Python Programming

Objectives

At the end of this module, you should be able to:

- Start writing basic codes in Python
- work with Numbers, variables & basic operations in python
- Work with Data Types – Lists, tuples & dictionary in python
- Create Rule based systems using if else, for and while loop
- Write user defined functions
- Writing codes using OOPs
- Numpy
- Pandas
- Matplotlib
- Seaborn

Python

Python is one of the mostly used programming language in the world.

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language.

Guido van Rossum created this during 1985- 1990.

Python is an opensource programming language and its source code is available under the GNU General Public License (GPL).

Some key features of python –

- Python is Interpreted: Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- Python is Interactive: You can sit at a Python prompt and interact with the interpreter directly to write your programs.
- Python is Object-Oriented: Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

Getting Started –

Type the following text at the Python prompt and press Enter –

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

If you are running the older version of Python (Python 2.x), use of parenthesis as inprint function is optional. This produces the following result –

```
Hello, World!
```

Declaring Variables

Variables are nothing but reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory.

```
count = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "Aashish Pandey"       # A string

print (count)
print (miles)
print (name)
```

Here, 100, 1000.0 and "John" are the values assigned to counter, miles, and name variables, respectively. This produces the following result –

```
100
1000.0
Aashish Pandey
```

Data Types in Python

Python has five standard data types –

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

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

```
var1 = 1
var2 = 10
```

Python supports three different numerical types –

- int (signed integers)
- float (floating point real values)
- complex (complex numbers)

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows either pair of single or double quotes.

```
str = 'Hello World!'

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 string two times
print (str + "TEST") # Prints concatenated string
```

This will produce the following result –

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One of the differences between them is that all the items belonging to a list can be of different data type.

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']


print (list)          # Prints complete list
print (list[0])       # Prints first element of the list
print (list[1:3])     # Prints elements starting from 2nd till 3rd
print (list[2:])      # Prints elements starting from 3rd element
print (tinylist * 2)  # Prints list two times
print (list + tinylist) # Prints concatenated lists
```

This produces the following result −

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Python 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 parenthesis.

The main difference 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. For example –

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print (tuple)           # Prints complete tuple
print (tuple[0])        # Prints first element of the tuple
print (tuple[1:3])      # Prints elements starting from 2nd till 3rd
print (tuple[2:])       # Prints elements starting from 3rd element
print (tinytuple * 2)   # Prints tuple two times
print (tuple + tinytuple) # Prints concatenated tuple
```

This produces the following result −

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

Python Dictionary

Python's dictionaries are kind of hash-table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example –

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}



print (dict['one'])       # Prints value for 'one' key
print (dict[2])           # Prints value for 2 key
print (tinydict)          # Prints complete dictionary
print (tinydict.keys())   # Prints all the keys
print (tinydict.values()) # Prints all the values
```

This produces the following result –

```
This is one
This is two
{'name': 'john', 'dept': 'sales', 'code': 6734}
dict_keys(['name', 'dept', 'code'])
dict_values(['john', 'sales', 6734])
```

Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type-names as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

S.No.	Function & Description
1	int(x [,base]) Converts x to an integer. The base specifies the base if x is a string.
2	float(x) Converts x to a floating-point number.
3	complex(real [,imag]) Creates a complex number.
4	str(x) Converts object x to a string representation.
5	repr(x) Converts object x to an expression string.
6	eval(str) Evaluates a string and returns an object.
7	tuple(s) Converts s to a tuple.

8	list(s) Converts s to a list.
9	set(s) Converts s to a set.
10	dict(d) Creates a dictionary. d must be a sequence of (key,value) tuples.
11	frozenset(s) Converts s to a frozen set.
12	chr(x) Converts an integer to a character.
13	unichr(x) Converts an integer to a Unicode character.
14	ord(x) Converts a single character to its integer value.
15	hex(x) Converts an integer to a hexadecimal string.
16	oct(x) Converts an integer to an octal string.

Python Arithmetic Operators

Assume variable a holds the value 10 and variable b holds the value 21, then

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
*** Multiplication**	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
**** Exponent**	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Operators in Python

Python Comparison Operators

These operators compare the values on either side of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds the value 10 and variable b holds the value 20, then

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!= b) is true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Assignment Operators

Assume variable a holds the value 10 and variable b holds the value 20, then

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c – a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constants or variables or any other identifier names. All the Python keywords contain lowercase letters only.

and	**exec**	**Not**
as	finally	Or
assert	for	Pass
break	from	Print
class	global	Raise
continue	if	Return
def	import	Try
del	in	While
elif	is	With
else	lambda	Yield
except		

Decision Making – IF Loop

The IF statement is similar to that of other languages. The if statement contains a logical expression using which the data is compared and a decision is made based on the result of the comparison.

