

PYTHON

- ✓ **Python** is object-oriented, interpreted, dynamic & widely used high-level programming language for general-purpose programming.
- ✓ **Python** created by **Guido van Rossum** and first released in 1991 from python software foundation.
- ✓ **Python** is platform independent & open source language.
- ✓ File name extensions in python : .py
- ✓ Web site : www.python.org
- ✓ Python is a clear and powerful object-oriented programming language, comparable to Perl, Ruby, Scheme, or Java.
- ✓ Python is used to develop the different types of application such as web-apps, stand alone apps, enterprise apps, ERP and e-commerce application, scientific & numeric computing..etc.

What is Python (Programming)?

Python is a general-purpose language. It has wide range of applications from Web development (like: Django and Bottle), scientific and mathematical computing (Orange, SymPy, NumPy) to desktop graphical user Interfaces (Pygame, Panda3D).

History of Python

Python is a fairly old language created by Guido Van Rossum. The design began in the late 1980s and was first released in February 1991.

Why Python was created?

In late 1980s, Guido Van Rossum was working on the Amoeba distributed operating system group. He wanted to use an interpreted language like ABC (ABC has simple easy-to-understand syntax) that could access the Amoeba system calls. So, he decided to create a language that was extensible. This led to design of a new language which was later named Python.

Why the name Python?

No. It wasn't named after a dangerous snake. Rossum was fan of a comedy series from late seventies. The name "Python" was adopted from the same series "Monty Python's Flying Circus". Python was named for the BBC TV show Monty Python's Flying Circus.

Java vs. python:

1. Python simple language :

I will write the code to print Ratan world on screen in C, Java, Python. Decide yourself which is simple.

Case 1: printing Hello world.....

In java

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("ratan world");
    }
}
```

In python

```
Print ("ratan world")
```

case 2: Taking input from end-user.....

In java :

```
Scanner s = new Scanner(System.in);
System.out.println("etner a number");
int n = s.nextInt();
System.out.println(n);
```

In python:

```
n = input("enter a number")
print(n)
```

case 3: swapping two numbers.

In java :

```
int a=10,b=20;
int temp;
temp=a;
a=b;
b=temp;
```

In python :

```
a,b=10,20
a,b=b,a
```

2. Python is dynamically typed.

In java we must declare the type of the variables by using data type concept. But python is dynamically typed no need to declare the data type.

In java: byte short int long float double char Boolean (Type declaration mandatory)

```
int eid=111;
String ename="ratan";
float esal=100000.34;
```

In python: Type declaration not required because it is dynamically typed.

```
a=10          by default it is : int
b=10.5        by default it is : float
c=True        by default it is : boolean
d="ratan"     by default it is : String
```

3. Importance of python & it is a general purpose language: build anything

Many big names such as Yahoo, IBM, Nokia, Google, Disney, NASA, Mozilla and much more rely on Python.

Python is used to develop the different types of application such as web-apps, stand alone apps, enterprise apps, ERP and e-commerce application, scientific & numeric computing..etc.

4. Python Single line code

In other languages it will take more lines of code but in python we can write the code in less number of line.

Case 1 : variable declaration single line of code in python.

In java :

```
int eid=111;
String ename="ratan";
float esal=100000.34;
```

in python :

```
eid,ename,esal=111,"ratan",100000.34
```

case 2: swaping two variables

In java :

```
int a=10;
int b=20;
int temp;
temp=a;
a=b;
b=temp;
```

In python :

