiting distribution for the type of measurement we have described, however it is not the only limiting distribution that occurs in error analysis. If measuring the same quantity multiple times with the same method, and assuming no systematic error, and if errors are random and independent, and if the number of measured values is large, measured values should follow Gaussian distribution.

### Precision and Accuracy

**Precision** refers to the closeness of two or more measurements to each other. Using the example above, if you weigh a given substance five times, and get 3.2 kg each time, then your measurement is very precise. Precision is independent of accuracy. You can be very precise but inaccurate, as described below. You can also be accurate but imprecise. The standard deviation measures the **precision** of a single typical measurement. It is common experience that the mean of a number of measurements gives a more precise estimation than a single measurement. This is quantified by the standard error of the mean.

**Accuracy** refers to the closeness of a measured value to a standard or known value. For example, if in lab you obtain a weight measurement of 3.2 kg for a given substance, but the actual or known weight is 10 kg, then your measurement is not accurate. In this case, your measurement is not close to the known value.



### 11.9.2 Visualizing errors using the `errorbar()` method

Experimental error is always with us; uncertainty is associated with every quantitative result. Error bars on a graph describe the uncertainty in the measurements. One way to represent the error bar in a measurement is the standard deviation.

**Standard error** is often used to quantify the error in your estimate of the mean value. It depends on the standard deviation, but also on how many times you repeated the measurement. If you measure the same quantity many times, your estimate of the mean will be better, and the error will decrease. If the measurement is repeated  $N$  times, the estimate of an error is given by the standard error:

$$SE = \frac{\sigma}{\sqrt{N}}$$

Where  $\sigma$  is the standard deviation, and  $N$  is the number of measurements (values).

We will use the `errorbar()` method to visualize errors. Here is an example, data are not from experiments.

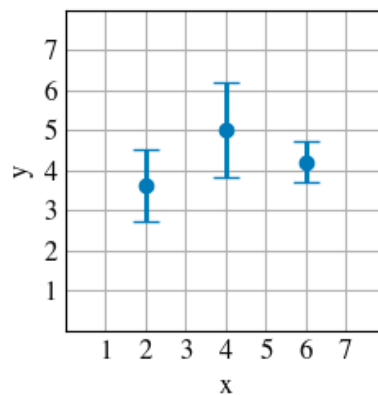
```

import matplotlib.pyplot as plt
import numpy as np
plt.rcParams['font.size']=14
plt.rcParams['font.family']='Times'

# make data
x = [2, 4, 6]
y = [3.6, 5, 4.2]
yerr = [0.9, 1.2, 0.5]

# plot
fig, ax = plt.subplots()
ax.errorbar(x, y, yerr, fmt='o', linewidth=2, capsize=6, color='b')
ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_xlim((1,7))
ax.grid()
plt.show()

```



### 11.9.3 Visualizing errors using the `bar()` or `barh()` methods

Bar charts with error bars are useful to show the error or precision in a set of measurements or calculated values. We use the measured coefficient of thermal expansion (CTE) of three metals: Aluminum, Copper and Steel. The units for coefficient of thermal expansion is per degrees Celsius ( $/^{\circ}\text{C}$ ).

We calculate the mean and standard error for each metal, and use a bar graph where we also visualize the standard error.

```

import matplotlib.pyplot as plt
import numpy as np
# Store raw data in 1D arrays
aluminum = np.array([6.4e-5, 3.01e-5, 2.36e-5, 3.0e-5, 7.0e-5, 4.5e-5, 3.8e-5, 4.2e-5, 2.62e-5, 3.6e-5])

```

```
copper = np.array([4.5e-5 , 1.97e-5 , 1.6e-5, 1.97e-5, 4.0e-5, 2.4e-5, 1.9e-5, 2.41e-5 , 1.85e-5, 3.3e-5 ])

steel = np.array([3.3e-5 , 1.2e-5 , 0.9e-5, 1.2e-5, 1.3e-5, 1.6e-5, 1.4e-5, 1.58e-5, 1.32e-5 , 2.1e-5])

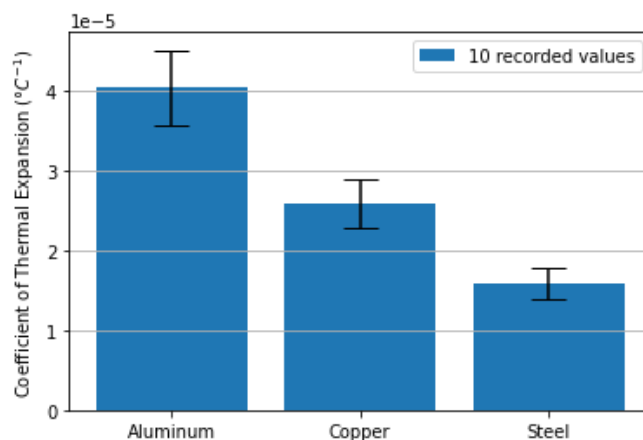
# Calculate the average
a_mean = np.mean(aluminum)
c_mean = np.mean(copper)
s_mean = np.mean(steel)

# Calculate the standard error
N=np.size(aluminum)
a_stderr = np.std(aluminum)/np.sqrt(N)
c_stderr = np.std(copper)/np.sqrt(N)
s_stderr = np.std(steel)/np.sqrt(N)

# Create lists for the plot
materials = ['Aluminum', 'Copper', 'Steel']
x_pos = np.arange(len(materials)) # need it to define the x values
L_mean = [a_mean, c_mean, s_mean]
L_stderr = [a_stderr, c_stderr, s_stderr]

# Build the plot
fig, ax = plt.subplots()
ax.bar(x_pos, L_mean, yerr=L_stderr, capsize=10, label='10 recorded values')
ax.set_ylabel('Coefficient of Thermal Expansion ( $^{\circ}\text{C}^{-1}$ )')

ax.set_xticks(x_pos) # set the x values
ax.set_xticklabels(materials) # set the x tick labels
ax.yaxis.grid(True) # set grid for only the y axis
ax.legend()
plt.show()
```



## 11.9 Statistical plots

### 11.9.1 hist(x, bins)

Histograms are a type of bar plot for numeric data that group the data into a series of adjacent bins. Histograms are useful for analyzing patterns of frequency in numerical data sets. In statistics, histograms are used to graph the probability distribution of the data.

In Matplotlib histograms are created with the `hist(x, bins)` method where `x` is a 1D ndarray, and `bins` is an integer number defining the number of bins.

`hist(x, bins)` method shows the counts of values that fall into each interval or bin. The most important [histogram-specific setting is the bin size](#), which is determined by the number of bins.

Here we generate random numbers from a Gaussian distribution, with mean 10 and std 0.1, and make two histograms with different number of bins.

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

plt.rcParams['font.size']=12
plt.rcParams['font.family']='Times'
# Generate numbers from a normal distribution
mu, sigma = 10, 0.1 # mean and standard deviation
an = np.random.normal(mu, sigma, 10000)

fig, axs = plt.subplots(1, 2, figsize=(10,4))

axs[0].grid()
axs[0].hist(an, 100, label='100 bins')
axs[0].set_xlabel('mass (Kg)')
axs[0].set_ylabel('count')
axs[0].legend()
axs[1].grid()
axs[1].hist(an, 10, label='10 bins')
axs[1].set_xlabel('mass (Kg)')
axs[1].set_ylabel('count')
axs[1].legend()
plt.show()
```