```
var1 = 100
if var1:
   print ("1 - Got a true expression value")
   print (var1)

var2 = 0
if var2:
   print ("2 - Got a true expression value")
   print (var2)
print ("Good bye!")
```

Output

When the above code is executed, it produces the following result –

```
1 - Got a true expression value
100
Good bye!
```

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested ifconstruct.

In a nested if construct, you can have an if...elif...else construct inside another if...elif...else construct.

Example

```
# !/usr/bin/python3
num = int(input("enter number"))
if num%2 == 0:
   if num%3 == 0:
      print ("Divisible by 3 and 2")
   else:
      print ("divisible by 2 not divisible by 3")
```

```
else:
   if num%3 == 0:
      print ("divisible by 3 not divisible by 2")
   else:
      print  ("not Divisible by 2 not divisible by 3")
```

Output

When the above code is executed, it produces the following result –

```
enter number8
divisible by 2 not divisible by 3

enter number15
divisible by 3 not divisible by 2

enter number12
Divisible by 3 and 2

enter number5
not Divisible by 2 not divisible by 3
```

For Loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Example

range() generates an iterator to progress integers starting with 0 up to n-1. To obtain a list object of the sequence, it is type casted to list(). Now this list can be iterated using thefor statement.

```
>>> for var in list(range(5)):
   print (var)
```

Output

This will produce the following output.

```
0
1
2
3
4
```

While Loop

Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

Example

```
#!/usr/bin/python3


count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1


print ("Good bye!")
```

Output

When the above code is executed, it produces the following result −

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

Reading Input from Keyboard

Python 2 has two versions of input functions, input() and raw_input(). The input() function treats the received data as string if it is included in quotes '' or "", otherwise the data is treated as number.

In Python 3, raw_input() function is deprecated. Further, the received data is always treated as string.

```
>>> x = input("something:")
something:10
>>> x
'10'

>>> x = input("something:")
something:'10' #entered data treated as string with or without ''
>>> x
"'10'"

>>> x = raw_input("something:") # will result NameError
Traceback (most recent call last):
   File "<pyshell#3>", line 1, in
  <module>
   x = raw_input("something:")
NameError: name 'raw_input' is not defined
```

Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword def followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return

# Now you can call printme function
printme("This is first call to the user defined function!")
printme("Again second call to the same function")
```

When the above code is executed, it produces the following result −

```
This is first call to the user defined function!
Again second call to the same function
```

File I/O Handling

Writing data to the file

```
#!/usr/bin/python3


# Open a file
fo = open("foo.txt", "w")
fo.write( "Python is a great language.\nYeah its great!!\n")


# Close opend file
fo.close()
```

The above method would create foo.txt file and would write given content in that file and finally it would close that file. If you would open this file, it would have the following content –

```
Python is a great language.
Yeah its great!!
```

Reading Data from a file

```
# Open a file
fo = open("foo.txt", "r+")
str = fo.read(10)
print ("Read String is : ", str)


# Close opened file
fo.close()
```

This produces the following result –

```
Read String is :  Python is
```

Object Oriented Programming

- Python has been an object-oriented language since the time it existed. Due to this, creating and using classes and objects are downright easy.