```
a,b=10,20
a,b=b,a
```

5. Python platform independent & open source software

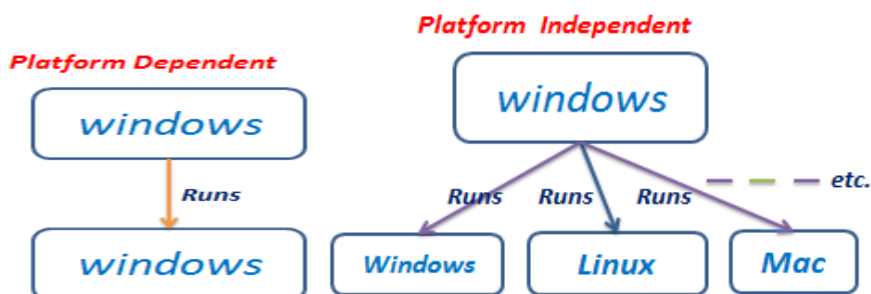
- ✓ Once we develop the application by using any one operating system(windows) that application runs only on same operating system is called platform dependency.

ex : C,CPP

- ✓ Once we develop the application by using any one operating system(windows) that application runs on in all operating system is called platform independency.

ex : python

- ✓ Free of cost & source code is open.



Python Version release dates:

- *Implementation started - December, 1989*
- *Internal releases at [Centrum Wiskunde & Informatica](#) – 1990*

- **Python 0.9.0 - February 20, 1991**
 - *Python 0.9.1 - February, 1991*
 - *Python 0.9.2 - Autumn, 1991*
 - *Python 0.9.4 - December 24, 1991*
 - *Python 0.9.5 - January 2, 1992*
 - *Python 0.9.6 - April 6, 1992*
 - *Python 0.9.8 - January 9, 1993*
 - *Python 0.9.9 - July 29, 1993*

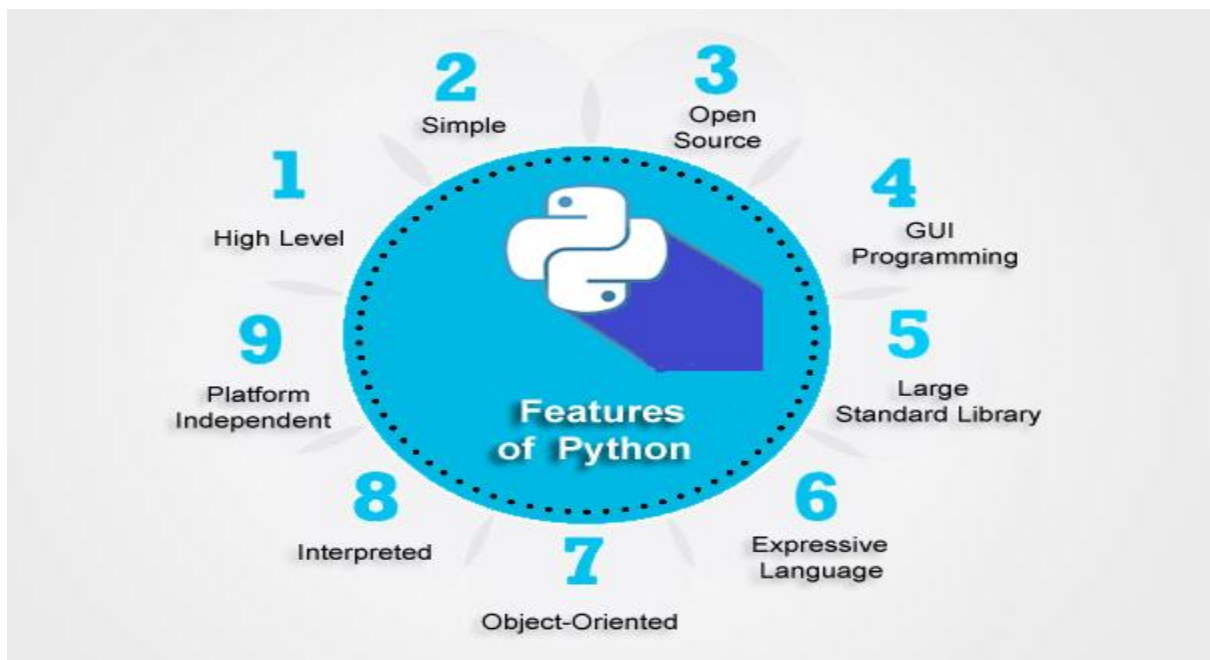
- **Python 1.0 - January 1994**
 - *Python 1.2 - April 10, 1995*
 - *Python 1.3 - October 12, 1995*
 - *Python 1.4 - October 25, 1996*
 - *Python 1.5 - December 31, 1997*
 - *Python 1.6 - September 5, 2000*

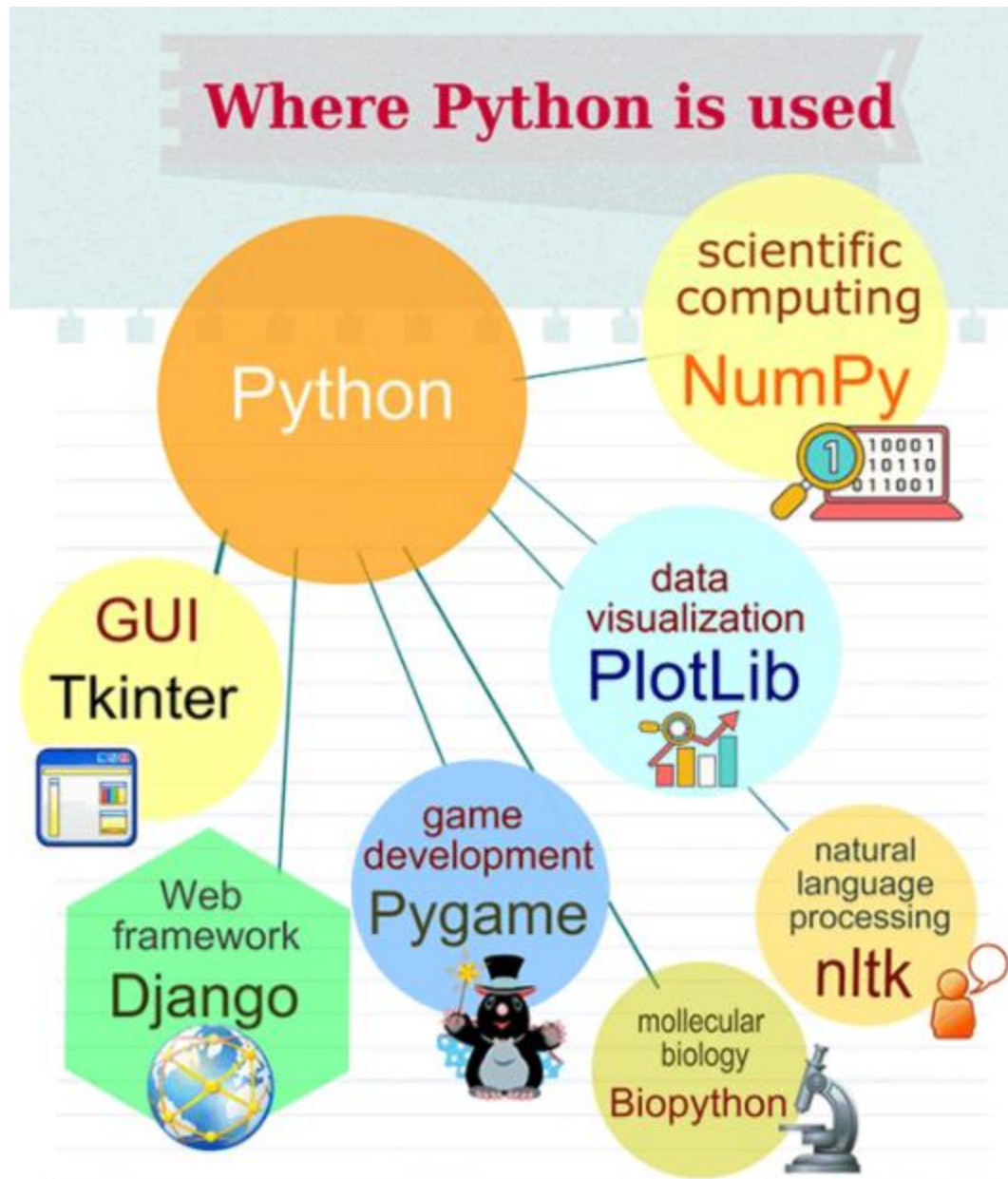
- **Python 2.0 - October 16, 2000**
 - *Python 2.1 - April 15, 2001*
 - *Python 2.2 - December 21, 2001*
 - *Python 2.3 - July 29, 2003*
 - *Python 2.4 - November 30, 2004*
 - *Python 2.5 - September 19, 2006*
 - *Python 2.6 - October 1, 2008*
 - *Python 2.7 - July 4, 2010*

- **Python 3.0 - December 3, 2008**
 - *Python 3.1 - June 27, 2009*
 - *Python 3.2 - February 20, 2011*
 - *Python 3.3 - September 29, 2012*
 - *Python 3.4 - March 16, 2014*
 - *Python 3.5 - September 13, 2015*
 - *Python 3.6 - December 23, 2016*
 - *Python 3.7 - June 27, 2018*

Features of python:

- ✓ **Easy to Use**
 - It is a programmer friendly; it contains syntaxes just like English commands.
 - Uses an elegant syntax, making the programs you write easier to read.
- ✓ **High Level Language**
 - Python is a clear and powerful object-oriented programming language, comparable to Perl, Ruby, Scheme, or Java.
- ✓ **Expressive Language**
 - The code is easily understandable.
- ✓ **Interpreted**
 - The execution done in line by line format.
- ✓ **Platform Independent**
 - We can run this python code in all operating systems.
- ✓ **Open Source**
 - Python is free of cost, source code also available.
- ✓ **Object-Oriented language**
 - Python supports object-oriented programming with classes and multiple inheritance
- ✓ **Huge Standard Library**
