Unit - 3

PYTHON PROGRAMMING LANGUAGE

INTRODUCTION

- Python invented by **Guido Van Rossum**. In 1989 working in National Research Institute(NRI) at Netherland. Python officially released in Feb'20 1991.

- **Python** is a dynamic, high level, free open source and interpreted programming language. It supports object-oriented programming as well as procedural oriented programming.

  Ex: Compiled Languages
  Compiled languages are converted directly into machine code that the processor can execute. As a result, they tend to be faster and more efficient to execute than interpreted languages. They also give the developer more control over hardware aspects, like memory management and CPU usage .
  Ex: C, C++

- Interpreters run through a program line by line and execute each command. Interpreted languages were once significantly slower than compiled languages.
  Ex: PHP, Ruby, Python, and JavaScript.

- In Python, we don't need to declare the type of variable because it is a dynamically typed language.

For example, x = 10
Here, x is a variable can be anything like values type of string, int etc. But in python no need declare type variable.

Features in Python

There are many features in Python, some of which are discussed below –

1. **Easy to code:**
   Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, Javascript, Java, etc. It is very easy to code in python language and anybody can learn python basics in a few hours or days. It is also a developer-friendly language.

   Python having less code for that reason also Python become very popular.

The code required for C language, Java, Python as follows To print (view) "Hello World" as follows.

C language	Java	Python

#include<studio.h>
void main()
{
Print("Hello World");
}
	Public class Hello World
	{
	Public static void main(strings[ ]args);
	{
	System.out.println("Hello World");
	}
	}
		Print("Hello World")

As per above Python required very less code (only one statement) to print "Hello World"

2. **Free and Open Source:**
   Python language is freely available at the official website and you can download it from the given download link below click on the **Download Python** keyword.

3. **Interpreted Language**
   Python is an interpreted language. It comes with the **IDLE (Interactive Development Environment)**. This is an interpreter and follows the **REPL structure (Read-Evaluate-Print-Loop)**. It executes and displays the output of one line at a time.

4. **Object-Oriented Language:**
   Python supports object-oriented language and concepts of classes and objects come into existence. Python programming supports Abstraction, Encapsulation Inheritance and Polymorphism etc. The object-oriented procedure helps to programmer to write reusable code and develop applications in less code.
   i)Abstraction – To represent the essential features without represent background details.
   Ex: NOKIA 1400 features are calling and sms.

   ii)Encapsulation- It hides the implementation details of a class from other object.

   iii)Inheritance- A class acquire a property of another class is known as Inheritance.

   iv)Polymorphism- One function behaves different forms. Many form of a single object is called Polymorphism.

5. **GUI Programming Support:**
   Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in python.
   PyQt5 is the most popular option for creating graphical application with Python. PyQt is Python binding of cross-platform GUI took kit QT (i.e. Quality and Technology)

6. **High-Level Language:**
   Python is a high-level language. When we write programs in python, we do not need to remember the system architecture, nor do we need to manage the memory.

7. **Extensible feature:**
   Python is an **Extensible** language. We can write us some Python code into C or C++ language and also we can compile that code in C/C++ language.

8. **Python is Portable language:**
   Python language is also a portable language. For example, if we have python code for windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.

9. **Python is Integrated language:**
   If we have Python code in Windows, if we want run this code on other platforms like LINXU, UNIX etc. No need of change the code. Just run the code in any platform. Python is also an Integrated language because we can easily integrated python with other languages like C, C++, etc.

10. **Interpreted Language:**
    Python is an Interpreted Language because Python code is executed one line at a time. like other languages C, C++, Java, etc. there is no need to compile python code this

makes it easier to debug our code. The source code of python is converted into an immediate form called **bytecode**.

11. Large Standard Library:

- Python has a large standard library which provides a rich set of module and functions so you do not have to write your own code for every single thing. There are many libraries present in python for such as regular expressions, unit-testing, web browsers, etc.
- We can also install packages from the **PyPI(Python Package Index)** if you want even more functionality.

12. Dynamically Typed Language:

Python is a dynamically-typed language. That means the type (for example- int, double, long etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.

Ex: C and Java are statistically programming languages.

Ex: int a, b

```
int a=30, b=20;
```

Ex: Python is dynamically programming language.

Ex:
```
>>> a= 10, b=20
>>> print (a+b)
    30
>>> type(x)
    <class 'int'>
```

13. Platform-Independent:

Python is platform-independent. If you write a program, it will run on different platforms like **Windows**, **Mac** and **Linux**. You don't need to write them separately for each platform.

14. Embeddable:

The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.

15. Python developed by borrowing features from other languages like C, C++

- Functional programming features from C
- OOP (Object Oriented Programming) features from C++
- Scripting language features from PERL and SHELL Script
- Modular programming features from MODULAR-3

16. Python is a General Purpose High Level Language

Python code develop for almost all applications related to...

- Desktop application
- Database applications
- Machine Learning applications
- Artificial Intelligence(AI) applications
- Internet of Things(IoT) applications
- Robotic applications

Local Variables And Global Variables In Python

Local Variables

Local variables are declared within a function where variables have been declared.

Global Variables

These global variables are declared above the user defined function. These variables can use by entire program.

Program1: Python program to declare global variables and local variable

```
$nano localglobal.py

a=100
def display():        //user defined function
       b=50          //local variable
      print("Inside user defined function")
      print("local variable:",b)

display()            //function call main function
print("Outside the user defined function")
print("global variable:", a)
```

Output
$python3 localglobal.py

```
Inside user defined function
local variable: b
Outside the user defined function
global variable
```

Program2: Python program for global variables and local variable assign same variable name.

```
n=100
def glob():
    print("global variable")
    print("global variable",n)

def loc():
    n=50
    print("local variable")
    print("local variable",n)

loc()
glob()

output:
$python3 localglobal2.py
```

```
Local variable=50
Global variable=100
```

Program3: Python program to declare global variables and local variable with values

```
$nano localglobal3.py

g=100
def display():
    l=50
    Print("Inside user defined function: local variable:", l)

display()
Print("Outside user defined function: global variable:",g)

Output:
$python3 localglobal3.py

Inside user defined function: local variable:50
user defined function: global variable:100
```

