

# Fundamentals of Python

## Table of Contents

### **Chapter - 1 : Introduction**

### **Chapter - 2 : Python Fundamentals**

1. Operators in Python
2. Variable and Data Type in Python
3. Data Structures in Python
4. Itertools

### **Chapter - 3 : Python Programming Constructs**

1. Conditional Statements [Selection]
2. Iterative Statements [Repetition]

### **Chapter - 4 : Functions**

### **Chapter - 5 : Classes and Objects**

# Introduction

Python is a computer programming language created in 1989 by a person named Guido Van Rousom. After its creation there have been a few important releases such as Python 1.0 in 1994, Python 2.0 in 2000, and Python 3.0 in 2008. In the HighRadius System we use the 3.6 and 3.7 versions.

Python is a Multi-Purpose programming language. It is used for developing GUI (Graphical User Interfaces), various scripting purposes, creating backend applications, web scraping and various other things. It is an Interpreted Language, that is, it is executed in a sequential manner and does not need to be compiled before it is executed. It is a strongly and dynamically typed programming language which is extendable and portable. It can be used to combine various programming languages together to work cohesively as one distinct entity. In addition to that, Python is also a free and open source programming language which means that it is free to use and everyone can contribute to its development.

The reason why Python is such a popular programming language is because it is a high level programming language which is simple to learn. It has the support of various packages such as Pandas, Numpy, Scikit-Learn and many more. They can be harnessed to many tasks such as numerical manipulations, visualization tools and even various machine learning algorithms. It is widely used in various industries for its powerful and multifaceted features.

## Python Fundamentals

The notebook that contains examples of all the fundamentals of python has been included in this [link](#). Make a copy of this in your drive and explore as much as you want.

Python is a very simple coding language that uses a very familiar language to code. It uses indentation to define blocks of code and they need to be consistent throughout the block.

```
[1] print("Hello World")  
    print(1+2)
```

```
Hello World  
3
```

```
[2] print("Additon Example")  
    a = 10  
    b = 30  
    print(a+b)
```

```
Additon Example  
40
```

The above example depicts the simplicity of python as a coding language. Indentation is very important in python and not following proper indentation structure causes an error.

```
[3] print("Additon Example")
    a = 10
    b = 30
    print(a+b)

File "<ipython-input-3-1e6fca0a7e8e>", line 2
    a = 10
    ^
IndentationError: unexpected indent
```

Semicolons have almost no use in python but using them would not throw any error. It is not considered good practice while writing python code. It can be used to separate many commands in a single line.

## Operators in Python

There are many operators in python that can be used for many purposes. They are stated below.

Operator	Description	Example	Operator	Description	Example
+	Addition	2 + 4 == 6	,	Comma	range(0, 10)
-	Subtraction	2 - 4 == -2	:	Colon	def X():
*	Multiplication	2 * 4 == 8	.	Dot	self.x = 10
**	Power of	2 ** 4 == 16	=	Assign equal	x = 10
/	Division	2 / 4.0 == 0.5	;	semi-colon	Print("hi"); print("there")
//	Floor division	2 // 4.0 == 0.0	+=	Add and assign	x = 1; x += 2
%	String interpolate or modulus	2 % 4 == 2	-=	Subtract and assign	x = 1; x -= 2
<	Less than	4 < 4 == False	*=	Multiply and assign	x = 1; x *= 2
>	Greater than	4 > 4 == False	/=	Divide and assign	x = 1; x /= 2
<=	Less than equal	4 <= 4 == True	//=	Floor divide and assign	x = 1; x //= 2
>=	Greater than equal	4 >= 4 == True	%=	Modulus assign	x = 1; x %= 2
==	Equal	4 == 5 == False	**=	Power assign	x = 1; x **= 2
!=	Not equal	4 != 5 == True		Boolean Or,	
<>	Not equal	4 <> 5 == True		Boolean And,	
()	Parenthesis	len('hi') == 2		Boolean Not	(a or b) and c
[]	List brackets	[1,3,4]			
{}	Dict curly braces	{'x': 5, 'y': 10}	or, and, not		

## Variables and Data Types in Python

In Python, variables are considered as storage placeholders for texts and numbers.

