Introduction to Python

Lecture notes

Contents

1 Python

Python is a high-level programming language characterized by a simple and readable syntax. Moreover Python is open-source and a wide community works continuously for its development. With Python we can easily create algorithms and applications for numerical computation, data analysis and models development. All these features make it very popular inside the science community.
To program with Python there are several compilers, we will use Jupyter which has a nice interface and allows also to write text and equations.

1.1 Install Anaconda

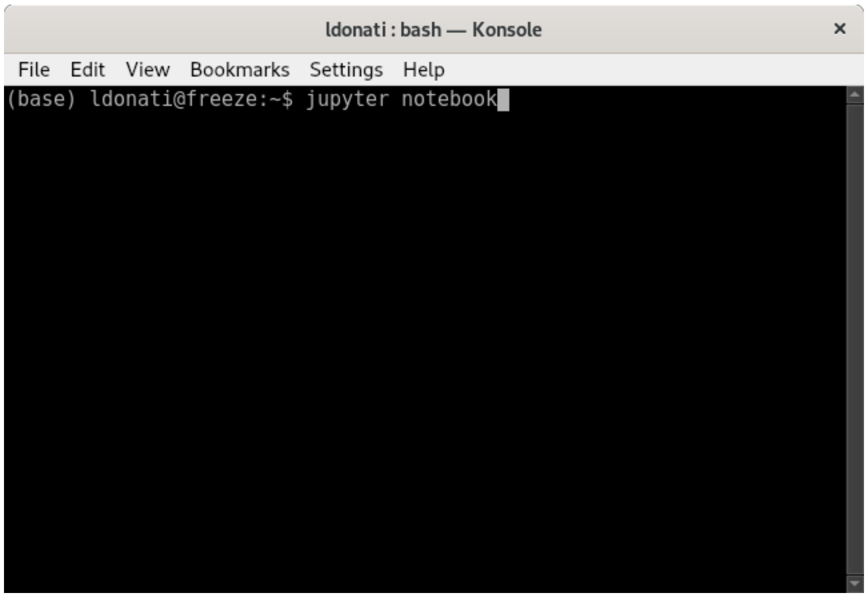
Anaconda is a software suite for Python that contains several libraries and utilities such as Jupyter.
To install Anaconda, you can follow these steps:

1. Go to the Anaconda download page: `https://www.anaconda.com/products/individual`
2. Download the appropriate installer for your operating system (Windows, macOS, or Linux).
3. Double-click the downloaded file to launch the Anaconda installer.
4. Follow the prompts in the installer to complete the installation process.
   Note: You may be asked to specify a location for the installation.

That's it! You should now have both Anaconda and Jupyter installed on your computer.
As an alternative, you can also use Anaconda in the cloud without installing it:
`https://www.anaconda.com/code-in-the-cloud`.

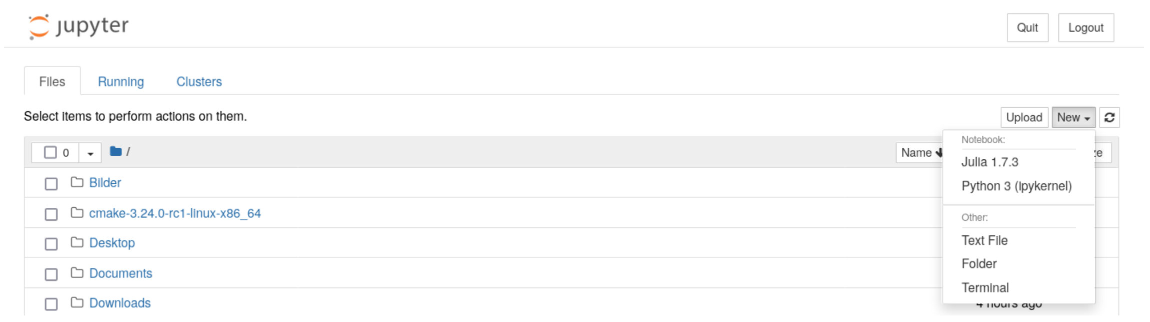
1.2 Start

Once the installation is complete, open your terminal (on Windows, open the Anaconda Prompt) and type "jupyter notebook" to launch Jupyter.



```
(base) ldonati@freeze:~$ jupyter notebook
```