Installation of Nano Editor

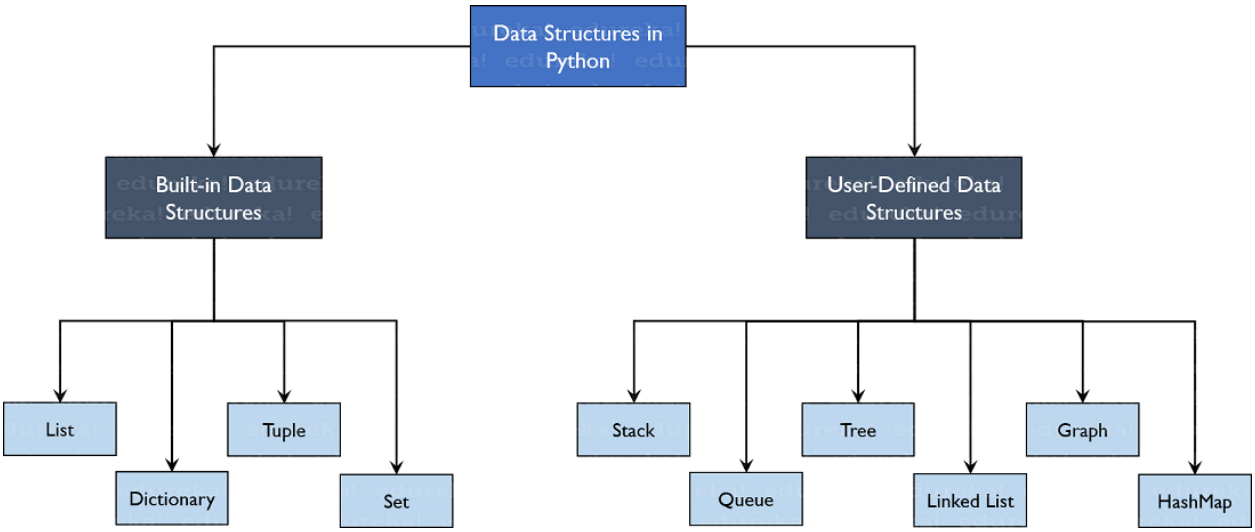
```
$sudo apt update
$sudo apt install nano
```

Data Structures in Python

data plays a very important role which means that this data should be stored efficiently and the access to it must be timely.

Data Structure

Organizing, managing and storing data is important as it enables easier access and efficient modifications. Data Structures allows you to organize your data in such a way that enables you to store collections of data, relate them and perform operations on them accordingly.



Built-in Data Structures
As the name suggests, these Data Structures are built-in with Python which makes programming easier and helps programmers use them to obtain solutions faster. Let's discuss each of them in detail.

Lists
Lists are used to store data of different data types in a sequential manner. There are addresses assigned to every element of the list, which is called as Index. The index value starts from 0 and goes on until the last element called the positive index. There is also negative indexing which starts from -1 enabling you to access elements from the last to first. Let us now understand lists better with the help of an example program.

Creating a list
To create a list, you use the square brackets and add elements into it accordingly. If you do not pass any elements inside the square brackets, you get an empty list as the output.

```
>>>my_list = [] #create empty list
>>>print(my_list)
>>>my_list = [1, 2, 3, 'example', 3.132] #creating list with data
>>>print(my_list)
```

Output:
[]
[1, 2, 3, 'example', 3.132]

Adding Elements
Adding the elements in the list can be achieved using the append(), extend() and insert() functions.

- The append() function adds all the elements passed to it as a single element.
- The extend() function adds the elements one-by-one into the list.
- The insert() function adds the element passed to the index value and increase the size of the list too.

```
>>>my_list = [1, 2, 3]
>>>print(my_list)
>>>my_list.append([555, 12]) #add as a single element
>>>print(my_list)
>>>my_list.extend([234, 'more_example']) #add as different elements
>>>print(my_list)
>>>my_list.insert(1, 'insert_example') #add element i
>>>print(my_list)
```

Output:
[1, 2, 3]
[1, 2, 3, [555, 12]]
[1, 2, 3, [555, 12], 234, 'more_example']
[1, 'insert_example', 2, 3, [555, 12], 234, 'more_example']

Deleting Elements
- To delete elements, use the del keyword which is built-in into Python but this does not return anything back to us.
- If you want the element back, you use the pop() function which takes the index value.
- To remove an element by its value, you use the remove() function.

```
>>>my_list = [1, 2, 3, 'example', 3.132, 10, 30]
>>>del my_list[5] #delete element at index 5
>>>print(my_list)
>>>my_list.remove('example') #remove element with value
>>>print(my_list)
>>>a = my_list.pop(1) #pop element from list
>>>print('Popped Element: ', a, ' List remaining: ', my_list)
>>>my_list.clear() #empty the list
>>>print(my_list)
```

Output:
[1, 2, 3, 'example', 3.132, 30]
[1, 2, 3, 3.132, 30]
Popped Element: 2 List remaining: [1, 3, 3.132, 30]
[]

Accessing Elements

Accessing elements is the same as accessing Strings in Python. You pass the index values and hence can obtain the values as needed.

```
>>>my_list = [1, 2, 3, 'example', 3.132, 10, 30]
for element in my_list: #access elements one by one
>>>print(element)
>>>print(my_list) #access all elements
>>>print(my_list[3]) #access index 3 element
>>>print(my_list[0:2]) #access elements from 0 to 1 and exclude 2
>>>print(my_list[::-1]) #access elements in reverse
```

Output:

1
2
3
example
3.132
10
30
[1, 2, 3, 'example', 3.132, 10, 30]
example
[1, 2]
[30, 10, 3.132, 'example', 3, 2, 1]

Other Functions

You have several other functions that can be used when working with lists.

- The **len()** function returns to us the length of the list.
- The **index()** function finds the index value of value passed where it has been encountered the first time.
- The **count()** function finds the count of the value passed to it.
- The **sorted()** and sort() functions do the same thing, that is to sort the values of the list. The **sorted()** has a return type whereas the sort() modifies the original list.

```
>>>my_list = [1, 2, 3, 10, 30, 10]
>>>print(len(my_list)) #find length of list
>>>print(my_list.index(10)) #find index of element that occurs first
>>>print(my_list.count(10)) #find count of the element
>>>print(sorted(my_list)) #print sorted list but not change original
>>>my_list.sort(reverse=True) #sort original list
>>>print(my_list)
```

Output:

6
3
2
[1, 2, 3, 10, 10, 30]
[30, 10, 10, 3, 2, 1]

Dictionary

Dictionaries are used to store key-value pairs.

Ex:phone directory where hundreds and thousands of names and their corresponding numbers have been added. Now the constant values here are Name and the Phone Numbers which are called as the keys. And the various names and phone numbers are the values that have been fed to the keys. If you access the values of the keys, you will obtain all the names and phone numbers. So that is what a key-value pair is.

And in Python, this structure is stored using Dictionaries. Let us understand this better with an example program.