Python is dynamically typed, such that there is no need to declare what the type of each variable is when it is declared or initialized [type() method is used to find the data type]

<code>x = 123</code>	<code># integer</code>
<code>x = 123L</code>	<code># long integer</code>
<code>x = 3.14</code>	<code># double float</code>
<code>x = "hello"</code>	<code># string</code>
<code>x = [0,1,2]</code>	<code># list</code>
<code>x = (0,1,2)</code>	<code># tuple</code>
<code>x = open('hello.py', 'r')</code>	<code># file</code>

Although you don't need to define the type of a variable, python is strongly typed in the sense that operations can not be performed between two dissimilar data types.

```
[4] a = [1,2]
    a+"hi"
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-4-a238315bcc9f> in <module>()
      1 a = [1,2]
----> 2 a+"hi"

TypeError: can only concatenate list (not "str") to list
```

## Python Programming Constructs

Constructs control the flow of the program. If we dive deep into the types of constructs, they are primarily of three types : Sequence, Selection and Repetition.

A **Sequence** is an order in which the code will get executed. **Selection** is the part where it is decided which block of code will get executed based on some conditions. **Repetition** is the construct that decides which part of the code will get executed multiple times based on specific criteria.

### Conditional Statements [Selection]

Branching in Python can be achieved through the following keywords: if, elif (else-if) and else. The scope of the statement block is decided through indentation (cascading in case of nested conditions).

An example of the construct can be seen in the following figure,

```

if condition:
    statement
    statement
    # ... some more indented statements if
    necessary
elif <Condition>:
    statetement
else:
    statement

```

#### Ternary

```
max = a if (a > b) else b
```

An example of the construct in use can be found in the below code snippet,

#### ▼ If-else

```

[ ] a = 33
    b = 200                                #be mindful of indent
    if b > a:
        print("b is greater than a")

```

b is greater than a

```

[ ] #elif keyword
    if b > a:
        print("b is greater than a")
    elif a == b:
        print("a and b are equal")

```

b is greater than a

```

[ ] #else keyword
    if b > a:
        print("b is greater than a")
    elif a == b:
        print("a and b are equal")
    else:
        print("a is greater than b")

```

b is greater than a

Here the score is compared and according to specific conditions (>90,>60 and <=90) different sets of code blocks are executed.

## Iterative Statements [Repetition]

Iterative constructs in python are achieved through loops. They are primarily of two types: **for loop** and **while loop**.

## Iterations and Looping

```
[ ] #for loop
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
#for loop does not require indexing
```

apple  
banana  
cherry

```
[50] #while loop
i = 1
while i < 6:
    print(i)
    i += 1
```