 - Code can be grouped into modules and packages.
- ✓ **GUI Programming**
 - Graphical user interfaces can be developed using Python.
- ✓ **Integrated**
 - It can be easily integrated with languages like C, C++, JAVA etc.
- ✓ **Extensible**
 - Is easily extended by adding new modules implemented in a compiled language such as C or C++.





Web Technology uses **PYTHON** with **django**

Mobile Apps uses **PYTHON** with **FLASK**

Big Data Hadoop uses **PYTHON**

Data Science Analytics uses **PYTHON**

Amazon Web Services uses **PYTHON**

Automation Tools DevOps uses **PYTHON**

Reporting Tableau uses **PYTHON**

Scripting and GUI Apps Uses **PYTHON**

Scientific Applications uses **PYTHON** with **Numpy, Scipy**

Gaming, Networking, Embedded Systems, AI

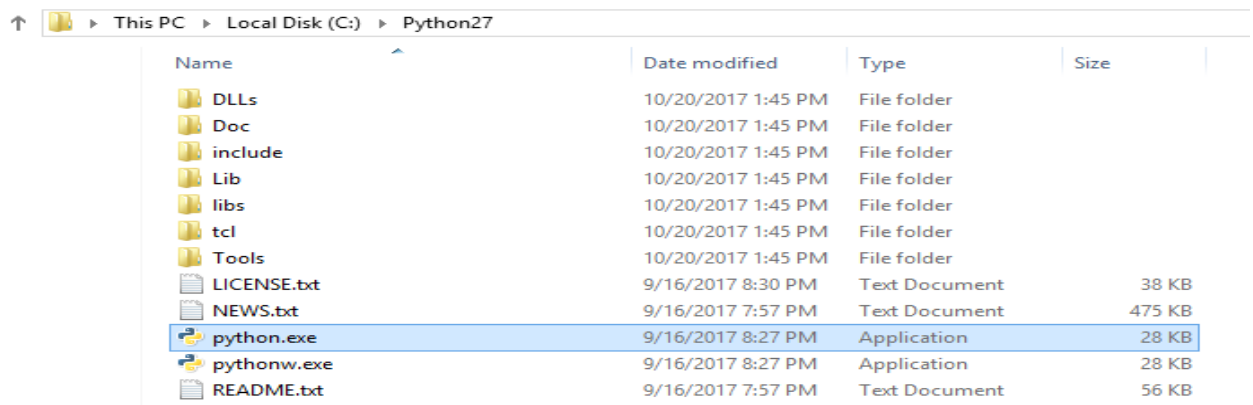
Python has significant advantages in terms of

- ✓ Pyramid, Flask, Bottle and Django are among the well known frameworks of Python.
- ✓ Pyramid is a small framework which makes real-world Web applications productive.
- ✓ Django is more mature and is one among the largest Web-based frameworks for Python But Django is very hard to customise and troubleshoot, in some situations, due to its all-encompassing nature. This is one of its drawbacks.
- ✓ Tornado is lighter in weight and has a few more features than Django. As I said earlier, Tornado is known for its high performance
- ✓ Building API services for Mobile (Flask)
- ✓ Doing Scientific computations (Numpy, Scipy)
- ✓ PlotLib would enable you to create data visualizations.
- ✓ Dealing with Data (Pandas)
- ✓ Numerous other things like scripting/ text processing/ machine learning/Natural Language Processing/ Text mining/ Web mining/ OpinionMining/ Sentiment analysis/ Big data system



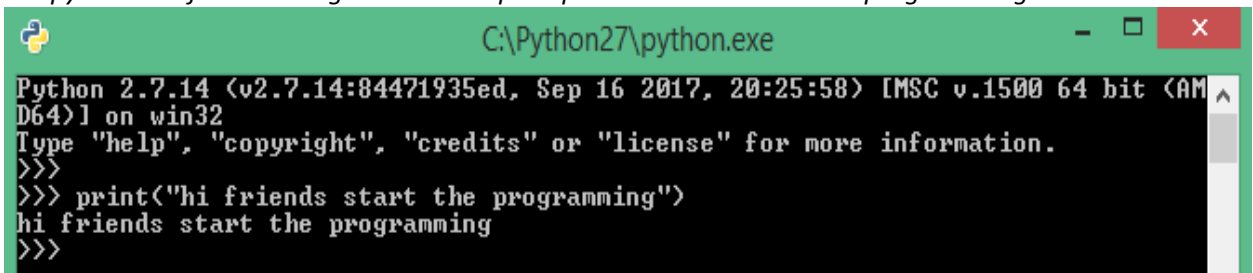
There are six different ways to run the python application

Approach- 1: By using python command line



Name	Date modified	Type	Size
DLLs	10/20/2017 1:45 PM	File folder	
Doc	10/20/2017 1:45 PM	File folder	
include	10/20/2017 1:45 PM	File folder	
Lib	10/20/2017 1:45 PM	File folder	
libs	10/20/2017 1:45 PM	File folder	
tcl	10/20/2017 1:45 PM	File folder	
Tools	10/20/2017 1:45 PM	File folder	
LICENSE.txt	9/16/2017 8:30 PM	Text Document	38 KB
NEWS.txt	9/16/2017 7:57 PM	Text Document	475 KB
python.exe	9/16/2017 8:27 PM	Application	28 KB
pythonw.exe	9/16/2017 8:27 PM	Application	28 KB
README.txt	9/16/2017 7:57 PM	Text Document	56 KB

Just click on python.exe file we will get command prompt shown below start the programming.

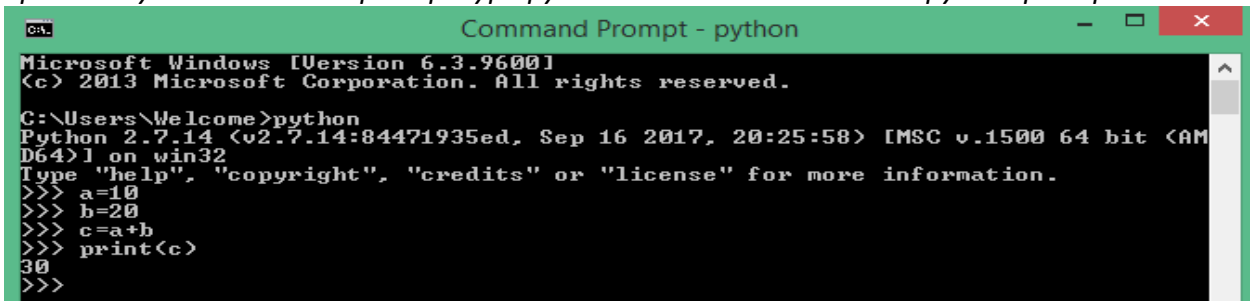