Creating a Dictionary

Dictionaries can be created using the flower braces or using the dict() function. You need to add the key-value pairs whenever you work with dictionaries.

```
>>>my_dict = {} #empty dictionary
>>>print(my_dict)
>>>my_dict = {1: 'Python', 2: 'Java'} #dictionary with elements
>>>print(my_dict)
```

Output:

{}

{1: 'Python', 2: 'Java'}

Changing and Adding key, value pairs

To change the values of the dictionary, you need to do that using the keys. So, you firstly access the key and then change the value accordingly. To add values, you simply just add another key-value pair as shown below.

```
>>>my_dict = {'First': 'Python', 'Second': 'Java'}
>>>print(my_dict)
>>>my_dict['Second'] = 'C++' #changing element
>>>print(my_dict)
>>>my_dict['Third'] = 'Ruby' #adding key-value pair
>>>print(my_dict)
```

Output:

{'First': 'Python', 'Second': 'Java'}

{'First': 'Python', 'Second': 'C++'}

{'First': 'Python', 'Second': 'C++', 'Third': 'Ruby'}

Deleting key, value pairs

- To delete the values, you use the pop() function which returns the value that has been deleted.
- To retrieve the key-value pair, you use the popitem() function which returns a tuple of the key and value.
- To clear the entire dictionary, you use the clear() function.

```
>>>my_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}
>>>a = my_dict.pop('Third') #pop element
>>>print('Value:', a)
>>>print('Dictionary:', my_dict)
```

```
>>>b = my_dict.popitem() #pop the key-value pair
>>>print('Key, value pair:', b)
>>>print('Dictionary', my_dict)
>>>my_dict.clear() #empty dictionary
>>>print('n', my_dict)
```

Output:
Value: Ruby
Dictionary: {'First': 'Python', 'Second': 'Java'}
Key, value pair: ('Second', 'Java')
Dictionary {'First': 'Python'}
{}

Accessing Elements

You can access elements using the keys only. You can use either the get() function or just pass the key values and you will be retrieving the values.

```
>>>my_dict = {'First': 'Python', 'Second': 'Java'}
>>>print(my_dict['First']) #access elements using keys
>>>print(my_dict.get('Second'))
```

Output:
Python
Java

Other Functions

You have different functions which return to us the keys or the values of the key-value pair accordingly to the keys(), values(), items() functions accordingly.

```
>>>my_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}
>>>print(my_dict.keys()) #get keys
>>>print(my_dict.values()) #get values
>>>print(my_dict.items()) #get key-value pairs
>>>print(my_dict.get('First'))
```

Output:
dict_keys(['First', 'Second', 'Third'])
dict_values(['Python', 'Java', 'Ruby'])
dict_items([('First', 'Python'), ('Second', 'Java'), ('Third', 'Ruby')])
Python

Tuple

Tuples are the same as lists are with the exception that the data once entered into the tuple cannot be changed no matter what. The only exception is when the data inside the tuple is mutable, only then the tuple data can be changed. The example program will help you understand better.

Creating a Tuple

You create a tuple using parenthesis or using the tuple() function.

```
>>>my_tuple = (1, 2, 3) #create tuple
>>>print(my_tuple)
```

Output:
(1, 2, 3)

Accessing Elements
Accessing elements is the same as it is for accessing values in lists.
my_tuple2 = (1, 2, 3, 'edureka') #access elements for x in my_tuple2:

```
>>>print(x)
>>>print(my_tuple2)
>>>print(my_tuple2[0])
>>>print(my_tuple2[:])
>>>print(my_tuple2[3][4])
```

Output:
1
2
3
edureka
(1, 2, 3, 'edureka')
1
(1, 2, 3, 'edureka')

Appending Elements
To append the values, you use the '+' operator which will take another tuple to be appended to it.

```
>>>my_tuple = (1, 2, 3)
>>>my_tuple = my_tuple + (4, 5, 6) #add elements
>>>print(my_tuple)
```

Output:
(1, 2, 3, 4, 5, 6)

Other Functions
These functions are the same as they are for lists.

```
>>>my_tuple = (1, 2, 3, ['hindi', 'python'])
>>>my_tuple[3][0] = 'english'
>>>print(my_tuple)
>>>print(my_tuple.count(2))
>>>print(my_tuple.index(['english', 'python']))
```

Output:
(1, 2, 3, ['english', 'python'])
1
3

Sets
Sets are a collection of unordered elements that are unique. Meaning that even if the data is repeated more than one time, it would be entered into the set only once. It resembles the sets that you have learnt in arithmetic. The operations also are the same as is with the arithmetic sets. An example program would help you understand better.

Creating a set
Sets are created using the flower braces but instead of adding key-value pairs, you just pass values to it.

```
>>>my_set = {1, 2, 3, 4, 5, 5, 5} #create set
>>>print(my_set)
```

Output:
{1, 2, 3, 4, 5}

Adding elements
To add elements, you use the add() function and pass the value to it.

```
>>>my_set = {1, 2, 3}
>>>my_set.add(4) #add element to set
>>>print(my_set)
```

Output:
{1, 2, 3, 4}

Operations in sets
The different operations on set such as union, intersection and so on are shown below.

```
>>>my_set = {1, 2, 3, 4}
>>>my_set_2 = {3, 4, 5, 6}
>>>print(my_set.union(my_set_2), '----------', my_set | my_set_2)
>>>print(my_set.intersection(my_set_2), '----------', my_set & my_set_2)
>>>print(my_set.difference(my_set_2), '----------', my_set - my_set_2)
>>>print(my_set.symmetric_difference(my_set_2), '----------', my_set ^ my_set_2)
>>>my_set.clear()
>>>print(my_set)
```

- The union() function combines the data present in both sets.
- The intersection() function finds the data present in both sets only.
- The difference() function deletes the data present in both and outputs data present only in the set passed.
- The symmetric_difference() does the same as the difference() function but outputs the data which is remaining in both sets.

Output:
{1, 2, 3, 4, 5, 6} ———- {1, 2, 3, 4, 5, 6}
{3, 4} ———- {3, 4}
{1, 2} ———- {1, 2}
{1, 2, 5, 6} ———- {1, 2, 5, 6}
set()

User-defined Data Structures
User-defined Data Structures, the name itself suggests that users define how the Data Structure would work and define functions in it. This gives the user whole control over how the data needs to be saved, manipulated and so forth.

Let us move ahead and study the most prominent Data Structures in most of the programming languages.

Control Flow Statements in Python

Generally a program executes its statements from beginning to end, but not many program execute all their statements in strict order from beginning to end, programs, depending upon the need, can choose to execute one of the available alternative or even repeat a set of statements. Such statements are called **Program control statements**.

- A program's control flow is the order in which the program's code executes.
- The control flow of a Python program is regulated by conditional statements, loops, and function calls.
- Python has three types of control structures:
    - Sequential - default mode
    - Selection - used for decisions and branching
    - Repetition - used for looping, i.e., repeating a piece of code multiple times.

1. Sequential

Sequential statements are a set of statements whose **execution process happens in a sequence**. The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.