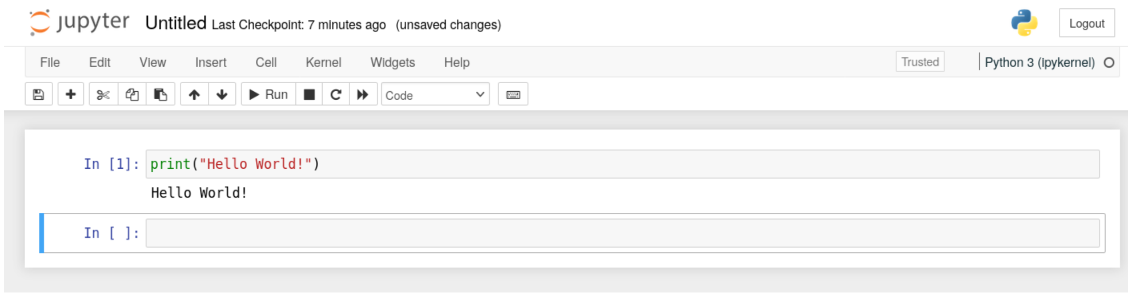
This command opens your web browser (Firefox, Chrome, etc.) and shows the directories in your computer. Choose a "work directory", or create a new one.
Then, click on **New** (right side top) and click on "Python 3 (Ipykernel)" to create a new Python file.



This command will open a new tab in your browser. Jupyter is made of "cells", i.e. sections within you can write commands or text. For example, write in the first cell the command:

```
print("Hello World!")
```

Then, press SHIFT+ENTER. This command will execute the instruction in your cell and it will print the sentence "Hello World!". Note that after you press SHIFT+ENTER, Jupyter will create a new cell where you can write new commands.



```
In [1]: print("Hello World!")
        Hello World!

In [ ]:
```

1.3 Variables

One of the most important component of Python (and in general of all programming languages), are the *variables*. A variable is a part of computer memory containing some data. In Python there are five different variables: numbers, strings, lists, tuples and dictionary. Moreover NumPy introduces a new variable called array. During this tutorial we will see only the Numbers and arrays, but at the end of this manual you can find information also about strings, lists and tuples.

1.3.1 Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example:

```
a = 5
```

You can also delete the reference to a number object by using the "del" statement:

```
del a
```

Python supports four different numerical types

1. int (integer numbers): **x=10**
2. float (real numbers): **x=10.8**
3. complex (complex numbers): **x=10+3J**

1.4 Operations

The algebric operations in Python are:

1. Sum

```
a = 5
b = 3
c = a + b
```

c=8

2. Difference

```
a = 5
b = 3
c = a - b
```

c=2

3. Multiplication

```
a = 5
b = 3
c = a * b
```

c=15

4. Exponent

```
a = 5
b = 3
c = a ** b
```

c=125

5. Division

```
a = 5
b = 3
c = a / b
```

c=1.6666666666666667

6. Floor Division (division ignoring decimals)

```
a = 5
b = 3
c = a // b
```

c=1.0

7. Modulus

```
a = 5
b = 3
c = a % b
```

c=2

1.5 NumPy

NumPy is an open source extension module for Python. It introduces a new type of variables, called "array" that we will use to define vectors and matrices (but it can define also n-dimensional objects).

To import NumPy there are three different methods:

1.

```
import numpy
```

We need to call any NumPy function, adding the prefix "numpy.", e.g.: numpy.array().

2.

```
import numpy as np
```

We can just use the prefix "np.", e.g.: np.array()

3.

```
from numpy import *
```

The prefix is not necessary anymore, e.g.: array().

