Python For Data Science

mukhtarmid@gmail.com

Python is an open-source, interpreted, high-level language and provides a great approach for object-oriented programming. It is one of the best languages used by data scientists for various data science projects/applications. Python offers great functionality to deal with mathematics, statistics, and scientific function. It provides great libraries to deal with data science applications.

One of the main reasons why Python is widely used in the scientific and research communities is because of its ease of use and simple syntax which makes it easy to adapt for people who do not have an engineering background. It is also more suited for quick prototyping.

According to engineers from academia and industry, deep learning frameworks available with Python APIs and scientific packages have made Python incredibly productive and versatile. There has been a lot of evolution in deep learning Python frameworks and it's rapidly upgrading.

In terms of application areas, ML scientists prefer Python as well. When it comes to areas like building fraud detection algorithms and network security, developers leaned towards Java, while for applications like natural language processing (NLP) and sentiment analysis, developers opted for Python, because it provides a large collection of libraries that help to solve the complex business problem quickly, build strong system and data application.

- Below are some facts about Python Programming Language:
    - Python is currently the most widely used multi-purpose, high-level programming language.
    - Python allows programming in Object-Oriented and Procedural paradigms.

- Python programs generally are smaller than other programming languages like Java. Programmers have to type relatively less and the indentation requirement of the language makes them readable all the time.
- It uses elegant syntax, hence the programs are easier to read.
- It is a simple to access language, which makes it easy to achieve the program working.
- The large standard library and community support.
- In Python, it is also simple to extend the code by appending new modules that are implemented in other compiled languages like C++ or C.
- Python is an expressive language that is possible to embed into applications to offer a programmable interface.
- Allows developers to run the code anywhere, including Windows, Mac OS X, UNIX, and Linux.
- It is free software in a couple of categories. It does not cost anything to use or download Pythons or to add it to the application.

Variables Assignment

Think of a variable as a name attached to a particular object. In Python, variables need not be declared or defined in advance, as is the case in many other programming languages. To create a variable, you just assign it a value and then start using it. The assignment is done with a single equals sign (=)

Rules for creating variables in Python:

- A variable name must start with a letter or the underscore character.
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
- Variable names are case-sensitive (name, Name, and NAME are three different variables).
- The reserved words(keywords) cannot be used to name the variable.

A Simple Program to Demonstrate Python Variable Assignment

Program:

```
# An integer assignment
age = 45
# A floating point
salary = 1456.8
# A string
name = "John"
print(age)
print(salary)
print(name)
```

Output:

```
45
1456.8
John
```

Data Types

In programming, the data type is an important concept.
Variables can store data of different types, and different types can do different things.
Python has the following data types built-in by default, in these categories:

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

- Numbers
    - Integers
    - Floating-Point Numbers
    - Complex Numbers
- Strings
- Boolean
- List
- Tuple
- Range
- Dictionary
- Set

Numbers

Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data type. Python provides the `type()` function to know the data type of the variable. Similarly, the `isinstance()` function is used to check whether an object belongs to a particular class.

1. **Int** (`int`)- Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to int
2. **Float** (`float`)- Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate up to 15 decimal points.
3. **complex** (`complex`)- A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

A Simple Program to Demonstrate Python Numeric Data Types

Python creates Number objects when a number is assigned to a variable. For example;

Program:

```
a = 5
print("The type of a", type(a))
b = 40.5
print("The type of b", type(b))
c = 1+3j
print("The type of c", type(c))
print(" c is a complex number", isinstance(1+3j,complex))
```

Output:

```
The type of a <class 'int'>
```

```
The type of b <class 'float'>

The type of c <class 'complex'>

c is complex number: True
```

Strings (`str`)

The string can be defined as the sequence of characters represented in quotation marks. In Python, we can use single, double, or triple quotes to define a string. String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string. In the case of string handling, the operator + is used to concatenate two strings as the operation `"hello"+" python"` returns `"hello python"`. The operator `*` is known as a repetition operator as the operation `"Python" *2` returns `'Python Python'`.

A Simple Program to Demonstrate Python String Data Types

Program:

```
str = "string using double quotes"
print(str)
s = '''A multiline
string'''
print(s)
```

Output:

```
string using double quotes

A multiline

string
```

String Methods

Method	Description	Examples

capitalize()	Returns a copy of the string with its first character capitalized and the rest lowercased.	`>>>mystring = "hello python"`
		`>>>print(mystring.capitalize())`
		`Output: Hello python`

Casefold()	Returns a casefolded copy of the string. Casefolded strings may be used for caseless matching.	`>>> mystring = "hello PYTHON"`
		`>>>print(mystring.casefold())`
		`Output: hello python`

Center(width, [fillchar])	Returns the string centered in a string of length *width*. Padding can be done using the specified *fillchar* (the default padding uses an ASCII space). The original string is returned if *width* is less than or equal to len(s)	`>>>mystring = "Hello"`
		`>>>x = mystring.center(12, "-")`
		`>>>print(x)`
		`Output: ---Hello----`

Count(sub, [start], [end])	Returns the number of non-overlapping occurrences of substring (*sub*) in the range [*start*, *end*]. Optional arguments *start*and *end* are interpreted	`>>>mystr = "Hello Python"`
		`>>>print(mystr.count("o"))`
		`Output: 2`

	as in slice notation.	`>>>print(mystr.count("th"))`
		`Output: 1`
		`>>>print(mystr.count("l"))`
		`Output: 2`
		`>>>print(mystr.count("h"))`
		`Output: 1`
		`>>>print(mystr.count("H"))`
		`Output: 1`
		`>>>print(mystr.count("hH"))`
		`Output: 0`

Encode(encoding = “utf-g”, errors = “strict”)	Returns an encoded version of the string as a bytes object. The default encoding is utf-8. errors may be given to set a different error handling scheme. The possible value for errors are:
	• strict (encoding errors raise a UnicodeError)
	• ignore
	• replace
		`>>>mystr = 'python!'`
		`>>>print('The string is:', mystr)`
		`Output: The string is: python!`
		`>>>print('The encoded version is: ', mystr.encode("ascii", "ignore"))`
		`Output: The encoded version is: b'python!'`

	• xmlcharrefreplace • backslashreplace • any other name registered via codecs.register_error()	`>>>print('The encoded version (with replace) is:', mystr.encode("ascii", "replace"))` `Output: The encoded version (with replace) is: b'python!'`

endswith(suffix, [start], [end])	Returns True if the string ends with the specified suffix, otherwise it returns False.	`>>>mystr = "Python"`
		`>>>print(mystr.endswith("y"))`
		`Output: False`
		`>>>print(mystr.endswith("hon"))`
		`Output: True`

Expandtabs(tabsize=8)	Returns a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size.	`>>>mystr = "1\t2\t3"`
		`>>>print(mystr)`
		`Output: 1 2 3`
		`>>>print(mystr.expandtabs())`
		`Output: 1 2 3`
		`>>>print(mystr.expandtabs(tabsize=15))`
		`Output: 1 2 3`

