Before starting to do machine learning programming in Python, please install Python Anaconda distribution from https://anaconda.org/anaconda/python. After that we will use Jupyter as a web-based interactive notebook environment for writing and running code.

Anaconda Python by default contains all the basic required packages for starting programming in machine learning. But in case, any extra package is needed, 'pip' is the recommended installer.

Command: pip install

Syntax: pip install 'SomePackage'

After installing, libraries need to be loaded by invoking the

command:

Command: import <<library-name>>0

Syntax: import os

Basic data types in Python

Python has five standard data types −

1. Number

2. String

3. List

4. Tuple

5. Dictionary

Number:

Type/ Class	*Description*	*Examples*
Int	integer numbers	–12, –3, 0, 125, 2
Float	real or floating point numbers	–2.04, 4.0, 14.23
Complex	complex numbers	3 + 4j, 2 – 2j

Boolean data type (bool) is a subtype of integer. It is a unique data type, consisting of two constants, True and False.

Example

```
>>> num1 = 10

>>> type(num1)

<class 'int'>

>>> num2 = -1210

>>> type(num2)

<class 'int'>

>>> var1 = True

>>> type(var1)

<class 'bool'>
```

```
>>> float1 = -1921.9
>>> type(float1)
<class 'float'>
>>> float2 = -9.8*10**2
>>> print(float2, type(float2))
-980.0000000000001 <class 'float'>
>>> var2 = -3+7.2j
>>> print(var2, type(var2))
(-3+7.2j) <class 'complex'>
```

String

String is a group of characters. These characters may be alphabets, digits or special characters including spaces.For example,

```
>>> str1 = 'Hello Friend'
>>> str2 = "452"
```

List

List is a sequence of items separated by commas and the items are enclosed in square brackets [ ].

Example

```
#To create a list
>>> list1 = [5, 3.4, "New Delhi", "20C", 45]
#print the elements of the list list1
>>> print(list1)
[5, 3.4, 'New Delhi', '20C', 45]
```

Tuple

Tuple is a sequence of items separated by commas and items are enclosed in parenthesis ( ). This is unlike list, where values are enclosed in brackets [ ]. Once created, we cannot change the tuple.

Example

```
#create a tuple tuple1
>>> tuple1 = (10, 20, "Apple", 3.4, 'a')
#print the elements of the tuple tuple1
>>> print(tuple1)
(10, 20, "Apple", 3.4, 'a')
```

Dictionary

Dictionary in Python holds data items in key-value pairs. Items in a dictionary are enclosed in curly brackets { }. Dictionaries permit faster access to data. Every key is separated from its value using a colon (:) sign. The key : value pairs of a dictionary can be accessed using the key. The keys are usually strings and their values can be any data type. In order to access any value in the dictionary, we have to specify its key in square brackets [ ].

Example

```
#create a dictionary
>>> dict1 = {'Fruit':'Apple',
'Climate':'Cold', 'Price(kg)':120}
>>> print(dict1)
{'Fruit': 'Apple', 'Climate': 'Cold',
'Price(kg)': 120}
>>> print(dict1['Price(kg)'])
120
```

for–while Loops & if–else Statement

Syntax:

```
for i in range(<lowerbound>,<upperbound>,<interval>):
print(i*i)
```

Example: Printing squares of all integers from 1 to 3.

```
for i in range(1,4):
print(i*i)
1
4
9
for i in range(0,4,2):
print(i)
0
2
4
```

While loop

Syntax:

```
while < condition>:
```

<do something>

Example: Printing squares of all integers from 1 to 3.

```
i = 0
while i < 5:
print(i*i)
i = i + 1
```

1

4

9

if-else statement

Syntax:

```
if < condition>:
<do something>
else:
<do something else>
```

Example: Printing squares of all integers from 1 to 3.

```
i = -5
if i < 0:
print(i*i)
else:
print(i)
```

25

Writing functions

Writing a function (in a script):

Syntax:

```
def functionname(parametername,…):
(function_body)
```

Example: Function to calculate factorial of an input number n.

```
def factorial(n):
fact = 1
for i in range(1,n+1):
fact = fact*i
```

return(fact);