Program1

```
## This is a Sequential statement
a=20
b=10
print(a-b)
print(a+b)
print(a*b)
print(a%b)
```

Program2

```
a=20
b=10
c=a-b
print("Subtraction is : ",c)
c=a+b
print("Addition is : ",c)
c=a%b
print("Divide is : ",c)
c=a*b
print("Multiplication is : ",c)
```

Output

```
Subtraction is :  10
Addition is :  30
Divide is :  0
Multiplication is :  200
```

2. Selection/Decision control statements

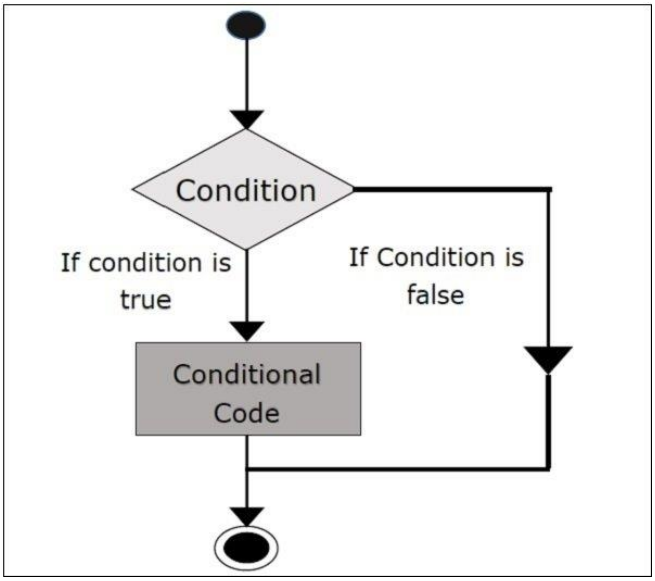
In Python, the selection statements are also known as **Decision control statements** or branching statements.

The selection statement allows a program to test several conditions and execute instructions based on which condition is true.

Some Decision Control Statements are:

- Simple if
- if-else
- nested if
- if-elif-else

Simple if: If statements are control flow statements that help us to run a particular code, but only when a certain condition is met or satisfied. A simple if only has one condition to check.



Program:

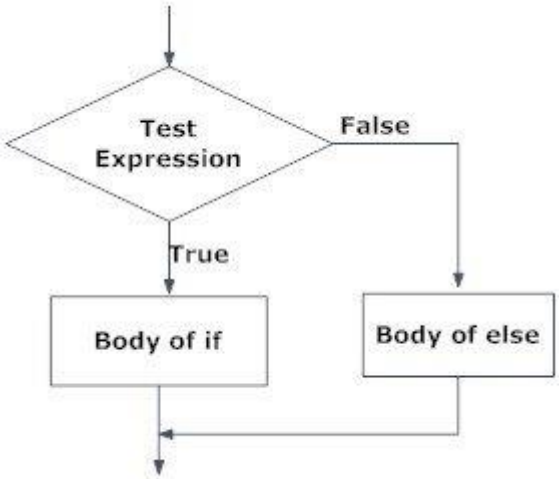
```
n = 10
if n % 2 == 0:
   print("n is an even number")
```

Output:

n is an even number

if-else

The if-else statement evaluates the condition and will execute the body of if the test condition is True, but if the condition is False, then the body of else is executed.



Program:

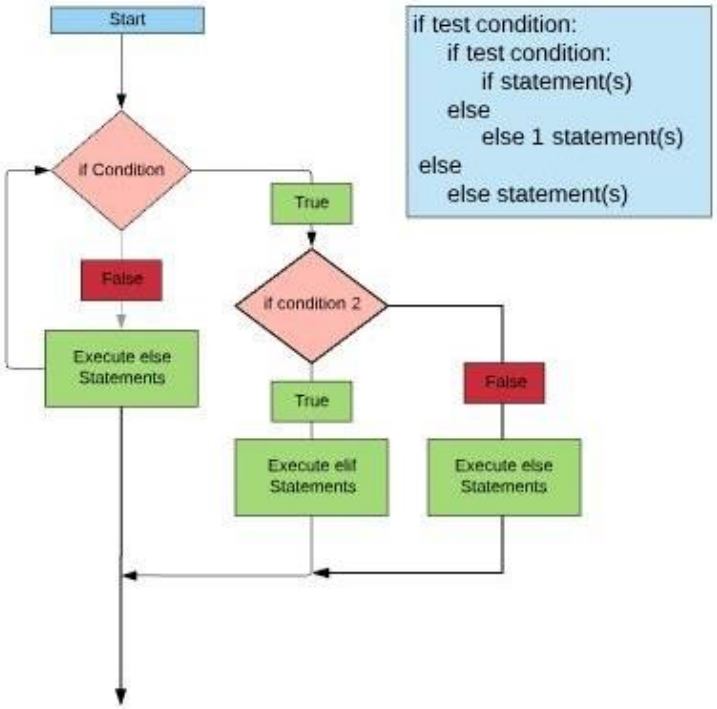
```
n = 5
if n % 2 == 0:
   print("n is even")
else:
   print("n is odd")
```

Output:

```
n is odd
```

nested if

Nested if statements are an if statement that are inside another if statement.



Program

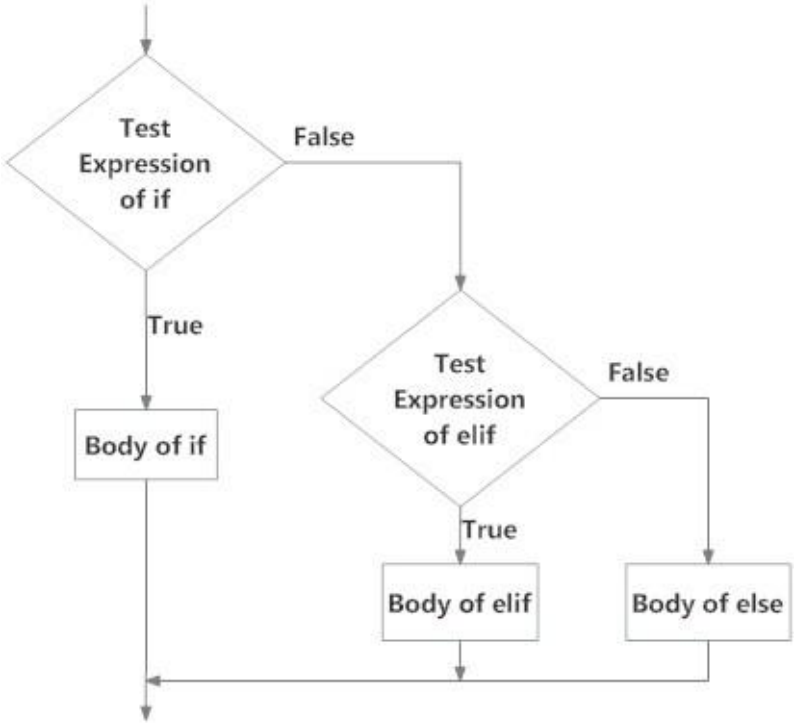
```
a = 5
b = 10
c = 15
if a > b:
   if a > c:
      print("a value is big")
   else:
       print("c value is big")
```

Output:

```
c is big
```

if-elif-else

The if-elif-else statement is used to conditionally execute a statement or a block of statements.