		`>>>print(mystr.expandtabs(tabsize=2))` `Output: 1 2 3`

Find(sub, [start], [end])	Returns the lowest index in the string where substring *sub* is found within the slice s[start:end].	`>>>mystring = "Python"`
		`>>>print(mystring.find("P"))`
		`Output: 0`
		`>>>print(mystring.find("on"))`
		`Output: 4`

Format(*args, **kwargs)	Performs a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces {}.	`>>>print("{} and {}".format("Apple", "Banana"))`
		`Output: Apple and Banana`
		`>>>print("{1} and {0}".format("Apple", "Banana"))`
		`Output: Banana and Apple`
		`>>>print("{lunch} and{dinner}".format(lunch="Peas", dinner="Beans"))`
		`Output: Peas and Beans`

format_map(mapping)	Similar to format(**mapping), except that mapping is used directly and not copied to a dictionary.	`>>>lunch = {"Food": "Pizza", "Drink": "Wine"}`
		`>>>print("Lunch: {Food},{Drink}".format_map(lunch))`
		`Output: Lunch: Pizza, Wine`
		`>>>class Default(dict):`
		`def __missing__(self, key):`
		`return key`
		`>>>lunch = {"Drink": "Wine"}`
		`>>>print("Lunch:{Food},{Drink}".format_map(Default(lunch)))`
		`Output: Lunch: Food, Wine`

Index(sub, [start], [end])	Searches the string for a specified value and returns the position of where it was found	`>>>mystr = "HelloPython"`
		`>>>print(mystr.index("P"))`
		`Output: 5`
		`>>>print(mystr.index("hon"))`

		Output: 8 >>>print(mystr.index("o")) Output: 4
isalnum	Returns True if all characters in the string are alphanumeric	>>>mystr = "HelloPython" >>>print(mystr.isalnum()) Output: True >>>a = "123" >>>print(a.isalnum()) Output: True >>>a= "$*%!!!" >>>print(a.isalnum()) Output: False
Isalpha()	Returns True if all characters in the string are in the alphabet	>>>mystr = "HelloPython" >>>print(mystr.isalpha()) Output: True >>>a = "123" >>>print(a.isalpha()) Output: False >>>a= "$*%!!!" >>>print(a.isalpha())

		Output: False
Isdecimal()	Returns True if all characters in the string are decimals	>>>mystr = "HelloPython" >>>print(mystr.isdecimal()) Output: False >>>a="1.23" >>>print(a.isdecimal()) Output: False >>>c = u"\u00B2" >>>print(c.isdecimal()) Output: False >>>c="133" >>>print(c.isdecimal()) Output: True
Isdigit()	Returns True if all characters in the string are digits	>>>c="133" >>>print(c.isdigit()) Output: True >>>c = u"\u00B2" >>>print(c.isdigit()) Output: True >>>a="1.23"

		>>>print(a.isdigit()) Output: False
isidentifier()	Returns True if the string is an identifier	>>>c="133" >>>print(c.isidentifier()) Output: False >>>c="_user_123" >>>print(c.isidentifier()) Output: True >>>c="Python" >>>print(c.isidentifier()) Output: True
Islower()	Returns True if all characters in the string are lower case	>>>c="Python" >>>print(c.islower()) Output: False >>> c="_user_123" >>>print(c.islower()) Output: True >>>print(c.islower()) Output: False
Isnumeric()	Returns True if all characters in the string are numeric	>>>c="133" >>>print(c.isnumeric())

		Output: True >>>c="_user_123" >>>print(c.isnumeric()) Output: False >>>c="Python" >>>print(c.isnumeric()) Output: False
isprintable()	Returns True if all characters in the string are printable	>>>c="133" >>>print(c.isprintable()) Output: True >>>c="_user_123" >>>print(c.isprintable()) Output: True >>>c="\t" >>>print(c.isprintable()) Output: False
isspace()	Returns True if all characters in the string are whitespaces	>>>c="133" >>>print(c.isspace()) Output: False

		>>> c="Hello Python"
		>>>print(c.isspace())
		Output: False
		>>>c="Hello"
		>>>print(c.isspace())
		Output: False
		>>> c="\t"
		>>>print(c.isspace())
		Output: True

istitle()	Returns True if the string follows the rules of a title	>>>c="133" >>>print(c.istitle()) Output: False >>>c="Python" >>>print(c.istitle()) Output: True >>>c="\t" >>>print(c.istitle()) Output: False
isupper()	Returns True if all characters in the string are upper case	>>>c="Python" >>>print(c.isupper()) Output: False >>> c="PYHTON" >>>print(c.isupper()) Output: True

		>>>c="\t" >>>print(c.isupper()) Output: False
join(iterable)	Joins the elements of an iterable to the end of the string	>>> a ="-" >>>print(a.join("123")) Output: 1-2-3 >>>a="Hello Python" >>>a="**" >>>print(a.join("Hello Python")) Output: H**e**l**l**o** **P**y**t**h**o**n
ljust(*width*[,*fillchar*])	Returns a left justified version of the string	>>>a="Hello" >>>b = a.ljust(12, "_") >>>print(b) Output: Hello_____
lower()	Converts a string into lower case	>>>a = "Python" >>>print(a.lower()) Output: python
lstrip([*chars*])	Returns a left trim version of the string	>>>a = " Hello "

		`>>>print(a.lstrip(), "!")` `Output: Hello`

maketrans(*x*[, *y*[, *z*]])	Returns a translation table to be used in translations	`>>>frm = "SecretCode"`
		`>>>to = "4203040540"`
		`>>>trans_table = str.maketrans(frm,to)`
		`>>>sec_code = "Secret Code".translate(trans_table)`
		`>>>print(sec_code)`
		`Output: 400304 0540`

partition(*sep*)	Returns a tuple where the string is parsed into three parts	`>>>mystr = "Hello-Python"`
		`>>>print(mystr.partition("-"))`
		`Output: ('Hello', '-', 'Python')`
		`>>>print(mystr.partition("."))`
		`Output: ('Hello-Python', '', '')`

replace(*old*, *new*[,*count*])	Returns a string where a specified value is replaced	`>>>mystr = "Hello Python.Hello Java.`

	with a specified value	`Hello C++."`
		`>>>print(mystr.replace("Hello", "Bye"))`
		`Output: Bye Python. Bye Java. Bye C++.`
		`>>>print(mystr.replace("Hello","Hell", 2))`
		`Output: Hell Python. Hell Java.Hello C++.`

rfind(*sub*[, *start*[,*end*]])	Searches the string for a specified value and returns the last position of where it was found	`>>>mystr = "Hello-Python"`
		`>>>print(mystr.rfind("P"))`
		`Output: 6`
		`>>>print(mystr.rfind("-"))`
		`Output: 5`
		`>>>print(mystr.rfind("z"))`
		`Output: -1`

rindex(*sub*[, *start*[,*end*]])	Searches the string for a specified value and returns the last position of where it was found	`>>>mystr = "Hello-Python"`
		`>>>print(mystr.rindex("P"))`
		`Output: 6`