Running the function (after compiling the script using source

(“script_name”):

>>> factorial(6)

720

Mathematical operations on data types

>>> n = 10

>>> m = 5

>>> n + m #addition

Output: 15

>>> n - m #subtraction

Output: 5

>>> n * m #multiplication

Output: 50

>>> n / m #division

Output: 2.0

Numpy

Numpy is a library for scientific computing. It is useful for working with arrays and matrices. Numpy is used in many scientific computing applications, including machine learning and deep learning. Numpy is imported using the import keyword. Numpy is usually imported using the alias np .

import numpy as np

Numpy arrays

Numpy arrays are used to store multiple items in a single variable. They can be created using the np.array function. Numpy arrays are similar to lists, but they are faster and more efficient. Numpy arrays can be created from lists, tuples, and other arrays.

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

array([1, 2, 3])

Numpy arrays are faster than lists

x = list(range(1000000))

y = np.array(x)

%timeit sum(x)

%timeit np.sum(y)

4.92 ms ± 9.35 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

130 µs ± 75.3 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

Why are numpy arrays faster than lists?

NumPy arrays are faster than lists because they are stored efficiently in memory, allowing for faster data access and manipulation. They also support vectorized operations, which are optimized and implemented in lower-level languages like C, making them much faster than equivalent Python loops. Additionally, NumPy takes advantage of CPU caching, optimized algorithms, and specialized functions, resulting in faster computations compared to pure Python list operations.

Creating arrays in numpy

- An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type
- The easiest way to create an ndarray is to use the array function in numpy module.
- Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array

```
data = np.array([1.5, -0.1, 3])
print(data,"and it's type is",type(data))
```

```
[ 1.5 -0.1  3. ] and it's type is <class 'numpy.ndarray'>
```

```
#difference data types converted to same
data=np.array(['Hello',1])
print(data)
```

```
['Hello' '1']
```

```
data=np.array([[1,2,6.5],['Hello',4.7,'ITER']])
print(data)
```

```
[['1' '2' '6.5']
 ['Hello' '4.7' 'ITER']]
```

Shape and dimension of ndarray

- **arr.ndim**: function return the number of dimensions of an array.
- **arr.shape**: shape of an array is the number of elements in each dimension.
- **arr.size**: Try this and see what you are getting.

```
data=np.array([[1,2,6.5],['Hello',4.7,'ITER']])
print(data)
```

```
[['1' '2' '6.5']
 ['Hello' '4.7' 'ITER']]
```

```
data.ndim
```

```
2
```

```
data.shape
```

```
(2, 3)
```

Other functions for creating new arrays

- numpy.zeros and numpy.ones create arrays of 0s or 1s, respectively, with a given length or shape.
- numpy.empty creates an array without initializing its values to any particular value.
- To create a higher dimensional array with these methods, pass a tuple for the shape.

```
x=np.zeros(10)
x
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
y=np.ones(7)
y
```

```
array([1., 1., 1., 1., 1., 1., 1.])
```

```
np.zeros((3, 6))
```

```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

```
np.empty((2, 3, 2))
```

```
array([[[1.37335591e-311, 2.47032823e-322],
        [0.00000000e+000, 0.00000000e+000],
        [1.16709769e-312, 2.42336543e-057]],

       [[4.75778252e-090, 6.88903061e-042],
        [8.26772458e-072, 8.37820819e+169],
        [3.99910963e+252, 2.17564768e-076]]])
```

```
np.empty(2)
```

```
array([1.37344046e-311, 1.37344046e-311])
```

Creating new arrays

- arange Like the built-in range but returns an ndarray instead of a list
- syntax: numpy.arange(start = 0, stop, step = 1, dtype = None)
- linspace() function is used to create an array of evenly spaced numbers within a specified range

```
import numpy as np
np.arange(5)
```

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

```
#generate an array starting from 10 to 20(exclusive)
#with step size 5
np.arange(10,20,5)
```

```
array([10, 15])
```

```
# generate 5 elements between 10 and 20
np.linspace(10, 20 ,5)
```

```
array([10. , 12.5, 15. , 17.5, 20. ])
```

Creating new arrays

- Produce an array of the given shape and data type with all values set to the indicated ”fill value”
- numpy.full(shape, fill value)
- eye/identity Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)
- Return a new array of given shape and type, filled with fill value.