Program:

```
x = 15
y = 12
if x == y:
   print("Both are Equal")
elif x > y:
    print("x is greater than y")
else:
    print("x is smaller than y")
```

Output

x is greater than y

Program

```
x=5
if x==1:
   print("One")

elif(x==2):
   print("Two")

elif(x==3):
   print("Three")

elif(x==4):
   print("Four")

else:
   print("Wrong Input")
```

3. Repetition

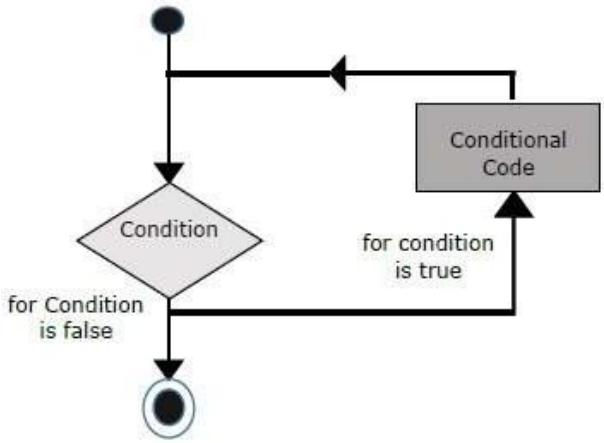
A repetition statement is used to repeat a group(block) of programming /instructions.

In Python, we generally have two loops/repetitive statements:

- for loop
- while loop

for loop

A for loop is used to iterate over a sequence that is either a list, tuple, dictionary, or a set. We can execute a set of statements once for each item in a list, tuple, or dictionary.



Program1

```
for i in range(1, 11):
    print(i)
```

Output
1
2
3
4
5
6
7
8
9
10

Program2

```
lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for i in range(0,5):
     print(lst[i])

for j in range(6,10):
    print(lst[j])
```

Output

```
0
1
2
3
4
6
7
8
9
```

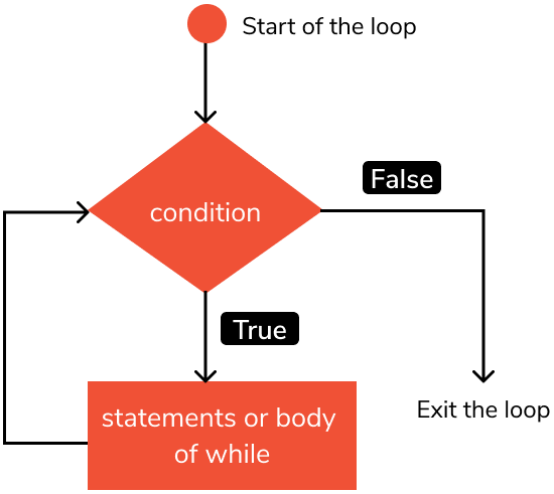
Program3

 # Python program to print Even Numbers in a List

```
# list of numbers
list1 = [10, 21, 4, 45, 66, 93]

# iterating each number in list
for num in list1:
    # checking condition
    if(num%2)==0:
        print(num,end=" ")    //or print("{0}",format(num))

Output:
10,4,66
```

While loop

In Python, while loops are used to **execute a block of statements repeatedly until a given condition is satisfied**. Then, the expression is checked again and, if it is still true, the body is executed again. This continues until the expression becomes false.



Program1

```
i = 1
while(i<=10):
    print(i)
    i += 1
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Program2

```
i = 1
while i <= 10:
    print(i)
    i += 1
else:
    print('Loop ended, i =', i)
```

output:

```
1
2
3
4
5
6
7
8
9
10
Loop ended, i = 11
```

```
# Python Program to Calculate Total Marks Percentage and Grade of a Student using if , else

print("Enter the marks of five subjects::")

subject_1 = float (input ())
subject_2 = float (input ())
subject_3 = float (input ())
subject_4 = float (input ())
subject_5 = float (input ())

total, average, percentage, grade = None, None, None, None

# It will calculate the Total, Average and Percentage
total = subject_1 + subject_2 + subject_3 + subject_4 + subject_5
average = total / 5.0
percentage = (total / 500.0) * 100

if average >= 90:
    grade = 'A'
elif average >= 80 and average < 90:
    grade = 'B'
elif average >= 70 and average < 80:
    grade = 'C'
elif average >= 60 and average < 70:
    grade = 'D'
else:
    grade = 'E'

# It will produce the final output
print ("\nThe Total marks is:   \t", total, "/ 500.00")
print ("\nThe Average marks is: \t", average)
print ("\nThe Percentage is:    \t", percentage, "%")
print ("\nThe Grade is:         \t", grade)
```

Python Variables and Data Types

Variables are memory locations where a user can store a value.

Ex:

```
x=100
Y="student"
Print(x)
Print(y)
```

If we want to perform arithmetic calculations on these variables.

Ex:

```
x=100
y=200
Print(x+y)
Print(x-y)
Print(x*y)
Print(x/y)
```

Variables declaration in python is case sensitive i.e. upper letter variables / lowercase variables are defined.

Variables data types are…
1)Number 2)Strings 3)List 4) Dictionary 5) Tuple 6) List

1.NUMBERS

Numbers are assigned with decimal point
Integer
x=10

float is assign with decimal point
float
x=10.5

complex
x=25j

Here J is an imaginary part and add to number.

Program1:

```
a=5
print(type(a))
print(type(5.0))
c=5+3j
print(c+3)
print(isinstance(c, complex))
```

Program2:

```
str_list="welcome to python"
age=50
Pi=3.14
c_num=3j+10
print("variable type is:",type(str_list))
print("variable type is:",type(age))
print("variable type is:",type(pi))
```

2)STRINGS

Strings are group of characteristics. We given strings with single / double codes
x='hello'
y="world"

Print("here is an example",x,y)'

>>>name = 'student'
>>>len(name)
>>>name[2]
'u'
>>>name[2:8]
'udent'
>>>name.lower()
Student
>>>name.upper()
'STUDENT'

3.LIST

Lists are collection of arrays. Ordered can be changed. Duplicate entries are present.

Ex: fruits =['apple','mango','banana']
>>>print(fruits)

Practical:

```
mylist=[10,20,30,30,'student','courses']

mylist
[10,20,30,30,'student','courses']

mylist[2:5]
[30,30,'student']

mylist
[10,20,35,30,'student','courses',[10]

mylist.append(10)
[10,20,35,30,'student','courses', 10]
```

4.DICTIONARY

It looks like a LIST but unordered can be changed. No duplicate entries are present

Ex:

```
Animals = {'reptiles':'snake',
           'mamals',: 'whale',
           'amphibians': 'frogs'}

>>>print(animals)
```

Practical:

```
>>>courses = {1:'Python',
              2:'Data science',
              3:'Learning'}

>>>courses
{1:'Python', 2:'Data science', 3:'machine learning'}

>>>courses[3]
'machine learning'

>>>courses['third']='hadoop'
>>>courses
{1:'python', 2:'data science', 'third': 'hadoop'}

In[]:courses['four']='machine learning'
>>>courses
→list courses
```

5. TUPLE

Tuple is ordered cannot be changed. It allow duplicate entries

Ex:

```
>>>animals = ('lion', 'tiger', 'monkey')
>>>print(animals)
```

Practical

```
>>>animals=(10,10,20,'tiger', 'lion', 'giraffee','tiger')
>>>animals
20 → shows
>>>animals.count('tiger')

2 → shows
```

6. SETS

Sets are un-ordered not allowed duplicate entries

Ex:

```
>>>animals={'lion', 'monkey', 'snake'}
```

```
>>>print(animals)
```

Practical

```
>>>myset={10,20,30,40,40,30,'student', 'courses'}

>>>myset
{10,20,30,40,'courses','student'}

>>>myset
```

➔ Shows error because SET object does not support for indexing

7.RANGE

Used to iterating with values

```
>>>range(5)
Range(0.5)

>>>list(range(5)
[0,1,2,4]

>>>range(5)
>>>(0,5)
>>>x=range(6)
      For n in x:
         Print(n)
```

→enter 2 times

Output:

1
2
3
4
5

8. TYPE CONVERSION

```
>>>X=(20)
>>>y=("name is")
>>>Z=20
>>>Print(str(x*y),y)
```

Type conversion is a converter of a type of variable to another data type

Ex:

```
>>>str(x) +name
'10name'
```

→ shows

FUNCTIONS IN PYTHON

Introduction

The main use of functions is 'Reusability' i.e. if we write functions, we call any number of times for this particular code.

Debugging is easy – To find errors and removing errors easy by writing these functions.

Python (function have)	**C (function have)**
Function definition	function declaration
Functional call	Function call
	Function definition

Function Definition

1.Every function start with '**def**' keyword followed by function_name, parameters
2.Every function(user defined function) should have a name (not equal to any keyword)
3.Parameters / arguments are inputs given to particular user defined functions. The parameters are optional. The parameters are included in parenthesis()
4.Every function name with / without value/empty end with '**:**' to achieve indentation.
5. Every function should written with value/ empty written by calling function.
6.In generally a function written a single value but in python can written multiple values. (these multiple values can be done using **tuple**)

Program1: Add Two numbers

```
$nano add.py
```

```
def add_num(a,b):                          # function definition
   sum=a+b
   return sum                              #return value
num1=24.50                                 #variable declaration
num2=55.54
print("The sum is", add_num(num1,num2))  #call to function
```

```
$python3 add.py
```

Program2: Add Two numbers (given values to the variables by user)

```
def add_num(a,b):                          # function for addition
   sum=a+b
   return sum                              #return value
num1=float(input("input the number one: ")) #input from user for num1
num2=float(input("input the number one: ")) #input from user for num2



print("The sum is",add_num(num1,num2)) #call to function
```

USER DEFINED FUNCTIONS IN PYTHON

Program1:

```
def greet():                    #function definition
                        step1
Step2                   print("Hello")
                        print("How do you do?")

Step3  greet()                  #function call
```

Greet is function name with empty parenthesis followed by colon

Bring function into action we should call it.

Control goes back to function call and execute remaining statements. Once defined a function we can call it n number of times.

Program2:

```
def greet(name):            #add a variable name to the function.
    print("Hello", name)

greet("jack")               #inside greet function used string as
                            a jack. Jack is an argument.
```

Explanation:

step1: when we call the greet function with argument jack
step2: this is passed name variable in side the function definition
step3: then statements inside the function are executed
step4: we use name parameter inside the body of the function
step5: the control back to the function call →greet("jack") to execute next statements after it.
Step6: but no statements here in the above progra2, after function call.

Program3: How to pass multiple arguments

```
def add_numbers(n1,n2):                    #function definition with 2
                                             parameters n1,n2
    result=n1+n2
    print("The sum is", result)

number1=5.4                                #outside function we declare
                                           number1, number2
number2=6.7

add_numbers(number1,number2)               #we are passing number1,number2
                                           as arguments to add_numbers
                                           function.
```

Output:
The sum is 12.100000001

Explanation:

- In this program we have passed number1, number2 as arguments to the add_numbers function.
- These arguments(number1,number2) are excepted as n1,n2 once they are passed add_numbers function.
- Inside the function add_numbers n1=5.4 and n2=6.7
- We add the numbers and result printed in side the function.

Note: In this program we have to get 12.1 exact figure. Because of floating-point representation error in
this program.

Program4: Return a value from Function

```
def add_numbers(n1,n2):        #function definition
    result = n1+n2
    return result

number1 = 5.4
number2 = 6.7
result = add_numbers(number1,number2)  #function call
print("The sum is", result)
```

Explanation:

- First step call add_numbers function with two arguments number1, number2 are expected by function definition as n1,n2.
- The sum of n1 and n2 are calculated.
- The result is return to the function call.
- The result value is assign to result variable.
- The result is printed outside the add_numbers function (line 1).

Program 5: To find length of marks (using built in functions)

```
marks = [55,64,75,80,34]

length =len(marks)
print("Length is", length)
```

```
Output:
Length is 5
```

Program 6: To print total marks

```
marks = [55,64,75,80,34]

length = len(marks)
print("length is", length)

marks_sum = sum(marks)
print("The total marks you got is", marks_sum)
```

```
Output:
Length is 5
The total marks you got is 308
```

Program 7: To find average marks

```
#function to find average marks

def find_average_marks(marks):  #function definition
    sum_of_marks = sum(marks)
    total_subjects = len(marks)
    average_marks = sum_of_marks / total_subjects
    return average_marks

marks = [85,75,80,90,55]
average_marks = find_average_marks(marks)
print("Your average marks is:", average_marks)
```

```
Output:
Your average marks is: 77.0
```

Program 8: To find Average Marks with Grade

```
#function to find average marks

def find_average_marks(marks):      #function definition
    sum_of_marks = sum(marks)
    total_subjects = len(marks)
    average_marks = sum_of_marks / total_subjects
    return average_marks