```

Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:25:58) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> print("hi friends start the programming")
hi friends start the programming
>>>
  
```

Approach-2: By using system command prompt

Open the system command prompt type python command then we will python prompt.

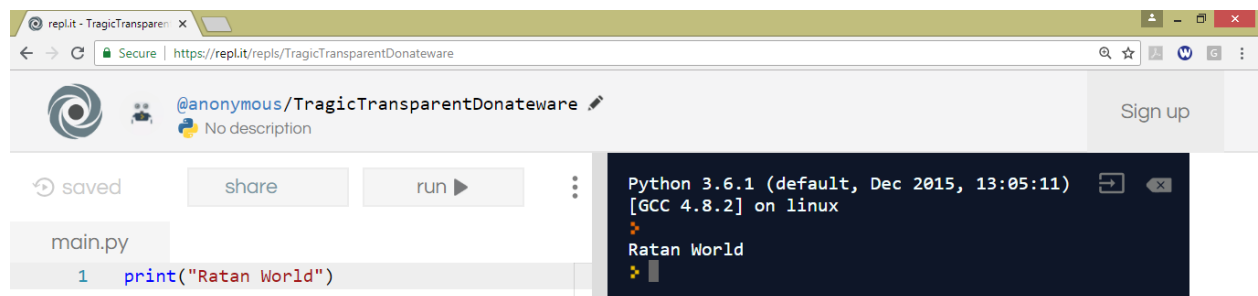


```

Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Welcome>python
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:25:58) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a=10
>>> b=20
>>> c=a+b
>>> print(c)
30
>>>
  
```

Approach-3 by using python online compiler



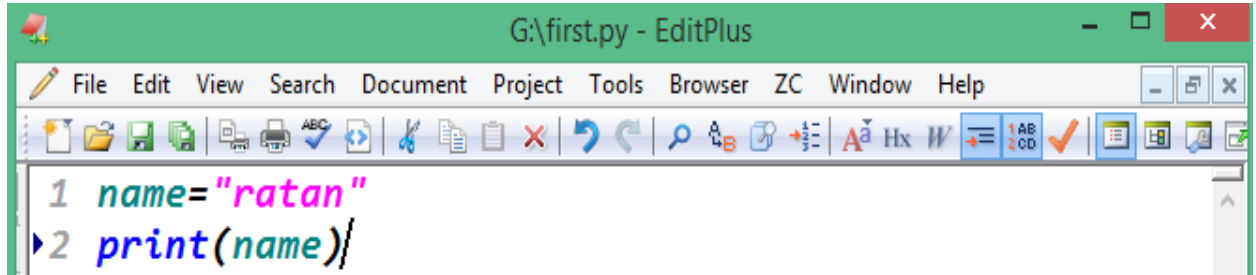
```

Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux

Ratan World
  
```

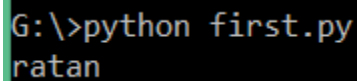

Approach 4: write the application by using editplus, run the application by using command prompt.

Step 1: Write the application in editor (notepad,editplus...etc) run from the command prompt.
Write the application in editor then save the application .py extension.

A screenshot of the EditPlus text editor window. The title bar reads 'G:\first.py - EditPlus'. The menu bar includes File, Edit, View, Search, Document, Project, Tools, Browser, ZC, Window, and Help. The toolbar contains various icons for file operations and editing. The code area shows two lines: '1 name="ratan"' and '2 print(name)'.

```
1 name="ratan"
2 print(name)
```

Step 2: Execute the above file by using following command.

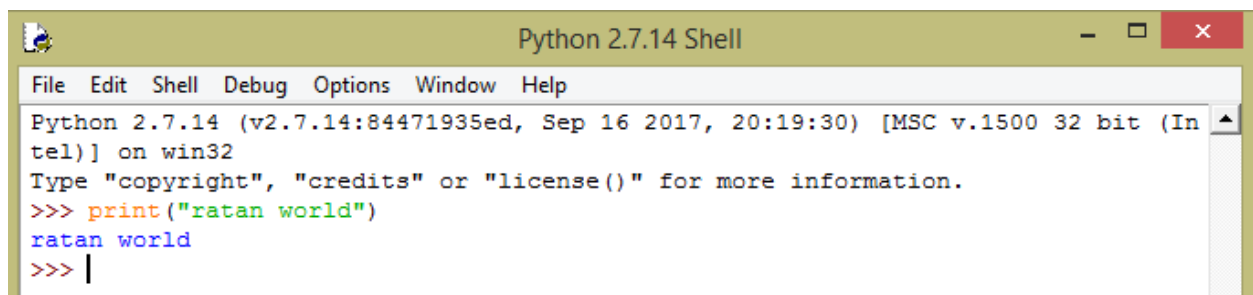
A screenshot of a Windows command prompt window. The prompt is 'G:\>' and the command 'python first.py' has been entered. The output 'ratan' is displayed on the next line.