The three methods are equivalent, but I suggest to use the second one, to remember which functions come from the NumPy module.
Another good reason to use the second method is that if we import (with the asterics) different modules that use the same name for different functions, then Python cannot distinguish them and the functions of the last module would override the functions of the other module.

1.5.1 Define an array

To define an array we need the function np.array().

```
a = np.array([1, 2, 3])  # Create a 1d array (vector)
b = np.array([[1,2,3],[4,5,6]])   # Create a 2d array (matrix)
```

It is also possible to specify if we want an array of int or float.

```
a = np.array([1, 2, 3],dtype=np.int64)  # Create a 1d array of int numbers
a = np.array([1, 2, 3],dtype=np.float64)# Create a 1d array of float numbers

#To declare a float array it is also possible to use the following syntax:
a = np.array([1., 2., 3.])
```

1.5.2 Other functions to create arrays

```
a = np.zeros((2,2))  # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                     #          [ 0.  0.]]"

b = np.ones((1,2))   # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7) # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                      #          [ 7.  7.]]"

d = np.eye(2)        # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                     #          [ 0.  1.]]"

e = np.random.uniform(0,1,(2,2))        #Matrix 2x2 with random values between 0 and 1
                                        #(extracted from a uniform distribution.

f = np.random.gaussian(0,1,(2,2)) #Matrix 2x2 with random values between 0 and 1
                                #(extracted from a Gaussian distribution
                                #with mu=0 and sigma=1).

g = np.linspace(0,5,10) # Return evenly spaced numbers over a specified interval
print(a)                        # prints array([ 0.   ,  0.55,  1.11,  1.66,
2.22,
                          #              2.77,  3.33,  3.88,  4.44,  5.  ])

b = a.copy()            #Copy the content of the array a in a new array b
```

1.5.3 Array indexing

One of the most important aspects of the array (but also of lists, tuples and strings) is the indexing system. The arrays are a sequence of numbers and to each element of the array corresponds an index.
If we have a vector (1d array) of N elements, the first elements has index 0, the last element has index $N-1$.
The following example shows how to extract elements from a vector using the indexing system.

```
a = np.array([10, 43, 53, 21, 62, 90])
print(a)        # Prints the complete array
print(a[0])     # Prints the first element of the array
print(a[2:5])   # Prints the elements starting from 3rd to 4th
print(a[2:])    # Prints the element starting from 3rd element
print(a[:2])    # Prints the first two elements of the array
print(a[5])     # Prints the last element
print(a[-1])    # Prints the last element
print(a[-2])    # Prints the second last element
print(a[-2:])   # Prints the last two elements
```

If we are working with matrices, we have to indicate the index of the row and of the column, with the syntax [row, column].

```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])      # Create the following
                                                        #2d array with shape (3, 4)
                                                        # [[ 1  2  3  4]
                                                        #  [ 5  6  7  8]
                                                        #  [ 9 10 11 12]]

b = a[0:2, 1:3] # Pull out the subarray consisting of the first 2 rows
                # and columns 1 and 2; b is the following array of shape (2, 2):
                # [[2 3]
                #  [6 7]]


a[0, 0] = 77            #to change the first element of the matrix
a[0:2, 0] = np.array([20,20])   #to change the first two elements of the first column
a[2, :] = np.array([20, 20, 20, 20])    #to change all the elements of the last row

#It is also possible to select and edit the elements
#of an array that respect some condition
a[a>6] = 30     #to change all the elements greater than 6
a[a>10,1] = 40  #to change all the elements of the second column greater than 6
```

1.5.4 Array operations

In the following example you can find the math operations that is possible to do with the array.

```
x = np.array([[1.,2.],[3.,4.]]) #matrix 2x2
y = np.array([[5.,6.],[7.,8.]]) #matrix 2x2
v = np.array([9.,10.])  #vector 1x2
w = np.array([11., 12.])  #vector 1x2

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y
print(np.subtract(x, y)

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the 1d array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the 2d array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

1.5.5 Array functions

Numpy provides many useful functions for performing computations on arrays, the full list can be found here:
http://docs.scipy.org/doc/numpy/reference/routines.math.html

The following script shows some of the most important functions.