		>>>print(mystr.rindex("-")) Output: 5 >>>print(mystr.rindex("z")) Traceback (most recent call last): File "<pyshell#253>", line1, in <module> print(mystr.rindex("z")) ValueError: substring not found
rjust(*width*[,*fillchar*])	Returns the string right justified in a string of length *width*.	>>>mystr = "Hello Python" >>>mystr1 = mystr.rjust(20,"-") >>>print(mystr1) Output: --------Hello Python
rpartition(*sep*)	Returns a tuple where the string is parted into three parts	>>>mystr = "Hello Python" >>>print(mystr.rpartition(".")) Output: ('', '', 'Hello Python') >>>print(mystr.rparti

		`tion(" "))` `Output: ('Hello', ' ', 'Python')`

rsplit(sep=None, maxsplit=-1)	Splits the string at the specified separator, and returns a list	`>>>mystr = "Hello Python"`
		`>>>print(mystr.rsplit())`
		`Output: ['Hello', 'Python']`
		`>>>mystr = "Hello-Python-Hello"`
		`>>>print(mystr.rsplit(sep="-",maxsplit=1))`
		`Output: ['Hello-Python', 'Hello']`

rstrip([*chars*])	Returns a right trim version of the string	`>>>mystr = "Hello Python"`
		`>>>print(mystr.rstrip(),"!")`
		`Output: Hello Python !`
		`>>>mystr = "----------Hello Python----------"`
		`>>>print(mystr.rstrip(), "-")`
		`Output:`

		----------Hello Python---------- - >>>print(mystr.rstrip(), "_") Output: ----------Hello Python---- ------- _
split(sep=None, maxsplit=-1)	Splits the string at the specified separator, and returns a list	>>>mystr = "Hello Python" >>>print(mystr.split()) Output: ['Hello', 'Python'] >>>mystr1="Hello,,Python" >>>print(mystr1.split(",")) Output: ['Hello', '', 'Python']
splitlines([keepends])	Splits the string at line breaks and returns a list	>>>mystr ="Hello:\n\n Python\r\nJava\nC++\n" >>>print(mystr.splitlines()) Output: ['Hello:', '', ' Python','Java','C++']

		`>>>print(mystr.splitlines(keepends=True))`
		`Output: ['Hello:\n', '\n', '`
		`Python\r\n', 'Java\n',`
		`'C++\n']`

startswith(*prefix*[,*start*[, *end*]])	Returns true if the string starts with the specified value	`>>>mystr = "Hello Python"`
		`>>>print(mystr.startswith("P"))`
		`Output: False`
		`>>>print(mystr.startswith("H"))`
		`Output: True`
		`>>>print(mystr.startswith("Hell"))`
		`Output: True`

strip([*chars*])	Returns a trimmed version of the string	`>>>mystr = "Hello Python"`
		`>>>print(mystr.strip(),"!")`
		`Output: Hello Python !`

		`>>>print(mystr.strip(), "")` `Output: Hello Python`

swapcase()	Swaps cases, lower case becomes upper case and vice versa	`>>>mystr = "Hello PYthon"`
		`>>>print(mystr.swapcase())`
		`Output: hELLO python`

title()	Converts the first character of each word to upper case	`>> mystr = "Hello PYthon"`
		`>>>print(mystr.title())`
		`Output: Hello Python`
		`>>>mystr = "HELLO JAVA"`
		`>>>print(mystr.title())`
		`Output: Hello Java`

translate(*table*)	Returns a translated string	`>>>frm = "helloPython"`
		`>>>to = "40250666333"`
		`>>>trans_table = str.maketrans(frm,to)`
		`>>>secret_code = "SecretCode".translate(trans_table)`

		`>>>print(secret_code)` `Output: S0cr06 C3d0`

upper()	Converts a string into upper case	`>>>mystr = "hello Python"`
		`>>>print(mystr.upper())`
		`Output: HELLO PYTHON`

zfill(*width*)	Fills the string with a specified number of 0 values at the beginning	`>>>mystr = "999"`
		`>>>print(mystr.zfill(9))`
		`Output: 000000999`
		`>>>mystr = "-40"`
		`>>>print(mystr.zfill(5))`
		`Output: -0040`

Boolean (`bool`)

Boolean type provides two built-in values, `True` and `False`. These values are used to determine the given statement true or false. It is denoted by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'.

The “truthiness” of an object of the Boolean type is self-evident: Boolean objects that are equal to `True` are truthy (true), and those equal to `False` are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false.

A Simple Program to Demonstrate Python Boolean Data Types

Program:

```
print(type(True))
print(type(False))
print(false)
```

Output:

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

List (`list`)

Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). In a simple language, a list is a collection of things, enclosed in [ ] and separated by commas. Lists are the simplest containers that are an integral part of the Python language. Lists need not be homogeneous always which makes it the most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

We can use slice `[:]` operators to access the data of the list. The concatenation operator (`+`) and repetition operator (`*`) work with the list in the same way as they were working with the strings.

1. **Adding Elements to a List**
    - Using the `append()` method

      Elements can be added to the List by using the built-in `append()` function. Only one element at a time can be added to the list by using the `append()` method, for the addition of multiple elements with the `append()` method,

loops are used. Tuples can also be added to the list with the use of the append method because tuples are immutable. Unlike Sets, Lists can also be added to the existing list with the use of the `append()` method.

- Using the `insert()` method

  `append()` method only works for the addition of elements at the end of the List, for the addition of elements at the desired position, the `insert()` method is used. Unlike `append()` which takes only one argument, the `insert()` method requires two arguments(position, value).

- Using `extend()` method

  Other than `append()` and `insert()` methods, there's one more method for the Addition of elements, `extend()` this method is used to add multiple elements at the same time at the end of the list.

2. **Accessing elements from the List**

   In order to access the list items refer to the index number. Use the index operator [ ] to access an item in a list. The index must be an integer. Nested lists are accessed using nested indexing.

   Negative indexing

   In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

3. **Reversing a List**

   A list can be reversed by using the (`list.reverse()`) method

4. **Removing Elements from the List**
    - Using the `remove()` method

      Elements can be removed from the list by using the built-in `remove()` function but an Error arises if the element doesn't exist in the list. `remove()` method only removes one element at a time, to remove a range of elements, the iterator is used. The `remove()` method removes the specified item.

    - Using the `pop()` method

      `pop()` function can also be used to remove and return an element from the list, but by default it removes only the last element of the list, to remove an element from a specific position of the List, the index of the element is passed as an argument to the `pop()` method.

5. **Slicing of a List**

   We can get substrings and sublists using a slice. In Python List, there are multiple ways to print the whole list with all the elements, but to print a specific range of elements from the list, we use the Slice operation. Slice operation is performed on Lists with the use of a colon(`:`). To print elements from beginning to a range use `[:Index]`, to print elements from end-use `[:-Index]`, to print elements from specific Index till the end use [Index:], to print elements within a range, use `[Start Index: End Index]` and to print the whole List with the use of slicing operation, use `[:]`. Further, to print the whole list in reverse order, use `[::-1]`.

6. **List Comprehension**

   Chances are you'll use a lot of lists as a Python programmer. While we're all big fans of for loops (and nested for loops), Python provides a more concise method for handling lists and list comprehension.

In order to keep your code elegant and readable, it's recommended that you use Python's comprehension features.