```
G:\>python first.py
ratan
```

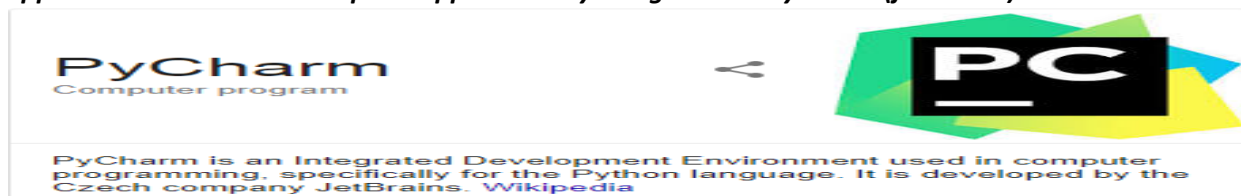
Approach - 5 : By using IDLE

IDLE is the standard Python development environment. Its name is an acronym of "Integrated Development Environment". It works well on both Unix and Windows platforms. It has a Python shell window, which gives you access to the Python interactive mode.

Once we installed python just earch python IDLE.



Approach-6 : We can develop the application by using IDE like PyCharm (jet brains).



Python Keywords :(33 keywords in python)

The above keywords are altered in different versions of python so some extra keywords are added or some might be removed. So it is possible to get the current version of keywords by typing following code in the prompt.

python 2.7 : 31 keywords

```
import keyword
print(keyword.kwlist)

['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'exec', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try',
'while', 'with', 'yield']
```

In python 2.7 print is a keyword to print the data, so print the data parenthesis not required

```
print "Ratan World"
```

python 3.7 : 33 keywords

```
import keyword
print(keyword.kwlist)

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else',
'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass',
'raise', 'return', 'try', 'while', 'with', 'yield']
```

In python 3.x print is a function used to print the data (print is not a keyword) so parenthesis mandatory.

```
print ("Ratan World")
```

To check the given data is keyword or not use below code:

case 1: python 2.7 : True

```
import keyword
print(keyword.iskeyword("print"))
```

case 2: python 3.x : False

```
import keyword
print(keyword.iskeyword("print"))
```

Escape sequence character : start with back slash

```
print("hi \"ratan\" sir")
print("hi \'ratan\' sir")
print("hi \\ratan\\ sir")
print("hi\\tratan\\tsir")
print("hi\\nratan\\nsir")
```

Python Comments:

- ✓ Comments are used to write description about application logics to understand the logics easily.
- ✓ The main objective comments the code maintenance will become easy.
- ✓ The comments are non-executable code.

There are two types of comments in python.

1. Single line comments : write the description in single line & it starts with #

Syntax: # statement

2. Multiline comments:

- write the description in more than one line starts with `"""` ends with `"""` (triple quotes)
- in python while writing the comments we can write double quote or single quote (") or (')

Syntax: `"""`

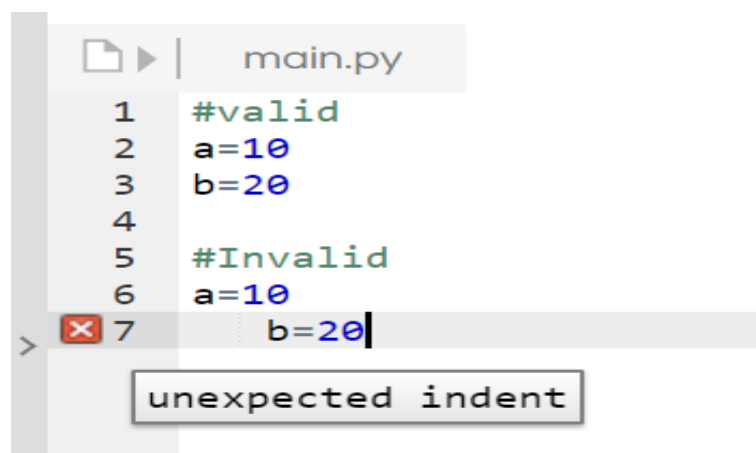
```
St-1
St-2
.....
St-n
"""
```

Syntax: `'''`

```
St-1
St-2
.....
St-n
'''
```

Indentation error:

- ✓ Python uses whitespace (spaces and tabs) to define program blocks whereas other languages like C, C++ use braces ({}) to indicate blocks of codes for class, functions or flow control.
- ✓ The number of whitespaces (spaces and tabs) in the indentation is not fixed, but all statements within the block must be the indented same amount. In the following program, the block statements have no indentation.



Identifiers in python:

Any name in python like function-name, class-name, variable-name...etc is called identifier. The identifier must obey following rules.

- The identifier contains a-z, A-Z, 0-9, _ but should not start with numeric's & does not allow special symbols.

```
class MyClass123:    valid
class 123MyClass:    invalid
class MyClass_123:   valid
class _MyClass:      valid
```

- We cannot use special symbols like !, @, #, \$, % etc. in our identifier.

```
name@one="ratan"    invalid
var$123="durga"     invalid
```

- Identifier can be of any length.

- The identifiers are case sensitive.

```
Name="ratan"
NAME="durga"
```

- Keywords are not allowed as identifier. (**global**, **del** are keywords)

```
class del:
global=100
```

- Is it possible to use predefined class-name & data types names as a identifier but not recommended. (here ABC is predefined abstract class)

```
class ABC:
    def m1(self):
        print("possible")
ABC().m1()
```

General purpose Data types in python:

		Type		value
Number	---	int , float	---	10,20,30 10.5
String	---	str	---	"ratan" 'anu'
Boolean	---	bool	---	True-1 False-0 (starts with capital)

In java: must specify the type of the variable by using data types

```
int eid=111;
String ename="ratan";
double esal=10000.45;
```

In python : dynamically typed long : no need to specify the data type

```
eid=111                                    by default int
ename="ratan"                            by default str
esal=10000.45                            by default float
```

Ex: type() function is used to check the type of the variable.