#calculate the grade and return it
def compute_grade(average_marks):
    if average_marks >= 80:
        grade = 'A'
    elif average_marks >= 60:
        grade = 'B'
    elif average_marks>= 50:
        grade = 'C'
    else:
        grade = 'F'
    return grade


marks = [55,64,75,80,65]
average_marks = find_average_marks(marks)
print("Your average marks is:", average_marks)

grade = compute_grade(average_marks)
print("Your grade is:", grade)

Output:
Your average marks is: 67.8
Your grade is: B
```

Recursion in Function

- Function calling itself is called as **recursion.**
- When function calling itself then it will going infinite recursive ie. There is no end.
- So every recursion function should terminate after some final recursive calls.
- Base Case is for terminating from recursion
- The function should terminate to stop it recursive calls. Recursive calls are calling itself.

Program1

```
def greet():          #function definition
    print("Hello")
    greet()

greet()               #function call
```

Explanation: The function call it self and give output infinitely.

Ex:- Factorial

```
5! = 5 * 4!
     5 * 4 * 3!
     5 * 4 * 3 * 2!
     5 * 4 * 3 * 2 * 1!
```

Program to print factorial of a given number

```
def factorial(n):
    if(n==1):
       return 1

    else:
       Return n*factorial(n-1)   //step2

N=int(input("enter n value"))
Result=factorial(n)              //step1
Print(result)
```

n=5
res=factorial(5)

step2		step1
5*24=120	<-	5*factorial(4)
4*6=24	<-	4*factorial(3)
2*3=6	<-	3*factorial(2)
2*1=2	<-	2*factorial(1)

Output:

```
$python3 fact.py
Enter a value: 5
Factorial of 5 is 120
```

Program to print factorial of a given number

```
def factorial(n):

    return 1 if (n==1 or n==0) else n * factorial(n - 1) #step2

num=int(input("Enter a value"))       #step1
print ("Factorial of a number is:",factorial(n)) #step3


Output:
Enter a value4
Factorial of 4 is (4x3x2x1)= 24
```

EXCEPTION HANDLING IN PYTHON

Exception is…

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block (true block of statements execute). After the try: block, include an **except:** statement(false block statements execute), followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of *try....except...else* blocks –

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Program1: Cant divide with zero - Exception handling

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("the value",c)    #true answer
except:
    print("Can't divide with zero")  #false answer
```

Program2(i): File not found – Exception Handling

```
try:
    #this will throw an exception if the file doesn't exist.
    fileptr = open("file.txt","r")
except IOError:
    print("File not found")
else:
    print("The file opened successfully")
    fileptr.close()
```

```
Output:
File not found
```

Program2(ii): File not found – Exception Handling

```
try:
    #this will throw an exception if the file doesn't exist.
    fileptr = open("file.txt","r")
    print("The file opened successfully")
    fileptr.close()
except IOError:
    print("File not found")
```

Program2(iii): File not found – Exception Handling

```
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print("Error: can\'t find file or read data")
else:
   print("Written content in the file successfully")
```

Program3: Arithmetic Expression – Exception Handling

```
try:
    a=10/0
except(ArithmeticError, IOError):
    print("Arithmetic Exception")
else:
    print("Successfully Done")
```

Program4: Age valid for Vote – Exception Handling

```
try:
    age = int(input("Enter the age for Vote:"))
    if(age<18):
        raise ValueError
    else:
        print("the age is valid for Vote")
except ValueError:
    print("The age is not valid for Vote")
```

Program5: File Handling – Exception arise when we reading a file if not created.

```
filename = 'John.txt'
try:
    with open(filename) as f_obj:
        contents = f_obj.read()
except FileNotFoundError:
    msg = "Sorry, the file "+ filename + "does not exist."
    print(msg) # Sorry, the file John.txt does not exist.
```

FILE HANDLING IN PYTHON

Opening a file To open a file in Python, we use the open() function. The syntax of open() is as follows:

```
file_object= open(file_name, access_mode)
```

This function returns a file object called file handle which is stored in the variable file_object. We can use this variable to transfer data to and from the file (read and write) by calling the functions defined in the Python's IO module. If the file does not exist, the above statement creates a new empty file and assigns it the name we specify in the statement.

The file_object has certain attributes that tells us basic information about the file, such as:

<file.closed> returns true if the file is closed and false otherwise.

<file.mode> returns the access mode in which the file was opened.

<file.name> returns the name of the file.

The writelines() method

This method is used to write multiple strings to a file. We need to pass an enterable object like lists, tuple, etc. Containing strings to the writelines() method. Unlike write(), the writelines() method does not return the number of characters written in the file. The following code explains the use of writelines().

```
>>> myobject=open("myfile.txt",'w')

>>> lines=['Hellow everyone\n", "Writing I am \n", "This is
third line"]

>>> myobject=open("myfile.txt",'r')

>>> print(myobject.read())

>>> myobject.close()
```

The read() method

This method is used to read a specified number of bytes of data from a data file. The syntax of read() method is:

```
>>>file_object.read(n)
```

Consider the following set of statements to understand the usage of read() method:

```
>>>myobject=open("myfile.txt",'r')

>>> myobject.read(10)

'Hello ever'
```

```
>>> myobject.close()
```

If no argument or a negative number is specified in read(), the entire file content is read. For example,

```
>>> myobject=open("myfile.txt",'r')

>>> print(myobject.read())

Hello everyone
Writing multiline strings
This is the third line


>>> myobject.close()
```

Program: To Create a text file and Write data and Read it.

```
fobject=open("testfile.txt","w") # creating a data file

sentence=input("Enter the contents to be written in the file: ")

fobject.write(sentence) # Writing data to the file

fobject.close() # Closing a file

print("Now reading the contents of the file: ")

fobject=open("testfile.txt","r")

#looping over the file object to read the file

for str in fobject:

 print(str)

fobject.close()
```

Output

```
Enter the contents to be written in the file: Learning python makes us funny

Now reading the contents of the file:

Learning python makes us funny.
```

2-3 To create a text file and write data in it

program to create a text file and add data

```
fileobject=open("practice.txt","w+")
while True:
 data= input("Enter data to save in the text file: ")
 fileobject.write(data)
 ans=input("Do you wish to enter more data?(y/n): ")
 if ans=='n': break
fileobject.close()
```

Output:

```
Enter data to save in the text file: everyone must
Do you wish to enter more data?(y/n): y
Enter data to save in the text file: learn python
Do you wish to enter more data?(y/n): n
Everyone must learn python
```

EXCEPTION HANDLING PYTHON PACKAGES

Python With JSON

- JSON (JavaScript Object Notation) is a popular data format used for representing structured data.
- It's common to transmit and receive data between a server and web application in JSON format.
- JSON is a data exchange format similar to XML

Address Book record in JSON format

```
{
    "name": "tom"
    "address": "1 green street, NY",
    "phone": "123456"
}
```

Address Book record in XML format

```
<name>tome</name>
<address>1 green street, NY</address>
<phone>123456</phone>
```

Note:- In XML format takes large volume of data. Not as library as JSON. JSON lightweight format compared with XML.