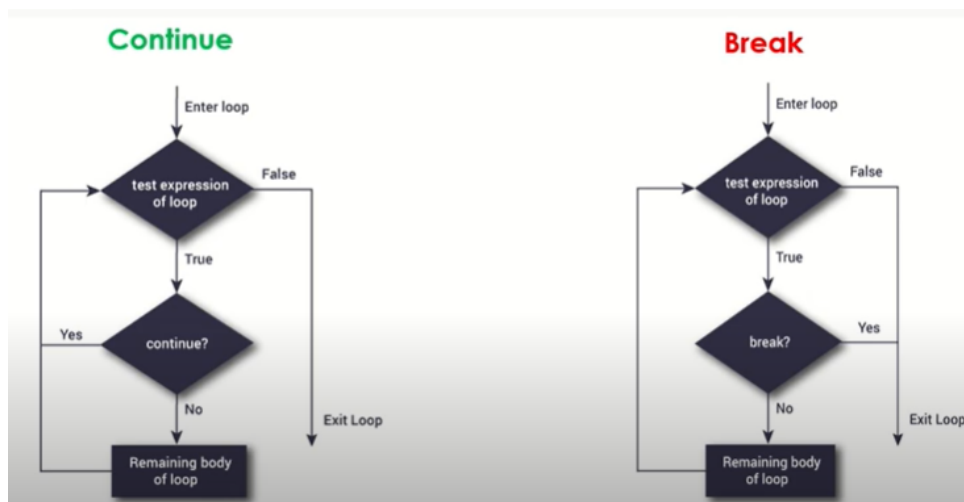
1  
2  
3  
4  
5

The conditions in which the loops will continue to execute or stop after a specific number of iterations are controlled through two keywords, i.e., **continue** and **break**.

**Continue** statement is used to tell python to skip the rest of the statements in a current loop construct and continue with the next iteration of the code block.

**Break**, on the other hand, is used to completely break out of the loop.

The following figure shows the use of **break** and **continue** in separate programming constructs as they are used in python.



Use of break and continue in python:

```
[ ] #break statement
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
```

```
apple
banana
```

```
[ ] #continue statement
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

```
apple
cherry
```

## Data Structures in Python

There are many ways to store data in python. They are in the form of various data structures. For example, lists, tuples, dictionaries, sets, and many more.

### 1. List

List is one of the simplest and most important data structures in python. They are defined by enclosing square brackets "[ ]" and each item is separated by a ",". Lists can be defined as a collection of items where each item has an assigned positional value (index value) starting from 0 (zero). It is mutable, i.e., its contents can be changed. It is similar to an array with some basic differences. For example, lists can store heterogeneous data types together under one name unlike matrices(arrays) that contain homogeneous data.

There are many **methods** that can be used to manipulate lists and do various operations. They are listed in the image below with their corresponding uses.

Append()	Add an element to the end of the list
Extend()	Add all elements of a list to the another list
Insert()	Insert an item at the defined index
Remove()	Removes an item from the list
Pop()	Removes and returns an element at the given index
Clear()	Removes all items from the list
Index()	Returns the index of the first matched item
Count()	Returns the count of number of items passed as an argument
Sort()	Sort items in a list in ascending order
Reverse()	Reverse the order of items in the list
copy()	Returns a copy of the list

There are many **inbuilt functions** that are applicable for a list. They are as follows :

round()	Rounds off to the given number of digits and returns the floating point number
sum()	Sums up the numbers in the list
cmp()	This function returns 1, if first list is "greater" than second list
max()	return maximum element of given list
min()	return minimum element of given list
len()	Returns length of the list or size of the list
filter()	tests if each element of a list true or not
map()	returns a list of the results after applying the given function to each item of a given iterable
lambda()	This function can have any number of arguments but only one expression, which is evaluated and returned.

A few examples of these methods being used are :

```
[5] x = [1,2,3]
    x.append(4)
    sum(x)
```

10

```
[6] def add(x):
    return x*2
    y = list(map(add,x))
    print(y)
```

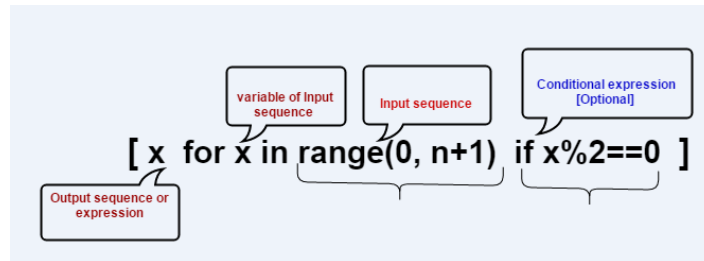
[2, 4, 6, 8]

In the above example, a list "x" is being initialized with values [1,2,3]. The append function is used to add "4" to the list and then sum(x) gives the total of the entire



list. In addition to that, there is a method defined to calculate the square of any number and it is being applied to the entire list using the map function.

List Comprehension is the process of creating new lists from pre-existing lists.



## 2. Tuple

A Tuple can be defined as an immutable list. It can not be altered. It is defined by initializing elements in between parentheses "`( )`". Once a tuple has been created, you can not add or alter elements in the tuple. It has only two methods: `count()` and `index()`. Count gives the frequency of a searched element while index provides the location of the searched element in the tuple (index starts with 0).

The benefit of using tuples is that they are faster than lists. It protects the data against any accidental changes to sensitive data if the user knows that the data will not need to be changed in the future. The main advantage of tuples is that it can be used as key values in dictionaries while lists can not.

Note that, tuples are immutable,i.e., once created, its elements cannot be changed