```
eid=111
ename="ratan"
esal=10000.45
print(eid,ename,esal)

print(type(eid))
print(type(ename))
print(type(esal))
```

Ex: python single line of code & different ways to initialize the variables in python.

```
eid,ename,esal=111,"ratan",10000.45
print(type(eid))
print(type(ename))
print(type(esal))

a,b,c=10,20,30
print(a+b+c)

a,b,c=10,10,10
print(a+b+c)

a=b=c=10
print(a+b+c)
```

Some important observations:

Case 1: concat : in python it is possible to concat only similar type of data.

```
print(10+20)
print("ratan"+"ratan")
print(10.5+20.5)
print(True+True)
print(10+"ratan")    #TypeError: unsupported operand type(s) for +: 'int' and 'str'
print(10+True)
print(10.5+False)
print(10+10.5)
```

- ✓ It is possible to combine int & float because these are belongs to number type.
- ✓ Possible to concat Number & Boolean because Boolean having integral data(True=1 False=0)

case 2 : swapping variables

```
a,b=10,20
a,b=b,a
print(a,b)
```

Case 3: reassigning variables

```
a=10
print(a)
a=100
print(a)
```

case 4: deleting variable

```
a=10
print(a)
del a
print(a) #NameError: name 'a' is not defined
```

case 5: printing the data with separator

```
print(1, 2, 3, 4, sep='*')
print(1, 2, 3, 4, sep='&')
print(1, 2, 3, 4, sep='-')
```

```
E:\>python first.py
1*2*3*4
1&2&3&4
1-2-3-4
```

Ex: printing the data in different formats.

```
eid,ename,esal=111,"ratan",10000.45
print(eid)
print(ename)
print(esal)

print("Emp id=",eid)
print("Emp name=",ename)
print("Emp sal=",esal)

print(eid,ename,esal)

print("Emp id=",eid,"Emp name=",ename,"Emp sal=",esal)
```

Ex:

Case 1: Run the application python 2.7 version

A Python program to show that there are two types in,

Python 2.7 : int and long

And in Python 3 there is only one type : **int**

 $x = 10$

```
print(type(x))
```

$$x = 1000$$

```
print(type(x))
```

```
$python main.py
```

```
<type 'int'>
```

```
<type 'long'>
```

Case 2: Run the application python 3.x version : Python 3 there is only one type : int

 $x = 10$

```
print(type(x))
```

$$x = 1000$$

```
print(type(x))
```

```
$python main.py
```

```
<class 'int'>
```

```
<class 'int'>
```


Number system:

We can represent int values in the following ways :

- 1) Binary form : **Base 2**
- 2) Octal form : **Base 8**
- 3) Decimal form : **Base 10**
- 4) Hexa decimal form : **base 16**

Binary form(Base-2): The allowed digits are : 0 & 1 ,Literal value should be prefixed with 0b or 0B

```
ex: a = 0B1111
    print(a) # 15
    a = 0b0101
    print(a) # 5
```

Octal Form(Base-8): The allowed digits are : 0 to 7 Literal value should be prefixed with 0o or 0O.

```
ex: a = 0o123
    print(a)
    a = 0O111
    print(a)
```

Decimal form(base-10): It is the default number system in Python The allowed digits are: 0 to 9

Eg: a = 10

Hexadecimal System: Hexadecimal system is base 16 number system.

```
ex: val = 0x1a
    print(val)
    val = 0xa
    print(val)
```

Note : A number with the prefix '0b' is considered binary, '0o' is considered octal and '0x' as hexadecimal.

ex: conversion of decimal to binary,octal, hexadecimal.

In this program, we have used built-in functions bin(), oct() and hex() to convert the given decimal number into respective number systems.

```
decimal = 10
print(bin(decimal))
print(oct(decimal))
print(hex(decimal))
```

```
ex: a,b,c,d= 0B10,0o10,10,0X10
    print(a,b,c,d)
```

ex: Finding ascii values.

```
c = input("Enter a character: ")
print("The ASCII value of '" + c + "' is",ord(c))
```

Getting input from the end-user:

In python 2.7 There are two ways to get the input from end-user,

1. **Input() function** : It takes any type of data
2. **raw_input() function** : It takes data only String format

in python 3.x it is possible to take the input from end-user only one way

1. **input() function** : it takes only String data just like raw_input()

ex : python 2.7 Taking data from end-user by using **input** function.

- ✓ If the user gives an integer value, the input function returns integer value
- ✓ If the user gives float value, the input function returns float value
- ✓ If the user gives string value, the input function returns string value

```
num1 = input("enter first num:")
num2 = input("enter second number:")
add = num1+num2
print "Addition=",add
```

output :

```
E:\>python first.py
enter first num:10
enter second
number:20
Addition= 30
```

```
E:\>python first.py
enter first num:20.5
enter second
number:30.3
Addition= 50.8
```

```
E:\>python first.py
enter first num:10
enter second number:"ratan"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
E:\>python first.py
enter first num:10
enter second
number:10.5
Addition= 20.5
```

```
E:\>python first.py
enter first num:"ratan"
enter second
number:"anu"
Addition= ratananu
```

ex-2 : in python 2.7:Taking data from end-user by using **raw_input()** it takes data only in String format.

- ✓ We entered 10 to raw_input : it will take string format.
- ✓ We entered 12.32 to raw_input : it will take string format.
- ✓ We entered "ratan" to raw_input : it will takes string format data.

```
num1 = raw_input('Enter first number: ')
num2 = raw_input('Enter second number: ')
# Add two numbers
sum=num1+num2
# Display the sum
print "Addition=",sum
```

output :

E:\>python first.py

```
Enter first number: 10
Enter second number: 20
Addition= 1020
```

E:\>python first.py

```
Enter first number: 10.5
Enter second number: 20.3
Addition= 10.520.3
```

E:\>python first.py

```
Enter first number: ratan
Enter second number: anu
Addition= ratananu
```

E:\>python first.py

```
Enter first number: 10
Enter second number: ratan
Addition= 10ratan
```

ex: Type conversion process.