1.Program to Convert from Python (dictionary) to JSON (string)

```
import json

# a Python object (dict):
x = {
  "name": "John",
  "age": 30,
  "city": "New York"
}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
import json
print(y)
```

Output:

```
{"name": "John", "age": 30, "city": "New York"}
```

Program1: Exception handling when entry of in-corrected data format

```
incorrect_json = '{ name":"John "age":30 "car:"None" }'
try:
     a_json = json.loads(incorrect_json)
     print(a_json)
except json.decoder.JSONDecodeError:
     print("String could not be converted to JSON")
```

Output

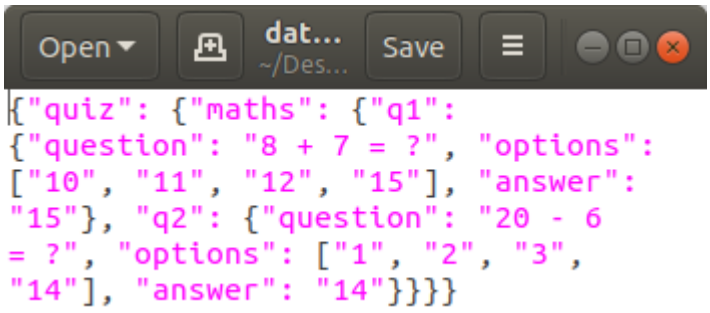
```
String could not be converted to JSON
```

XML with PYTHON

- XML stands for **eXtensible Markup Language**. It was designed to store and transport data. It was designed to be both human- and machine-readable.
- That's why, the design goals of XML emphasize simplicity, generality, and usability across the Internet.
- XML is a markup language which is designed to store data. It is case sensitive.
- XML offers you to define markup elements and generate customized markup language.
- The basic unit in the XML is known as an element. The XML language has no predefined tags. It simplifies data sharing, data transport, platform changes, data availability
- Extension of an XML file is .xml

Program to convert an XML file to JSON file

```
# import json module and xmltodict
# module provided by python
import json
import xmltodict

# open the input xml file and read
# data in form of python dictionary
# using xmltodict module
with open("test.xml") as xml_file:

    data_dict = xmltodict.parse(xml_file.read())
    xml_file.close()

    # generate the object using json.dumps()
    # corresponding to json data

    json_data = json.dumps(data_dict)

    # Write the json data to output
    # json file
    with open("data.json", "w") as json_file:
        json_file.write(json_data)
        json_file.close()
```



PYTHON WITH HTTPLIB2

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.

Python httplib2 module provides methods for accessing Web resources via HTTP. It supports many features, such as HTTP and HTTPS, authentication, caching, redirects, and compression.

Use httplib2 to Read Web Page
In the following example we show how to grab HTML content from a website called www.something.com.

```
#!/usr/bin/python3

import httplib2

http = httplib2.Http()
content = http.request("http://www.something.com")

print(content.decode())
```

An HTTP client is created with httplib2.HTTP(). A new HTTP request is created with the request() method; by default, it is a GET request. The return value is a tuple of response and content.

```
$ ./get_content.py
<html><head><title>Something.</title></head>
<body>Something.</body>
</html>
```

This is the output of the example

PYTHON WITH URLLIB.

- URLLIB is a Python module that can be used for opening URLs. It defines functions and classes to help in URL actions.

- With Python you can also access and retrieve data from the internet like XML, HTML, JSON, etc. You can also use Python to work with this data directly.

- URLLIB package is the URL handling module for python. It is used to fetch URLs (Uniform Resource Locators). It uses the urlopen function and is able to fetch URLs using a variety of different protocols.

URLIB function

- ✓ urllib.request for opening and reading.
- ✓ urllib.parse for parsing URLs
- ✓ urllib.error for the exceptions raised
- ✓ urllib.robotparser for parsing robot.txt files

urllib.request

This module helps to define functions and classes to open URLs (mostly HTTP). One of the most simple ways to open such URLs is :
urllib.request.urlopen(url)
We can see this in an example:

Program1 – used to make requests to URL to open page

```
#Used to make requests
import urllib.request
request_url = urllib.request.urlopen('https://
vaagdevi.edu.in/')
print(request_url.read())
```

Output:

```
The source code of the URL i.e. vaagdevi college of
engineering
```

.

Program for Exception handling Python with URLLIB

```
try:
    x =
urllib.request.urlopen('https://www.google.com/search?q=test')
    #print(x.read())

    saveFile = open('noheaders.txt','w')
    saveFile.write(str(x.read()))
    saveFile.close()

except Exception as e:
    print(str(e))
```

```
Output:
name 'urllib' is not defined
```

PYTHON WITH SMTPLib

Simple Mail Transfer Protocol (SMTP) is used as a protocol to handle the email transfer using Python. It is used to route emails between email servers. It is an application layer protocol which allows to users to send mail to another. The receiver retrieves email using the protocols **POP(Post Office Protocol)** and **IMAP(Internet Message Access Protocol)**. When the server listens for the TCP connection from a client, it initiates a connection on port 587.

Python provides **smtplib** module, which defines an SMTP client session object that can be used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. Python provides smptplib which is a library used to send mail to any Internet machine with an SMTP or ESMTP listener daemon. As smtplib is a part of the python standard library. Python provides a library named smtplib which can be used to connect to the mail server to send mail.

Here is the detail of the parameters −

host − This is the host running your SMTP server. You can specify IP address of the host or a domain name like tutorialspoint.com. This is optional argument.

port − If you are providing host argument, then you need to specify a port, where SMTP server is listening. Usually this port would be 25.

local_hostname − If your SMTP server is running on your local machine, then you can specify just localhost as of this option.

Python provides a smtplib module, which defines an the SMTP client session object used to send emails to an internet machine. For this purpose, we have to import the smtplib module using the import statement.

>>> **import smtplib**

Program1: Exception Handling by Send mail method – Python-SMTPLib

```
#!/usr/bin/python3

import smtplib

sender_mail = 'sender@fromdomain.com'

receivers_mail = ['reciever@todomain.com']

message = """From: From Person %s

To: To Person %s

Subject: Sending SMTP e-mail

This is a test e-mail message.

"""%(sender_mail,receivers_mail)
```

```
try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender_mail, receivers_mail, message)
   print("Successfully sent email")
except Exception:
   print("Error: unable to send email")
```

Output

Error: unable to send email

***