```
np.full((2, 3), 10)
```

```
array([[10, 10, 10],
       [10, 10, 10]])
```

```
res = np.identity(4)
res
```

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

```
arr2 = np.eye(4)
print(arr2)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

```
arr2 = np.eye(4,k=1)
print(arr2)
```

```
[[0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]
 [0. 0. 0. 0.]]
```

Data types for ndarrays

- The data type or dtype is a special object containing the information about data.
- numpy tries to infer a good data type for the array that it creates.
- You can explicitly convert or cast an array from one data type to another using ndarray’s astype method.

- A string which cannot be converted to float64, if we use astype method, a Value Error will be raised

converting data types for ndarrays

```
: arr1 = np.array([1, 2, 3], dtype=np.float64)
  arr2 = np.array([1, 2, 3], dtype=np.int32)
  print('arr1',arr1,'\n','arr2',arr2)
```

```
arr1 [1. 2. 3.]
 arr2 [1 2 3]
```

```
: print(arr1.dtype)
  print(arr2.dtype)
```

```
float64
int32
```

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

```
[1 2 3 4 5] int32
```

```
: arr = arr.astype(np.float64)
  print(arr,arr.dtype)
```

```
[1. 2. 3. 4. 5.] float64
```

Reshaping Numpy Arrays:

Oftentimes, the shape of a Numpy array needs to be changed without changing its content. One such situation is when we want to convert a 2-dimentional array with 1 row and several columns into a 1-dimensional array. This is required when there are some functions which accepts 1-dimensional array as an argument while the data is available as a 1xN dimensional array.

In the code below, y is a 2-dimensional array, with 1 row and 3 columns, i.e., has the shape 1x3.

```
import numpy as np
y=np.array([[1,2,3]], dtype='float32')
print(y)
print(y.shape)
print(y[0,1])
```

output:
```
[[1. 2. 3.]]
(1, 3)
 2.0
```

The **ravel()** function convert a multidimensional array into a single dimensional array as shown here.

```
y1D=y.ravel()
print(y1D)
print(y1D.shape)
print(y1D[1])
```

output:
```
[1. 2. 3.]
(3,)
2.0
```

A more general function for reshaping an array in the reshape() function.

```
y1Dv2=y.reshape(3)
print(y1Dv2)
print(y1Dv2.shape)
print(y1Dv2[1])

output: [1. 2. 3.]
        (3,)
        2.0
```

Arithmetic With NumPy arrays

- Batch operations on data can be performed in numpy without writing any for loops. This is known as vectorization.
- Any arithmetic operations between equal-size arrays apply the operation element-wise.

```
arr1 = np.array([[1., 2., 3.], [4., 5., 6.]]);arr1
```

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

```
arr2 = np.array([[1., 1., 1.], [2., 2., 2.]]);arr2
```

```
array([[1., 1., 1.],
       [2., 2., 2.]])
```

```
arr1+arr2
```

```
array([[2., 3., 4.],
       [6., 7., 8.]])
```

```
arr1*arr2
```

```
array([[ 1.,  2.,  3.],
       [ 8., 10., 12.]])
```

Arithmetic operations

- Arithmetic operations with scalars propagate the scalar argument to each element in the array
- Comparisons between arrays of the same size yield Boolean arrays.

```
7*arr2
```

```
array([[ 7.,  7.,  7.],
       [14., 14., 14.]])
```

```
arr2**2
```

```
array([[1., 1., 1.],
       [4., 4., 4.]])
```

```
7/arr2
```

```
array([[7. , 7. , 7. ],
       [3.5, 3.5, 3.5]])
```

```
arr1
```

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

```
arr2
```

```
array([[1., 1., 1.],
       [2., 2., 2.]])
```

```
arr1>arr2
```

```
array([[False,  True,  True],
       [ True,  True,  True]])
```