```
num1 = int(raw_input('Enter first number: '))
num2 = int(raw_input('Enter second number: '))
# Add two numbers
sum=num1+num2
# Display the sum
print "Addition=",sum
```

E:\>python first.py

```
Enter first number: 10
Enter second number: 20
Addition= 30
```

E:\>python first.py

```
Enter first number: 10.5
ValueError: invalid literal for int() with base 10: '10.5'
```

E:\>python first.py

```
Enter first number: "ratan"
ValueError: invalid literal for int() with base 10: "'ratan'"
```

ex: Taking data from user by using **raw_input()** in the form of String then converting after taking data .

```
num1 = raw_input("Enter first Number :")
num2 = raw_input("Enter Second Number :")
add= float(num1)+int(num2)
print "addition=",add
```

output :

E:\>python first.py

```
Enter first Number :10
Enter Second Number :20
addition= 30.0
```

E:\>python first.py

```
Enter first Number :20.3
Enter Second Number :10
addition= 30.3
```

E:\>python first.py

```
Enter first Number :ratan
Enter Second Number :10
ValueError: could not convert string to float: rattan
```

ex: Python3:

Getting input from the end-user by using **input()**: **Takes data only in string format.**

Note : In Python 3, 'raw_input()' is changed to 'input()'. Thus, 'input()' of Python 3 will behave as 'raw_input()' of Python 2.

```
num1 = input("enter first num:")
num2 = input("enter second number:")
add = num1+num2
print("Addition:",add)
```

E:\>python first.py

```
enter first num: 10
enter second number: 20
Addition: 1020
```

E:\>python first.py

```
enter first num: 10
enter second number: 20.3
Addition: 1020.3
```

E:\>python first.py

```
enter first num: 10.4
enter second number: 20.3
Addition: 10.420.3
```

E:\>python first.py

```
enter first num: ratan
enter second number: anu
Addition: ratananu
```

E:\>python first.py

```
enter first num: 10
enter second number: 20.3
Addition: 1020.3
```

E:\>python first.py

```
enter first num: 10
enter second number: ratan
Addition: 10ratan
```

ex : python 3: Type conversion process

```
num1 = int(input("Enter First Number:"))
num2 = int(input("Enter Second Number:"))
add=num1+num2
print ("Addition:",add)
```

output :

```
Enter First Number: 10
Enter Second Number: 20
Addition: 30
```

```
Enter First Number: 10.5
ValueError: invalid literal for int() with base 10: '10.5'
```

```
Enter First Number: ratan
ValueError: invalid literal for int() with base 10: 'ratan'
```

ex : Type conversion process

```
num1 = input("Enter First Number:")
num2 = input("Enter Second Number:")
add=float(num1)+int(num2)
print ("Addition:",add)
```

output :

```
Enter First Number: 10
Enter Second Number: 20
Addition: 30.0
```

```
Enter First Number: ratan
Enter Second Number: anu
ValueError: could not convert string to float: 'ratan'
```

ex: Basic data types Type conversion process

```
print(int(123.987))
print(int(True))
print(int(False))
print(float(False))
print(str(False))
```

```
a,b=10,20
print(str(a)+str(b))
```

```
print(int("10"))
print(float("10"))
#print(int("10.6")) #ValueError: invalid literal for int() with base 10: '10.6'
```

ex : *printing data in different ways.*

```
eid,ename,esal=111,"ratan",10000.45
```

```
print(eid)
```

```
print(ename)
```

```
print(esal)
```

```
print("Emp id=",eid)
```

```
print("Emp Name=",ename)
```

```
print("Emp Sal=",esal)
```

```
print(eid,ename,esal)
```

```
print("Emp id=",eid,"Emp name=",ename,"Emp sal=",esal)
```

Formatting data with the % & {} :

%d int **%s** string **%f/%g** floating point

```
eid,ename,esal=111,'ratan',10000.45
```

```
print("Emp id=%d Emp name=%s Emp sal=%g"%(eid,ename,esal))
```

```
print("Emp id:{} Emp name:{} Emp sal:{}".format(eid,ename,esal))
```

```
print("Emp id:{0} Emp name:{1} Emp sal:{2}".format(eid,ename,esal))
```

```
print ("Emp id={0} \n Emp name={1} \n Empsal={2}".format(eid,ename,esal))
```

ex:

```
name,age,sal = "balu",24,10000.34
```

```
print("%s age : %d salary is %g"%(name, age, sal))
```

```
print("{} age : {} salary is {}".format(name, age, sal))
```

```
print("{0} age : {1} salary is :{2}".format(name, age, sal))
```

Python Variables:

- ✓ A variable is nothing but a reserved memory location to store values.
- ✓ Variables are used to store the data by using that data we will achieve project requirement.
- ✓ Every variable must have some type i.e. ,
 - Number
 - String
 - List
 - Tuple
 - Dictionary....etc

There are two types of variables in python,

1. Local variables.
2. Global variables.

- ✓ The scope of the global variable : all the functions can use
- ✓ The scope of the local variable is within the function only.

Every function contains two parts

- Function declaration : declare the function by using **def** keyword
- Function calling : call the function by using function name

```
def disp():  
    print("good morning")  
disp()
```

ex-1: **function with arguments** : (function arguments are always local variables)

```
def disp1(name):  
    print("good morning : ",name)  
  
def disp2(age,name):  
    print("good evening : ",name)  
  
disp1("ratan")  
disp2(12,"Anu")
```

ex-2: function arguments are local variables

```
def add(a,b):  
    print(a+b)  
  
def mul(x,y):  
    print(x*y)  
  
add(10,20)  
mul(3,3)
```


ex-3: The variables declared inside the function is called local variables.

```
def disp1():
    name="ratan"      #local var
    print("Good morming :",name)

def disp2():
    name="anu"
    print("Good morming :",name)

disp1()
disp2()
```

ex-4: The scope of the local variable only within the function , if we are calling outside of the function we will get error message.

```
def f():
    name = "ratan"
    print (s)

f()
print (s)  # NameError: name 's' is not defined
```

ex-5: global variable : we can access global variables in all functions.

```
a,b=10,20      # global var
def add():
    print(a+b)
def mul():
    print(a*b)

add()
mul()
```

ex-6 : local variables vs. global variables : **different variable names**

- ✓ The variables which are declared inside the function is called local variable.
- ✓ The variables which are declared outside of the function is called global variables.