```
x = np.array([[1,2],[3,4]]) #define a matrix

#sum()
np.sum(x)  # Compute sum of all elements; prints "10"
np.sum(x, axis=0)  # Compute sum of each column; prints "[4 6]"
np.sum(x, axis=1)  # Compute sum of each row; prints "[3 7]"

#diagonal()
d = x.diagonal()  #d is a 1d array containing the elements of the diagonal

#transpose
y = x.T         #y is the transpose of x

#inverse
z = np.linalg.inv(x)

#trace
trace = np.trace(x)     #trace of x
```

```
#eigenvalues
eigenvalues = np.linalg.eig(x)[0]       #eigenvalues

#eigenvectors
eigenvectors = np.linalg.eig(x)[1]      #eigenvectors (in columns)

#floor
a = np.array([-1.2, 0.2, 1.8, 9.9])
b = np.floor(a)         #floor() approximates the array toward the
                        #first smallest integer
print(b) #[-2.  0.  1.  9.]

#determinant
np.linalg.det(x)
```

1.5.6 Math functions

The following are typical math functions that operate on numbers and arrays.

```
import numpy as np

x = np.array([0, 1, 2, 3])

#sine
np.sin(x)

#cosine
np.cos(x)

#tangent
np.tan(x)

#arcsine
np.arcsin(x)

#arccos
np.arccos(x)

#arctan
np.arctan(x)

#logarithm (base e)
np.log(x)

#exponential
np.exp(x)

#absolute value
np.abs(x)

#real part
np.real(x)

#imaginary part
np.imag(x)
```

1.6 Matplotlib

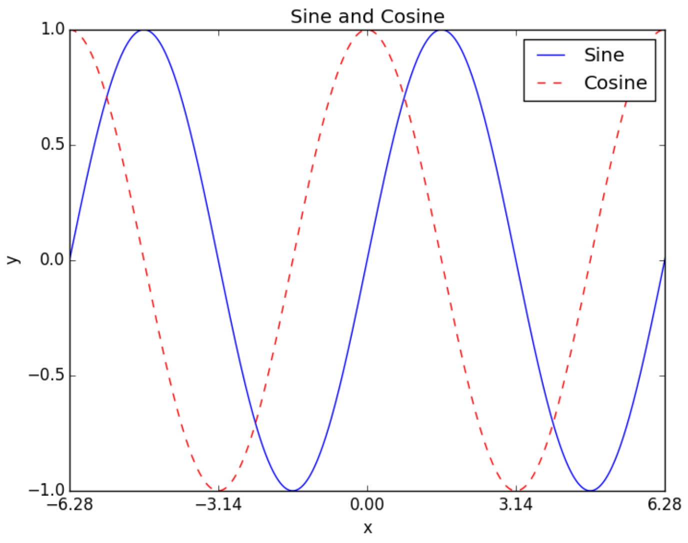
Matplotlib is another useful library that collects several functions to draw graphs. A complete list of the functions can be found here:
`http://matplotlib.org/gallery.html`

The following is an example to plot the function sin(x) and cos(x).

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

# Compute the x and y coordinates for points on sine and cosine curves
x = np.linspace(-10, 10, 1000)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin) #plot a blue line
plt.plot(x, y_cos, 'r--') #plot a red dashed line
plt.xlim(-2*np.pi,2*np.pi) #set the x range
plt.xticks(np.linspace(-2*np.pi,2*np.pi,5)) #set the position of the ticks
plt.xlabel('x')
plt.ylabel('y')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```

It is also possible to not open the graph and to save it directly in the working directory, replacing the last line with:

```
plt.savefig('name_figure.eps', format='eps', dpi=1000)
```



1.7 Control Statements: if, for, while loop

The control statements are blocks of instructions executed by Python (or in general by a programming language) only if particular conditions are satisfied or that can be repeated automatically many times. It is important to understand from the beginning that this kind of instructions slow the execution of a program, then, when it is possible, it is important to look for other solutions. In Python+NumPy for example, there is the possibility to *vectorize* an algorithm, but we will talk about this argument later.

1.7.1 For - loop

The "for statements" are a sequence of instructions repeated N times. To realize a for loop, we need a *counter* (i in the example), that increases its value at each iteration. The following example prints out all the element of a list:

```
x = np.array([10, 20, 30, 40, 50])
N = len(x)
for i in range(0,N):     #the counter i takes all the values of the list
 print(x[i])             #range(0,N) at each new iteration
```

the function range generates the indeces that we want to explore **range(N) = [0, 1, 2, 3, 4]**. As already explained, the indexes of an array go from 0 to $N-1$.
Another remark about the previous script, is that it is very important to leave a white space before **print(x[i])**, otherwise you will get an IndentationError. In fact the identation tells to Python which lines own to the block. Let's try a different code:

```
x = np.array([10, 20, 30, 40, 50])
N = len(x)
for i in range(N):
 print(x[i])
print("ok")
```

The result is:

```
10
20
30
40
50
ok
```

Let's now add a white space before **print("ok")**:

```
import numpy as np
x = np.array([10, 20, 30, 40, 50])
N = len(x)
for i in range(N):
 print(x[i])
 print("ok")
```

The result is:

```
10
ok
20
ok
30
ok
40
ok
50
ok
```

This time the instruction **print("ok")** was part of the block "for", and so Python printed out "ok" at each iteration. Let's add another white space before **print("ok")**:

```
import numpy as np
x = np.array([10, 20, 30, 40, 50])
N = len(x)
for i in range(N):
 print(x[i])
  print("ok")
```

This time, we will get an error:

```
print("ok")
^
IndentationError: unexpected indent
```

The reason is that, adding another white space, we are saying to Python that the instruction **print("ok")** owns to another block that actaully does not exist in this code.
A final remark, it is not important how many white spaces we use to define a block (I suggest to use at least three spaces or better a TAB space), but it is important that we use always the same number of white spaces inside the same block.

1.7.2 if - else statements

The "if statements" are a sequence of instructions executed only if a particular condition is satisfied. For example:

```
import numpy as np
x = 1
if x==1 :
        print("x is 1")
```

Also in this case, it is important to be careful with the identation.
The other comparison operators are:

- equal $==$
- not equal $!=$
- greater than $>$
- greater than or equal to $>=$
- smaller than $<$
- smaller than or equal to $<=$

We can also realize complex conditions using the logic operators "and" and "or":

```
x = 1
y = 3
z = 0
if ( x==1 and y<2 ) or z!=1 :
        print("true")
```

We are asking to Python: "If x is equal to 1 AND y is smaller than 2, OR z is not equal to 1" then print("true").
We can also add several options using "elif" (else if) and "else":

```
x = 21
if x == 1:
        print("1")
elif x == 2:
        print("2")
else:
        print("another number")
```

1.7.3 While loop

"While" is something similar to "for" and we will use it when we are asking to python to repeat some instructions, until a certain condition is reached. For example:

```
x = 0
while (x < 9):
        print("The count is:", x)
        x = x + 1
```

In this case we are asking to Python to increase the variable x until it is equal to 9.

1.8 Functions

A function is a block of code that is used to perform a specific calculation. Functions provide better modularity for your application and a high degree of code reusing. Some functions are already implemented in Python, for example **sum(x)** returns the sum of the elements of the list **x**. Other useful functions are **min(x)** (minimum element of a list), **max(x)** (maximum element of a list), **abs(x)** (absolute value of a number).
The function **help()** prints the description of a function (e.g.: help(sum)).
In the official Python documentation you can find a complete list of functions already implemented:
`https://docs.python.org/3/library/functions.html`

Other useful functions are contained in NumPy, but it is also possible to write your own functions, directly in the main program or in a separate file.
The following example function defines the distance between two points.