```
[ ] #access tuple items
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

banana

```
[8] thistuple[2] = "orange"
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-8-5410d42e4faf> in <module>()
----> 1 thistuple[2] = "orange"

TypeError: 'tuple' object does not support item assignment
```

## 3. Sets

A set contains an unordered collection of unique and immutable objects. All kinds of operations that are applicable to a set can be used for sets.

## ▼ Set Operations

```
▶ #access items; cannot access items by referring to an index
#example
thisset = {"apple", "banana", "cherry"}
for x in thisset:
    print(x)

banana
cherry
apple
```

Sets are immutable. Once created, we cannot change its contents.

```
✓ [35] #adding items
os    thisset.add("orange")      #adding one item at a time
      thisset.update(["orange", "mango", "grapes"])    #adding more than one item at a time.
```

```
✓ [36] #removing items
os    thisset.remove("banana")
      thisset.discard("banana")
      x = thisset.pop()      #pop will remove only the last added element
      thisset.clear()      #empties the set
      del thisset          #delete the set completely
```

```
[38] #join two sets
      set1 = {"a", "b", "c"}
      set2 = {1, 2, 3}
      set3 = set1.union(set2)
      print(set3)

{1, 2, 3, 'c', 'a', 'b'}
```

## 4. Dictionary

It is a python data structure that is used to store data in key-value pairs. They are a set of attributes that have corresponding values. It is an unordered, indexed, and changeable form of data that is written within curly braces.

One can perform various actions on a dictionary. For instance, printing all the keys of the dictionary, all the values in the dictionary, adding a new element into the dictionary and also removing a key-value pair from the dictionary.

### ▼ Dictionary

```
✓ [39] employee = {"e-id":1221,
os              "e-name":"Robert",
              "dob":"01-01-1990"
              }
      print(employee)

{'e-id': 1221, 'e-name': 'Robert', 'dob': '01-01-1990'}
```

## 5. Strings

Strings can be defined as a list or an ordered chain of characters. We can perform various operations or manipulations on these strings.

### ▼ Strings

```
✓ [49] word = "Hello-World"
      print(word.split("-"))
      print(word.replace("Hello", "Hi"))
      print(word[::-1])
      print(word.isalnum())

['Hello', 'World']
Hi-World
dlrow-olleH
False
```

In the above example, the `isalnum()` function produces a result as “False” as the string word contains a “-”.

## Itertools

Python’s `Itertool` is a module that provides various functions that work on iterators to produce complex iterators. This module works as a fast, memory-efficient tool that is used either by itself or in combination to form complex algebraic equations.

### ▼ Itertools

```
✓ [103] import itertools

      # for in loop
      for i in itertools.count(5, 5):
          if i == 35:
              break
          else:
              print(i, end = " ")

5 10 15 20 25 30
```

## Slicing Function

The Python `slice()` function allows us to slice a sequence. It means we can retrieve a part of a string, tuple, list, etc. We can specify the start, end, and step of the slice. The step lets you skip items in the sequence.

The **Syntax** of slice() is:

**slice(start, stop, step)**

### **slice() Parameters:**

slice() can take three parameters:

- **start** (optional) - Starting integer where the slicing of the object starts. Default to None if not provided.
- **stop** - Integer until which the slicing takes place. The slicing stops at index stop -1 (last element).
- **step** (optional) - Integer value which determines the increment between each index for slicing. Defaults to None if not provided.