Example

Following is an example of a simple Python class −

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary
```

- The variable *empCount* is a class variable whose value is shared among all the instances of a in this class. This can be accessed as *Employee.empCount* from inside the class or outside the class.

- The first method *__init__()* is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.

- You declare other class methods like normal functions with the exception that the first argument to each method is *self*. Python adds the *self* argument to the list for you; you do not need to include it when you call the methods.

Numpy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

Broadcasting two arrays together follows these rules:

- If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
- The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
- The arrays can be broadcast together if they are compatible in all dimensions.
- After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
- In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code.

For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

Pandas

pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.

Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language. It is already well on its way toward this goal.

pandas is well suited for many different kinds of data:

- Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
- Ordered and unordered (not necessarily fixed-frequency) time series data.
- Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
- Any other form of observational / statistical data sets. The data actually need not be labeled at all to be placed into a pandas data structure

The two primary data structures of pandas, Series (1-dimensional) and DataFrame (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering. For R users, DataFrame provides everything that R’s data.frame provides and much more. pandas is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

Here are just a few of the things that pandas does well:

- Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data
- Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects
- Automatic and explicit data alignment: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let Series, DataFrame, etc. automatically align the data for you in computations
- Powerful, flexible group by functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data

- Make it easy to convert ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects
- Intelligent label-based slicing, fancy indexing, and subsetting of large data sets
- Intuitive merging and joining data sets
- Flexible reshaping and pivoting of data sets
- Hierarchical labeling of axes (possible to have multiple labels per tick)
- Robust IO tools for loading data from flat files (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast HDF5 format
- Time series-specific functionality: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging, etc.

Many of these principles are here to address the shortcomings frequently experienced using other languages / scientific research environments. For data scientists, working with data is typically divided into multiple stages: munging and cleaning data, analyzing / modeling it, then organizing the results of the analysis into a form suitable for plotting or tabular display. pandas is the ideal tool for all of these tasks.

- pandas is fast. Many of the low-level algorithmic bits have been extensively tweaked in Cython code. However, as with anything else generalization usually sacrifices performance. So if you focus on one feature for your application you may be able to create a faster specialized tool.
- pandas is a dependency of statsmodels, making it an important part of the statistical computing ecosystem in Python.
- pandas has been used extensively in production in financial applications.

Seaborn

Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphics.

Seaborn is a library for making attractive and informative statistical graphics in Python. It is built on top of matplotlib and tightly integrated with the PyData stack, including support for numpy and pandas data structures and statistical routines from scipy and statsmodels.

Some of the features that seaborn offers are

- Several built-in themes for styling matplotlib graphics
- Tools for choosing color palettes to make beautiful plots that reveal patterns in your data
- Functions for visualizing univariate and bivariate distributions or for comparing them between subsets of data
- Tools that fit and visualize linear regression models for different kinds of independent and dependent variables
- Functions that visualize matrices of data and use clustering algorithms to discover structure in those matrices
- A function to plot statistical timeseries data with flexible estimation and representation of uncertainty around the estimate
- High-level abstractions for structuring grids of plots that let you easily build complex visualizations

Seaborn aims to make visualization a central part of exploring and understanding data. The plotting functions operate on dataframes and arrays containing a whole dataset and internally perform the necessary aggregation and statistical model-fitting to produce informative plots. If matplotlib “tries to make easy things easy and hard things possible”, seaborn tries to make a well-defined set of hard things easy too.

The plotting functions try to do something useful when called with a minimal set of arguments, and they expose a number of customizable options through additional parameters. Some of the functions plot directly into a matplotlib axes object, while others operate on an entire figure and produce plots with several panels. In the latter case, the plot is drawn using a Grid object that links the structure of the figure to the structure of the dataset.

Seaborn should be thought of as a complement to matplotlib, not a replacement for it. When using seaborn, it is likely that you will often invoke matplotlib functions directly to draw simpler plots already available through the pyplot namespace. Further, the seaborn functions aim to make plots that are reasonably “production ready” (including extracting semantic information from Pandas objects to add informative labels), but full customization will require changing attributes on the matplotlib objects directly. The combination of seaborn’s high-level interface and matplotlib’s customizability and wide range of backends makes it easy to generate publication-quality figures.

Summary

- Atomic Data Types – Integers, Strings, Float, Boolean and Complex
- Operators in Python – Arithmetic, Comparison and Logical
- Main Data Types – List, Tuple, Set and Dictionary
- Control Structure – IF else, For loop and While Loop
- Defining Functions in Python
- Object Oriented Programming using Python
- Numpy
- Matplotlib
- Pandas
- seaborn