List comprehension is a powerful and concise method for creating lists in Python that becomes essential the more you work with lists and lists of lists.

```
my_new_list = [ expression for item in list ]
```

You can see from this example that three ingredients are necessary for a python list comprehension to work.

1. First is the expression we'd like to carry out. *expression* inside the square brackets.
2. Second is the object that the expression will work on. *item* inside the square brackets.
3. Finally, we need an iterable list of objects to build our new list from. *list* inside the square brackets

A Simple Program to Demonstrate Python List Data Types

Program:

```
list1  = [1, "hi", "Python", 2]
#Checking type of given list
print(type(list1))
#Printing the list1
print (list1)
# List slicing
print (list1[3:])
```

```
# List slicing
print (list1[0:2])
# List Concatenation using + operator
print (list1 + list1)
# List repetition using * operator
print (list1 * 3)
```

Output:

```
[1, 'hi', 'Python', 2]
[2]
[1, 'hi']
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
```

List Operations

Operator	Description	Example
Repetition	The repetition operator enables the list elements to be repeated multiple times.	`L1*2 = [1, 2, 3, 4, 1, 2, 3, 4]`
Concatenation	It concatenates the list mentioned on either side of the operator.	`l1+l2 = [1, 2, 3, 4, 5, 6, 7, 8]`

Membership	It returns true if a particular item exists in a particular list otherwise false.	`print(2 in l1) prints True.`

Iteration	The for loop is used to iterate over the list elements.	`for i in l1:`
		`  print(i)`
		`Output`
		`1`
		`2`
		`3`
		`4`

Length	It is used to get the length of the list	`len(l1) = 4`

List Built-in functions

Function	Description	Example
cmp(list1, list2)	It compares the elements of both the lists.	This method is not used in the Python 3 and the above versions.

len(list)	It is used to calculate the length of the list.	`L1= [1,2,3,4,5,6,7,8]`
		`print(len(L1))`
		`8`

max(list)	It returns the maximum element of the list.	`L1 = [12,34,26,48,72]`
		`print(max(L1))`
		`72`

min(list)	It returns the minimum element of the list.	`L1 = [12,34,26,48,72]`
		`print(min(L1))`
		`12`

list(seq)	It converts any sequence to the list.	`str = "Johnson"`
		`s = list(str)`
		`print(type(s))`
		`<class list>`

Tuple (`tuple`)

Python Tuple is used to store the sequence of immutable Python objects. The tuple is similar to lists since the value of the items stored in the list can be changed, whereas the tuple is immutable, and the value of the items stored in the tuple cannot be changed.

1. **Creating a tuple**

   A tuple can be written as the collection of comma-separated (,) values enclosed with the small () brackets. The parentheses are optional but it is good practice to use them.

2. **Tuple indexing and slicing**

   The indexing and slicing in the tuple are similar to lists. The indexing in the tuple starts from 0 and goes to length(tuple) - 1.

   The items in the tuple can be accessed by using the index [] operator. Python also allows us to use the colon operator to access multiple items in the tuple.

3. **Negative Indexing**

   The tuple element can also be accessed by using negative indexing. The index of -1 denotes the rightmost element and -2 to the second last item and so on. The elements from left to right are traversed using negative indexing.

4. **Deleting Tuple**

   Unlike lists, the tuple items cannot be deleted by using the `del` keyword as tuples are immutable. To delete an entire tuple, we can use the `del` keyword with the tuple name.

A Simple Program to Demonstrate Python Boolean Data Types

Program:

```
tup  = ("hi", "Python", 2)
# Checking type of tup
print (type(tup))
#Printing the tuple
print (tup)
# Tuple slicing
print (tup[1:])
print (tup[0:1])
# Tuple concatenation using + operator
print (tup + tup)
# Tuple reputation using * operator
print (tup * 3)
# Adding value to tup. It will throw an error.
t[2] = "hi"
```

```
Output:

<class 'tuple'>

('hi', 'Python', 2)

('Python', 2)

('hi',)

('hi', 'Python', 2, 'hi', 'Python', 2)

('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)

Traceback (most recent call last):

  File "main.py", line 14, in <module>

    t[2] = "hi";

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

Basic Tuple Operations

Operator	Description	Example
Repetition	The repetition operator enables the tuple elements to be repeated multiple times.	`T1*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5)`
Concatenation	It concatenates the tuple mentioned on either side of the operator.	`T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9)`

Membership	It returns true if a particular item exists in the tuple otherwise false	`print (2 in T1)` `prints True.`
Iteration	The for loop is used to iterate over the tuple elements.	`for i in T1:` `print(i)` `Output` `1` `2` `3` `4` `5`
Length	It is used to get the length of the tuple.	`len(T1) = 5`

Tuple InBuilt Functions

SN	Function	Description
1	cmp(tuple1, tuple2)	It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false.
2	len(tuple)	It calculates the length of the tuple.
3	max(tuple)	It returns the maximum element of the tuple
4	min(tuple)	It returns the minimum element of the tuple.

5	tuple(seq)	It converts the specified sequence to the tuple.

Dictionary (`dict`)

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which, unlike other data types that hold only a single value as an element, Dictionary holds key:value pair. Key-Value is provided in the dictionary to make it more optimized.

1. **Creating a Dictionary**
   In Python, a dictionary can be created by placing a sequence of elements within curly `{}` braces, separated by 'comma'. Dictionary holds pairs of values, one being the Key and the other corresponding pair element being its Key:value. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be *immutable*.
   Dictionary can also be created by the built-in function `dict()`. An empty dictionary can be created by just placing curly braces`{}`.
2. **Adding elements to a Dictionary**
   In the Python Dictionary, the addition of elements can be done in multiple ways. One value at a time can be added to a Dictionary by defining value along with the key e.g. `Dict[Key] = 'Value'`. Updating an existing value in a Dictionary can be done by using the built-in `update()` method. Nested key values can also be added to an existing Dictionary.
3. **Accessing elements of a Dictionary**
   In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets.

4. **Accessing an element of a nested dictionary**

   In order to access the value of any key in the nested dictionary, use indexing `[]`

5. **Dictionary Comprehension**

Like List Comprehension, Python allows dictionary comprehension. We can create dictionaries using simple expressions.
A dictionary comprehension takes the form

```
{key: value for (key, value) in iterable}
```

A Simple Program to Demonstrate Python Boolean Data Types

Program:

```
d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}

# Printing dictionary

print (d)

# Accessing value using keys

print("1st name is "+d[1])

print("2nd name is "+ d[4])

print (d.keys())

print (d.values())
```

Output:

```
1st name is Jimmy
2nd name is mike
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
dict_keys([1, 2, 3, 4])
dict_values(['Jimmy', 'Alex', 'john', 'mike'])
```

Built-in Dictionary Functions

Python includes the following dictionary functions −

Sr.No.	Function with Description

1	`cmp(dict1, dict2)` Compares elements of both dict.
2	`len(dict)` Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	`str(dict)` Produces a printable string representation of a dictionary
4	`type(variable)` Returns the type of the passed variable. If the passed variable is a dictionary, then it would return a dictionary type.

Built-in Dictionary Methods

Sr.No.	Methods with Description
1	`dict.clear()` Removes all elements of dictionary *dict*
2	`dict.copy()` Returns a shallow copy of dictionary *dict*
3	`dict.fromkeys()` Create a new dictionary with keys from seq and values *set* to *value*.

4	`dict.get(key, default=None)` For *key* key, returns value or default if key not in dictionary
5	`dict.has_key(key)` Returns *true* if key in dictionary *dict*, *false* otherwise
6	`dict.items()` Returns a list of *dict*'s (key, value) tuple pairs
7	`dict.keys()` Returns list of dictionary dict's keys
8	`dict.setdefault(key, default=None)` Similar to get(), but will set dict[key]=default if *key* is not already in dict
9	`dict.update(dict2)` Adds dictionary *dict2*'s key-values pairs to *dict*
10	`dict.values()` Returns list of dictionary *dict*'s values

Set (`set`)

In Python, a Set is an unordered collection of data types that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

1. **Creating a Set**

   Sets can be created by using the built-in `set()` function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a 'comma'. A set contains only unique elements but at the time of set creation, multiple duplicate values can also be passed. Order of elements in a set is undefined and is unchangeable. Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

2. **Adding Elements to a Set**
   - Using `add()` method

     Elements can be added to the Set by using the built-in `add()` function. Only one element at a time can be added to the set by using `add()` method, loops are used to add multiple elements at a time with the use of `add()` method.

   - Using `update()` method

     For the addition of two or more elements `update()` method is used. The `update()` method accepts lists, strings, tuples as well as other sets as its arguments. In all of these cases, duplicate elements are avoided.

3. **Accessing a Set**

   Set items cannot be accessed by referring to an index, since sets are unordered and the items have no index. But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

4. **Removing elements from the Set**
   - Using `remove()` method or `discard()` method

Elements can be removed from the Set by using the built-in `remove()` function but a KeyError arises if the element doesn't exist in the set. To remove elements from a set without KeyError, use `discard()`, if the element doesn't exist in the set, it remains unchanged.

- Using `pop()` method

  `pop()` function can also be used to remove and return an element from the set, but it removes only the last element of the set.

- Using `clear()` method

  To remove all the elements from the set, `clear()` function is used.

A Simple Program to Demonstrate Python Boolean Data Types

Program:

```
# Creating Empty set
set1 = set()
set2 = {'James', 2, 3,'Python'}
#Printing Set value
print(set2)
# Adding element to the set
set2.add(10)
print(set2)
#Removing element from the set
set2.remove(2)
print(set2)
```

Output:

```
{3, 'Python', 'James', 2}
{'Python', 'James', 3, 2, 10}
```

```
{'Python', 'James', 3, 10}
```

Built-in Set Methods

SN	Method	Description
1	add(item)	It adds an item to the set. It has no effect if the item is already present in the set.
2	clear()	It deletes all the items from the set.
3	copy()	It returns a shallow copy of the set.
4	difference_update(....)	It modifies this set by removing all the items that are also present in the specified sets.
5	discard(item)	It removes the specified item from the set.
6	intersection()	It returns a new set that contains only the common elements of both the sets. (all the sets if more than two are specified).

7	intersection_update(....)	It removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).
8	Isdisjoint(....)	Return True if two sets have a null intersection.
9	Issubset(....)	Report whether another set contains this set.
10	Issuperset(....)	Report whether this set contains another set.
11	pop()	Remove and return an arbitrary set element that is the last element of the set. Raises KeyError if the set is empty.
12	remove(item)	Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.
13	symmetric_difference(....)	Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.

14	symmetric_difference_update(....)	Update a set with the symmetric difference of itself and another.
15	union(....)	Return the union of sets as a new set. (i.e. all elements that are in either set.)
16	update()	Update a set with the union of itself and others.

Types of Operator

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Membership Operators
- Identity Operators

Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
`+` Addition	Adds values on either side of the operator.	`a + b = 30`
`-` Subtraction	Subtracts right-hand operand from left-hand operand.	`a - b = -10`
`*` Multiplication	Multiplies values on either side of the operator	`a * b = 200`
`/` Division	Divides left-hand operand by right-hand operand	`b / a = 2`
`%` Modulus	Divides left-hand operand by right-hand operand and returns remainder	`b % a = 0`
`**` Exponent	Performs exponential	`a**b` =10 to the power 20

	(power) calculation on operators	
`//`Floor Division	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`

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 10 and variable b holds 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 the condition becomes true.	(`a != b`) is true.

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

Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns a value of a + b into c
+= Add AND	It adds right operand to the left operand and assigns the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assigns the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assigns 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 / a
%= Modulus AND	It takes modulus using two operands and assigns the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential	c **= a is equivalent to c = c

	(power) calculation on operators and assign value to the left operand	** a
`//=` Floor Division	It performs floor division on operators and assigns value to the left operand	c `//=` a is equivalent to c = c // a

Python Logical Operators

There are following logical operators supported by the Python language. Assume variable a holds 10 and variable b holds 20 then

Operator	Description	Example
`and` Logical AND	If both the operands are true then the condition becomes true.	`(a and b)` is true.
`or` Logical OR	If any of the two operands are non-zero then the condition becomes true.	`(a or b)` is true.
`not` Logical NOT	Used to reverse the logical state of its operand.	`not(a and b)` is false.

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below −

Operator	Description	Example
`in`	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	`x in y`, here results in a 1 if x is a member of sequence y.
`not in`	Evaluates to true if it does not find a variable in the specified sequence and false otherwise.	`x not in y`, here not in results in a 1 if x is not a member of sequence y.

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below −

Operator	Description	Example
`is`	Evaluates as true if the variables on either side of the operator point to the same object and false otherwise.	`x is y`, here it results in 1 if id(x) equals id(y).
`is not`	Evaluates as false if the	`x is not y`, there is no

	variables on either side of the operator pointing to the same object, and true otherwise.	results in 1 if id(x) is not equal to id(y).

Taking input in Python

Developers often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. While Python provides us with two inbuilt functions to read the input from the keyboard.

- `input ( prompt )`

`input ()`: This function first takes the input from the user and converts it into a string. The type of the returned object always will be `<type 'str'>`. It does not evaluate the expression, it just returns the complete statement as String. For example, Python provides a built-in function called input which takes the input from the user. When the input function is called it stops the program and waits for the user's input. When the user presses enter, the program resumes and returns what the user typed.

How the input function works in Python:

- When the `input()` function executes, program flow will be stopped until the user has given input.
- The text or message displayed on the output screen to ask a user to enter an input value is optional i.e. the prompt, which will be printed on the screen is optional.
- Whatever you enter as input, the input function converts it into a string. if you enter an integer value, the `input()` function converts it into a string. You need to explicitly convert it into an integer in your code using typecasting.
- This is how the input function takes it.

A Simple Program to Demonstrate Python Inputs

Program:

```
print("Enter a string")
a=input()
print("The string entered by user is",a)
```

Output:

```
Enter a string
Python
The string entered by user is Python
```

TypeCasting

The process of converting one data type to another data type is called Typecasting or Type Coercion or Type Conversion.

The topics that I'll be focusing on in this article are:

- Implicit Type Conversion
- Explicit Type Conversion

Implicit Type Conversion

When the type conversion is performed automatically by the interpreter without the programmer's intervention, that type of conversion is referred to as implicit type conversion.

Explicit Type Conversion

In this method, Python needs user involvement to convert the variable data type into a certain data type in order to the operation required.

Mainly type casting can be done with these data type functions:

- **`Int()`**: `int()` function takes a float or string as an argument and returns an int type object.
- **`float()`** : `float()` function takes int or string as an argument and returns a float type object.
- **`str()`** : `str()` function takes float or int as an argument and returns a string type object.

A Simple Programs to Demonstrate Python TypeCasting

Program:

```
#program to demonstrate implicit type conversion
#initializing the value of a
a = 10
print(a)
print("The type of a is ", type(a))
#initializing the value of b
b = 4.5
print(b)
print("The type of b is ", type(b))
#initializing the value of c
c = 4.0
print(c)
print("The type of c is ", type(c))
#initializing the value of d
d = 5.0
print(d)
print("The type of d is ", type(d))
#performing arithmetic operations
res = a * b
print("The product of a and b is ", res)
add = c + d
print("The addition of c and d is ", add)
```

Outcome:

```
10
The type of a is <class 'int'>
4.5
The type of b is <class 'float'>
4.0
The type of c is <class 'float'>
5.0
The type of d is <class 'float'>
The product of a and b is  45.0
The addition of c and d is  9.0
```

Program:

```
#program to demonstrate explicit type conversion
#initializing the value of a
a = 10.6
print("The type of 'a' before typecasting is ",type(a))
print(int(a))
print("The type of 'a' after typecasting is",type(a))
#initializing the value of b
b = 8.3
print("The type of 'b' before typecasting is ",type(b))
print(int(b))
print("The type of 'b' after typecasting  is",type(b))
#initializing the value of c
c = 7
print("The type of 'c' before typecasting is ",type(c))
print(float(c))
```

```
print("The type of 'c' after typecasting is",type(c))
```

Output:

```
The type of 'a' before typecasting is  <class 'float'>
10
The type of 'a' after typecasting is <class 'float'>
The type of 'b' before typecasting is  <class 'float'>
8
The type of 'b' after typecasting  is <class 'float'>
The type of 'c' before typecasting is  <class 'int'>
7.0
The type of 'c' after typecasting is <class 'int'>
```

Conditional Statements

Conditional statements are also called decision-making statements. Let's discuss some conditions making statements in detail. There are three types of conditional statements in python.

How to use "`if` condition"

Python `if` Statement is used for decision-making operations. It contains a body of code that runs only when the condition given in the if statement is true. If the condition is false, then the optional else statement runs which contains some code for the else condition.

When you want to justify one condition while the other condition is not true, then you use Python's if-else statement.

How to use "`else` condition"

The "`else` condition" is usually used when you have to judge one statement on the basis of another. If one condition goes wrong, then there should be another condition that should justify the statement or logic.

When "`else` condition" does not work

There might be many instances when your "`else` condition" won't give you the desired result. It will print out the wrong result as there is a mistake in program logic. In most cases, this happens when you have to justify more than two statements or conditions in a program.

How to use "`elif`" condition

To correct the previous error made by "else condition", we can use the "`elif`" statement. By using the "`elif`" condition, you are telling the program to print out the third condition or possibility when the other condition goes wrong or incorrect.

Python nested IF statements

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 `if` construct.

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

A Simple Programs to Demonstrate Python Conditional Statements

Program:

```
a = int(input("Enter a? "))
b = int(input("Enter b? "))
c = int(input("Enter c? "))
if a>b and a>c:
    print("a is largest")
if b>a and b>c:
    print("b is largest")
if c>a and c>b:
    print("c is largest")
```

Output:

```
Enter a? 100
Enter b? 120
Enter c? 130
c is largest
```

Program:

```
num = int(input("enter the number?"))
```

```
if num%2 == 0:
    print("Number is even...")
else:
    print("Number is odd...")
```

Output:

```
enter the number?10
Number is even
```

Program:

```
number = int(input("Enter the number?"))
if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50")
elif number==100:
    print("number is equal to 100")
else:
    print("number is not equal to 10, 50 or 100")
```

Output:

```
Enter the number?15
number is not equal to 10, 50 or 100
```

Looping Statements

If we want to execute a group of statements multiple times, then we should go for a looping kind of execution. There are two types of loops available in python. They are:

1. while loop
2. for loop

While Loop

The while loop contains an expression/condition. As per the syntax colon (:) is mandatory otherwise it throws a syntax error. The condition gives the result as bool type, either True or False. The loop keeps on executing the statements until the condition becomes False.

Parts of While Loop

Initialization: This is the first part of the while loop. Before entering the condition section, some initialization is required.

Condition: Once the initializations are done, then it will go for condition checking which is the heart of the while loop. The condition is checked and if it returns True, then execution enters the loop for executing the statements inside.

After executing the statements, the execution goes to the increment/decrement section to increment the iterator. Mostly, the condition will be based on this iterator value, which keeps on changing for each iteration. This completes the first iteration. In the second iteration, if the condition is False, then the execution comes out of the loop else it will proceed as explained in the above point.

Increment/Decrement section: This is the section where the iterator is increased or decreased. Generally, we use arithmetic operators in the loop for this section.

For Loop

Basically, a for loop is used to iterate elements one by one from sequences like string, list, tuple, etc. This loop can be easily understood when compared to the while loop. While iterating elements from the sequence we can perform operations on every element.

For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is a “for in” loop which is similar to for each loop in other languages.

Iterating by the index of sequences: We can also use the index of elements in the sequence to iterate. The key idea is to first calculate the length of the list and then iterate over the sequence within the range of this length.

A Simple Programs to Demonstrate Python Looping Statements

Program:

```
# Code to find the sum of squares of each element of the list using
for loop
# creating the list of numbers
numbers = [3, 5, 23, 6, 5, 1, 2, 9, 8]
# initializing a variable that will store the sum
sum_ = 0
# using for loop to iterate over the list
for num in numbers:
   sum_ = sum_ + num ** 2
print("The sum of squares is: ", sum_)
```

Output:

```
The sum of squares is:  774
```

Program:

```
import random
numbers = [ ]
for val in range(0, 11):
    numbers.append( random.randint( 0, 11 ) )
for num in range( 0, 11 ):
    for i in numbers:
        if num == i:
            print( num, end = " " )
```

Output:

```
0 2 4 5 6 7 8 8 9 10
```

Program:

```
# The control transfer is transfered
# when break statement soon it sees t
i = 0
str1 = 'javatpoint'
while i < len(str1):
    if str1[i] == 't':
        i += 1
        break
    print('Current Letter :', str1[i])
    i += 1
```

Output:

```
Current Letter : j
Current Letter : a
Current Letter : v
Current Letter : a
```

Functions

Python Functions are very easy to write in python and all non-trivial programs will have functions. Function names have the same rules as variable names. The function is a block of related statements designed to perform a computational, logical, or evaluative task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

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 first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- 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.

Calling a Function

After creating a function we can call it by using the name of the function followed by parenthesis containing parameters of that particular function.

Arguments of a Function

Arguments are the values passed inside the parentheses of the function. A function can have any number of arguments separated by a comma.

Types of Arguments

Python supports various types of arguments that can be passed at the time of the function call. Let's discuss each type in detail.

1. **Default arguments**

   A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument.

2. **Keyword arguments**

   The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

3. **Variable-length arguments**

   In Python, we can pass a variable number of arguments to a function using special symbols. There are two special symbols:

   - *args (Non-Keyword Arguments)
   - **kwargs (Keyword Arguments)

return statement

The function return statement is used to exit from a function and go back to the function caller and return the specified value or data item to the caller.

A Simple Program to Demonstrate Python Functions

Program:

```
def square( num ):
    """
    This function computes the square of the number.
    """
    return num**2
object_ = square(9)
print( "The square of the number is: ", object_ )
```

Output:

```
The square of the number is:  81
```

Anonymous Functions (`lambda`)

In Python, an anonymous function means that a function is without a name. As we already know the def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming besides other types of expressions in functions.

A Simple Programs to Demonstrate Python `lambda` Functions

Program:

```
# Python program to demonstrate
# lambda functions


string ='GeeksforGeeks'

# lambda returns a function object
print(lambda string : string)
```

Output:

```
<function <lambda> at 0x7f65e6bbce18>
```

Program:

```
# Python program to demonstrate
# lambda functions


x ="GeeksforGeeks"

# lambda gets pass to
print(lambda x : print(x))(x)
```

Output:

```
GeeksforGeeks
```

`map()` Function

map() function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

Parameters :

fun : It is a function to which a map passes each element of a given iterable.

iter : It is an iterable which is to be mapped.

The returned value from map() (map object) then can be passed to functions like list() (to create a list), set() (to create a set) .

A Simple Program to Demonstrate Python `map()` Functions

Program:

```
def calculateAddition(n):
  return n+n
numbers = (1, 2, 3, 4)
result = map(calculateAddition, numbers)
print(result)
# converting map object to set
numbersAddition = set(result)
print(numbersAddition)
```

Output:

```
<map object at 0x7fb04a6bec18>
{8, 2, 4, 6}
```

Using `lambda()` Function with `map()`

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

Program:

```
# Python code to illustrate
# map() with lambda()
# to get double of a list.
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(map(lambda x: x*2, li))
print(final_list)
```

Output:

```
[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]
```

`reduce()` Function

The reduce(fun,seq) function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along.This function is defined in the “functools” module.

Working :

- At the first step, the first two elements of sequence are picked and the result is obtained.
- Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.
- This process continues till no more elements are left in the container.

- The final returned result is returned and printed on the console.

A Simple Program to Demonstrate Python `reduce()` Functions

Program:

```
# python code to demonstrate working of reduce()
# using operator functions

# importing functools for reduce()
import functools

# importing operator for operator functions
import operator

# initializing list
lis = [1, 3, 5, 6, 2 ]

# using reduce to compute sum of list
# using operator functions
print("The sum of the list elements is : ", end="")
print(functools.reduce(operator.add, lis))

# using reduce to compute product
# using operator functions
print("The product of list elements is : ", end="")
print(functools.reduce(operator.mul, lis))

# using reduce to concatenate string
print("The concatenated product is : ", end="")
print(functools.reduce(operator.add, ["geeks", "for", "geeks"]))
```

Output:

```
The sum of the list elements is : 17
The product of list elements is : 180
```

```
The concatenated product is : geeksforgeeks
```

Using `lambda()` Function with `reduce()`

The `reduce()` function in Python takes in a function and a list as an argument. The function is called with a lambda function and an iterable and a new reduced result is returned. This performs a repetitive operation over the pairs of the iterable. The `reduce()` function belongs to the *functools* module.

Program:

```
# reduce() with lambda()
# Python code to illustrate
# to get sum of a list
from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print(sum)
```

Output:

```
193
```

`filter()` Function

The `filter()` method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

A Simple Program to Demonstrate Python `filter()` Functions

Program:

```
# function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False
# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']
# using filter function
filtered = filter(fun, sequence)
print('The filtered letters are:')
for s in filtered:
    print(s)
```

Output:

```
The filtered letters are:
e
e
```

Using `lambda()` Function with `filter()`

The `filter()` function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence “sequence”, for which the function returns True.

Program:

```
# Python code to illustrate
# filter() with lambda()
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final_list = list(filter(lambda x: (x%2 != 0) , li))
print(final_list)
```

Output:

```
[5, 7, 97, 77, 23, 73, 61]
```

`range()` Function

Python `range()` function for loop is commonly used hence, knowledge of the same is the key aspect when dealing with any kind of Python code. The most common use of the `range()` function in Python is to iterate sequence type (Python `range()` List, string, etc.) with for and while loop.

In simple terms, `range()` allows the user to generate a series of numbers within a given range. Depending on how many arguments the user is passing to the function, the user can decide where that series of numbers will begin and end as well as how big the difference will be between one number and the next.`range()` takes mainly three arguments.

- **start**: integer starting from which the sequence of integers is to be returned
- **stop**: integer before which the sequence of integers is to be returned. The range of integers ends at stop – 1.
- **step**: integer value which determines the increment between each integer in the sequence.

There are three ways you can call range() :

- `range(stop)` takes one argument.
- `range(start, stop)` takes two arguments.
- `range(start, stop, step)` takes three arguments.

`range(stop)`

When a user calls `range()` with one argument, the user will get a series of numbers that starts at 0 and includes every whole number up to, but not including, the number that user has provided as the stop.

`range(start, stop)`

When a user calls `range()` with two arguments, the user gets to decide not only where the series of numbers stops but also where it starts, so users don't have to start at 0 all the time. Users can use `range()` to generate a series of numbers from X to Y using a `range(X, Y)`.

`range(start, stop, step)`

When the user calls `range()` with three arguments, the user can choose not only where the series of numbers will start and stop but also how big the difference will be between one number and the next. If the user doesn't provide a step, then `range()` will automatically behave as if the step is 1.

Points to remember about Python range() function :

- `range()` function only works with the integers i.e. whole numbers.
- All arguments must be integers. Users can not pass a string or float number or any other type in a start, stop and step argument of a `range()`.
- All three arguments can be positive or negative.
- The step value must not be zero. If a step is zero python raises a `ValueError` exception.

- `range()` is a type in Python
- Users can access items in a `range()` by index, just as users do with a list.

A Simple Program to Demonstrate Python `range()` Functions

Program:

```
# empty range
print(list(range(0)))
# using the range(stop)
print(list(range(4)))
# using the range(start, stop)
print(list(range(1,7 )))
start = 5
stop = 12
step = 4
print(list(range(start, stop, step)))
```

Output:

```
[]
[0, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
[5, 9]
```

Modules

A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.

Modules in Python provide us the flexibility to organize the code in a logical way.

To use the functionality of one module into another, we must have to import the specific module.

Loading the Module in our Python Code

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

1. The import statement
2. The from-import statement

The `import` statement

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times it has been imported into our file.

The `from-import` statement

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement.

A Simple Programs to Demonstrate Python Modules

Create a Simple Module

The Python code for a module named aname normally resides in a file named aname.py. Here's an example of a simple module, support.py

Program:

```
def print_func( par ):
  print "Hello : ", par
  return
```

The `import` statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax −

Syntax:

```
import module1[, module2[,... moduleN]
```

Program:

```
# Import module support
import support
```

```
# Now you can call defined function that module as follows
support.print_func("Zara")
```

Output:

```
Hello : Zara
```

The `from-import` statement

Python from statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax −

Syntax:

```
from modname import name1[, name2[, ... nameN]]
```

For example, to import the function fibonacci from the module fib, use the following statement −

```
from fib import fibonacci
```

File Handling

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but like other concepts of Python, this concept here is also easy and short. Python treats files differently as text or binary and this is important. Each line of code includes a sequence of characters and they form a text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let's start with the reading and writing files.

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

- Open a file
- Read or write - Performing operation
- Close the file

Opening a File

Python provides an `open()` function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

SN	Access mode	Description
1	`r`	It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	`rb`	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
3	`r+`	It opens the file to read and write both. The file pointer exists at the beginning of the file.
4	`rb+`	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
5	`w`	It opens the file to write only. It overwrites the file if it previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
6	`wb`	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file.
7	`w+`	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file.

8	`wb+`	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	`a`	It opens the file in the append mode. The file pointer exists at the end of the previously written file if there exists any. It creates a new file if no file exists with the same name.
10	`ab`	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
11	`a+`	It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
12	`ab+`	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

The `close()` Method

Once all the operations are done on the file, we must close it through our Python script using the `close()` method. Any unwritten information gets destroyed once the `close()` method is called on a file object.

We can perform any operation on the file externally using the file system which is currently opened in Python; hence it is good practice to close the file once all the operations are done. file externally using the file system which is currently opened in Python; hence it is good practice to close the file once all the operations are done.

A Simple Programs to Demonstrate Python File Handling

File Object Attributes

If an attempt to open a file fails, then open returns a false value. Otherwise, it returns a file object that provides various information related to that file.

Program:

```
# file opening example in Python

fo = open("sample.txt", "wb")
print ("File Name: ", fo.name)
print ("Mode of Opening: ", fo.mode)
print ("Is Closed: ", fo.closed)
print ("Softspace flag : ", fo.softspace)
```

Output:

```
File Name: sample.txt
Mode of Opening: wb
Is Closed: False
Softspace flag: 0
```

File Reading in Python

For reading and writing text data, different text-encoding schemes are used, such as ASCII (American Standard Code for Information Interchange), UTF-8 (Unicode Transformation Format), UTF-16.

Once a file is opened using an open() method, then it can be read by a method called read().

Program:

```
# read the entire file as one string
with open('filename.txt') as f:
data = f.read()
# Iterate over the lines of the File
with open('filename.txt') as f:
for line in f :

   print(line, end=' ')
# process the lines
```

File Writing in Python

Similarly, for writing data to files, we have to use open() with 'wt' mode, clearing and overwriting the previous content. Also, we have to use the write() function to write into a file.

Program:

```
# Write text data to a file
with open('filename.txt' , 'wt') as f:
   f.write ('hi there, this is a first line of file.\n')
   f.write ('and another line.\n')
```

Output:

```
hi there, this is a first line of file.
and another line.
```

Closing a File in Python

In Python, it is not system critical to close all your files after using them, because the file will auto close after Python code finishes execution. You can close a file by using the close() method.

Syntax:

```
file_object.close();
```

Program:

```
try:
  # Open a file
  fo = open("sample.txt", "wb")
  # perform file operations
finally:
  # Close opened file
   fo.close()
```

Exception Handling

Exceptions are events that are used to modify the flow of control through a program when the error occurs. Exceptions get triggered automatically on finding errors in Python.

These exceptions are processed using five statements. These are:

1. **try/except**: catch the error and recover from exceptions hoisted by programmers or Python itself.
2. **try/finally**: Whether an exception occurs or not, it automatically performs the clean-up action.
3. **assert**: triggers an exception conditionally in the code.
4. **raise**: manually triggers an exception in the code.
5. **with/as**: implement context managers in older versions of Python such as - Python 2.6 & Python 3.0.

Why are Exceptions Used?

Exceptions allow us to jump out of random, illogical large chunks of codes in case of errors. Let us take a scenario in which you have given a function to do a specific task. If you went there and found those things missing that are required to do that particular task, what would you do? Either you stop working or think about a solution - where to find those items to perform the task. The same thing happens here in case of Exceptions also. Exceptions allow programmers to jump an exception handler in a single step, abandoning all function calls. You can think of exceptions to an optimized quality go-to statement, in which the program error that occurs at runtime gets easily managed by the exception block. When the interpreter encounters an error, it lets the execution go to the exception part to solve and continue the execution instead of stopping.

While dealing with exceptions, the exception handler creates a mark & executes some code. Somewhere further within that program, the exception is raised that solves the problem & makes Python jump back to the marked location; by not throwing away/skipping any active functions that were called after the marker was left.

Roles of an Exception Handler in Python

- **Error handling**: The exceptions get raised whenever Python detects an error in a program at runtime. As a programmer, if you don't want the default behavior, then code a 'try' statement to catch and recover the program from an exception. Python will jump to the 'try' handler when the program detects an error; the execution will be resumed.
- **Event Notification**: Exceptions are also used to signal suitable conditions & then passing result flags around a program and text them explicitly.
- **Terminate Execution**: There may arise some problems or errors in programs that need a termination. So try/finally is used to guarantee that closing-time operation will be performed. The 'with' statement offers an alternative for objects that support it.
- **Exotic flow of Control**: Programmers can also use exceptions as a basis for implementing unusual control flow. Since there is no 'go to' statement in Python so that exceptions can help in this respect.

A Simple Programs to Demonstrate Python Exception Handling

Program:

```
(a,b) = (6,0)
try:# simple use of try-except block for handling errors
    g = a/b
except ZeroDivisionError:
    print ("This is a DIVIDED BY ZERO error")
```

Output:

```
This is a DIVIDED BY ZERO error
```

Program:

```
(a,b) = (6,0)
try:
    g = a/b
except ZeroDivisionError as s:
    k = s
    print (k)
#Output will be: integer division or modulo by zero
```

Output:

```
division by zero
```

The ‘try - Except’ Clause with No Exception

Syntax:

```
try:
        # all operations are done within this block.
        . . . . . .
except:
        # this block will get executed if any exception is
encountered.
        . . . . . .
else:
        # this block will get executed if no exception is found.
        . . . . . .
```

‘except’ Clause with Multiple Exception

Syntax:

```
try:
        # all operations are done within this block.
        . . . . . .
except ( Exception1 [, Exception2[,….Exception N ] ] ] ) :
        # this block will get executed if any exception encounters
from the above lists of exceptions.
        . . . . . .
else:
        # this block will get executed if no exception is found.
        . . . . . .
```

The ‘try - Finally’ Clause

Syntax:

```
try:
        # all operations are done within this block.
        . . . . . .
        # if any exception is encountered, this block may get
skipped.
finally:
         . . . . . .
        # this block will definitely be executed.
```