**Return Type:** Returns a sliced object containing elements in the given range only.

Slicing a string:

```
# String Slicing
String = 'NewSlice'
s1 = slice(3)
s2 = slice(1, 5, 2)

print("String slicing")
print(String[s1])
print(String[s2])
```

```
String slicing
New
eS
```

---

Slicing a List:

```
# List Slicing
L = [1, 2, 3, 4, 5]
s1 = slice(3)
s2 = slice(1, 5, 2)
print("List slicing")
print(L[s1])
print(L[s2])
```

```
List slicing
[1, 2, 3]
[2, 4]
```

Slicing a tuple:

```
# Tuple Slicing
T = (1, 2, 3, 4, 5)
s1 = slice(3)
s2 = slice(1, 5, 2)
print("\nTuple slicing")
print(T[s1])
print(T[s2])
```

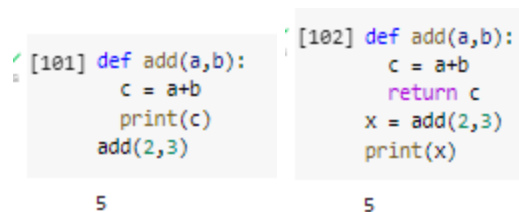
```
Tuple slicing
(1, 2, 3)
(2, 4)
```

# Functions

A function is a construct that is defined by the keyword “def”. The general syntax looks like this:

```
def function_name(Parameter List):  
    #Statements, i.e, the function body  
    return statement (if required)
```

An example of a function used to add two numbers is given below,



```
[101] def add(a,b):  
      c = a+b  
      print(c)  
      add(2,3)  
5
```

```
[102] def add(a,b):  
      c = a+b  
      return c  
      x = add(2,3)  
      print(x)  
5
```

## Lambda Function

We use lambda functions when we require a nameless function for a short period of time. In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

### With filter():

The filter() function in Python takes in a function and a list as arguments. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

```
# with filter()  
my_list = [1, 5, 4, 6, 8, 11, 3, 12]  
  
new_list = list(filter(lambda x: (x%2 == 0) , my_list))  
  
print(new_list)  
  
[4, 6, 8, 12]
```

### With map():

The map() function in Python takes in a function and a list. The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

```
# with map()
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2, my_list))

print(new_list)

[2, 10, 8, 12, 16, 22, 6, 24]
```

## Classes and Objects

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new objects of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by their class) for modifying or manipulating their state.

An object consists of :

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

In the below example, the **Car** is a class and **BMW** is an object of that class. The class Car has various attributes such as attr1 that contains the engine type and attr2 that contains the HorsePower of the car. We can use these attributes and the **start** function as they are used to manipulate the state of the object. The **self** keyword is used to refer to the object that is calling a particular class method.

```
[118] # Python3 program to
# demonstrate instantiating
# a class
class Car:
    # A simple class
    # attribute
    attr1 = "Petrol"
    attr2 = "750 HP"
    # A sample method
    def start(self):
        print("Engine Started : Engine Type ", self.attr1)
        print("Ready to GO : Horse Power ", self.attr2)
# Driver code
# Object instantiation
BMW = Car()
# Accessing class attributes
# and method through objects
print(BMW.attr1)
BMW.start()

Petrol
Engine Started : Engine Type  Petrol
Ready to GO : Horse Power  750 HP
```

## **\_\_init\_\_ method**

It is used to initialize the attributes for a class with specific values for a particular object. It is executed at the time of object creation for a particular class. An example of the use of the \_\_init\_\_ function can be seen below,

```
[119] class Person:
    # init method or constructor
    def __init__(self, name):
        self.name = name
    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)
p = Person('Robert')
p.say_hi()

Hello, my name is Robert
```