```
import numpy as np

def distance2points(r1, r2):
        # r1, r2 are numpy arrays
        result = np.sqrt( np.sum( ( r1 - r2 ) ** 2 ) )

        return result

r1 = np.array([-2, 2, 4])
r2 = np.array([3, 0, -1])

d  = distance2points(r1, r2)

print(d)
```

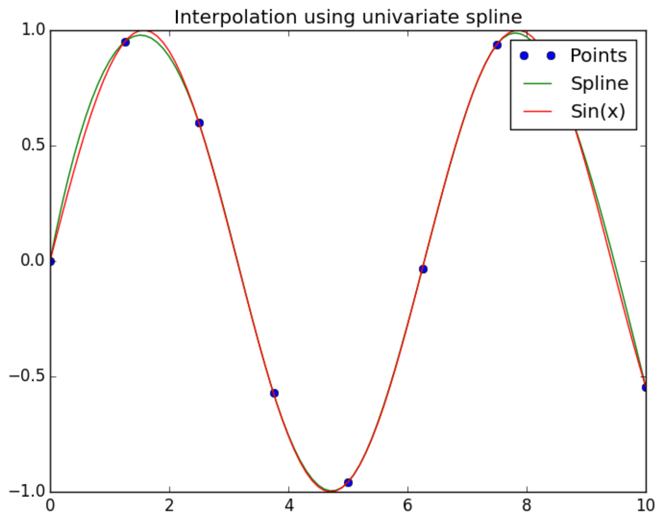
1.9 SciPy

Numpy provides a high-performance multidimensional array and basic tools to compute with and manipulate these arrays. SciPy builds on this, and provides a large number of functions that operate on numpy arrays and are useful for different types of scientific and engineering applications. The best way to get familiar with SciPy is to browse the documentation:
`http://docs.scipy.org/doc/scipy/reference/index.html`

The following example shows how to interpolate a set of points with the spline method imported from SciPy:

```
import numpy as np
from scipy.interpolate import InterpolatedUnivariateSpline
import matplotlib.pyplot as plt

# setup data
x = np.linspace(0, 10, 9)
```

```
y = np.sin(x)
xi = np.linspace(0, 10, 101)

# use fitpack2 method
ius = InterpolatedUnivariateSpline(x, y)
yi = ius(xi)


plt.plot(x, y, 'bo')
plt.plot(xi, yi, 'g')
plt.plot(xi, np.sin(xi), 'r')
plt.title('Interpolation using univariate spline')
plt.legend(['Points','Spline','Sin(x)'])

plt.show()
```



1.10 Other Python variables

1.10.1 Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks:

```
str = 'Hello World!'
```

It is possible to print out the content of a string using the "print" statement:

```
print(str)        # Prints complete string
print(str[0])     # Prints first character of the string
print(str[2:5])   # Prints characters starting from 3rd to 5th
print(str[2:])    # Prints string starting from 3rd character
print(str[:2])    # Prints the first two elements of the string
print(str[-1])    # Prints the last element
print(str[-2])    # Prints the second last element
print(str[-2:])   # Prints the last two elements
```

1.10.2 Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets.

```
lis1 = [1, 2, 3, 10, 50, 1000]
lis2 = ['abcd', 786 , 2.23, 'john', 70.2]
lis3 = [[1, 2, 3],[4,5,6],[7,8,9]]      #list of lists
```

We can also define a list as a sequence of numbers using the function range():

```
x = range(1,6)     # defines a list of numbers from 1 to 5
```

Another useful function is len(), to get the length of a list (or of a string):

```
len(x)
```

```
5
```

1.10.3 Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.
The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists.

```
lis1 = (1, 2, 3, 10, 50, 1000)
lis2 = ('abcd', 786 , 2.23, 'john', 70.2)
```

It is also possible to convert a list/tuple in a NumPy array.

```
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
b = np.asarray(a)
```