```
x,y=10,20      #global variables
def add(a,b):   #local variables
    print (a+b)
    print(x+y)
def mul(a,b):   #local variables
    print (a*b)
    print(x*y)

add(4,4)        # function calling
mul(5,5)        # function calling
```

ex-7: local vs. global variables : same variable name

if the application contains both local & global variables same name then to represent global variables use global() function.

```
a,b=10,20    #global variables
def add(a,b): #local variables
    print(a+b)
    print (globals()['a']+globals()['b'])

def mul(a,b): #local variables
    print(a*b)
    print (globals()['a']*globals()['b'])

# function calling
add(4,4)
mul(5,5)
```

ex-8: Observations**case 1:**

```
s = "I love ratan!"
def f():
    s = "I love ratan!"
    print(s)

f()
print(s)
```

output : I love ratan
I love ratan

UnboundLocalError : inside the function once

case 2:

```
s = "I love ratan!"
def f():
    print(s)
    s = "I love ratan!"
    print(s)

f()
print(s)
```

UnboundLocalError: local variable 's' referenced before assignment

Ex-9:**Case 1:**

```
x = 100
def foo():
    x = x * 2
    print(x)
foo()
```

UnboundLocalError: local variable 'x' referenced before assignment

Case 2 :

```
x = 100
def foo():
    x=10
    x = x * 2
    print(x)
foo()
```

output : 20

ex - 10:

case 1: Inside the function declaring global variable by using **global** keyword.

```
def disp():  
    global name  
    name="ratanit"  
    print(name)
```

```
disp()  
print(name)
```

case 2: Inside the function to represent global variables use **global** keyword.

```
str="ratan"  
def wish():  
    global str          #reference global variable inside the function  
    str="good morning"  
    print (str)
```

```
wish()  
print (str)
```

ex-11: Assignment – write the output

```
def a_fun():  
    global foo  
    foo = 'A'
```

```
def b_fun():  
    global foo  
    foo = 'B'
```

```
b_fun()  
a_fun()  
print (foo)
```

Ex-12: Assignment: global keyword: write the output

```
a = 10

def f():
    print ('Inside f() : ', a)

def g():
    a = 20
    print ('Inside g() : ',a)

def h():
    global a
    a = 30
    print ('Inside h() : ',a)

print ('global : ',a)
f()
print ('global : ',a)
g()
print ('global : ',a)
h()
print ('global : ',a)
```

ex-13 : Assignment : write the output

```
a_var = 10
b_var = 15
e_var = 25
d_var = 100
def a_func(a_var):
    print("in a_func a_var =", a_var)
    b_var = 100 + a_var
    d_var = 2 * a_var
    print("in a_func b_var =", b_var)
    print("in a_func d_var =", d_var)
    print("in a_func e_var =", e_var)
    return b_var + 10

c_var = a_func(b_var)

print("a_var =", a_var)
print("b_var =", b_var)
print("c_var =", c_var)
print("d_var =", d_var)
```

ex-14: Assignment : write the output

```
a,b,x,y = 1,15,3,4
```

```
def foo(x, y):  
    global a  
    a = 42  
    x,y = y,x  
    b = 33  
    b = 17  
    c = 100  
    print (a,b,x,y)
```

```
foo(17,4)  
print (a,b,x,y)
```

ex-15: Inner function : declaring function inside the another function

```
def outer():  
    print("Outer function")  
    def inner():  
        print("inner function")  
  
    inner() # calling inner function  
  
outer() # calling outer function
```

ex-16 : Inner function can access outer function variables

```
def outer():  
    print("Outer function")  
    name1="ratan"  
  
    def inner():  
        print("inner function")  
        name2="anu"  
        print(name1)  
        print(name2)  
  
    inner() #inner function calling  
    print(name1) # outer function variable printing  
  
outer() # outer funmction calling
```

Ex-17: Inside the inner function to represent outer function variable use **nonlocal** keyword.

```
def outer():
    name1="ratan"
    def inner():
        nonlocal name1
        name1="durga"

    inner()      #inner function calling
    print(name1)

outer()         # outer funmction calling
```

ex-18: **nonlocal vs global**

```
name='ratan'
def outer():
    var_outer = 'ratan'
    def inner():
        nonlocal var_outer
        var_outer="anu"
        global name
        name="ratanit"
        print (var_outer)

    print (var_outer)
    inner()      #calling of inner function
    print (var_outer)

outer() # outer function calling
print(name)
```

ex-19: **Assignment** : Write the output

```
def f():
    x = 42
    def g():
        global x
        x = 43
    print("Before calling g: ",x)
    g()
    print("After calling g: ",x)

f()
print("x in main: " ,x)
```

ex 20: Assignment : write the output

```
def outer():
    s="ratan"
    def inner1():
        s="durga"
    def inner2():
        nonlocal s
        s="sunny"
    def inner3():
        global s
        s="sravya"

    print(s)
    inner1()
    print(s)
    inner2()
    print(s)
    inner3()
    print(s)

outer()
print(s)
```

ex-21: non-local vs global

```
def disp():
    name="ratan"

    def disp1():
        nonlocal name
        name = "anu"
    def disp2():
        nonlocal name
        name = "durga"
    def disp3():
        global name
        name = "sunny"

    print(name)
    disp1()
    print(name)
    disp2()
    print(name)
    disp3()
    print(name)

disp()
print("global :", name)
```


ex: *Assigning local variable values to global variables.*

```
eid,ename,esal=1,'aaa',10000.56
```

```
def emp(eid,ename,esal):
    globals()['eid']=eid
    globals()['ename']=ename
    globals()['esal']=esal
    print(eid,ename,esal)
```

```
def disp():
    print(eid,ename,esal)
```

```
emp(111,'ratan',10000.45)
disp()
print(eid,ename,esal)
```

pass Statements :

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

Case: *while True:*
pass

Case: *This is commonly used for creating minimal classes:*
class MyEmptyClass:
pass

Case: *Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored.*

```
def initlog(*args):
    pass # Remember to implement this!
```

Ex:

```
a = int(input("Enter first value:"))
b = int(input("Enter first value:"))
def absolute_value(n):
    if n < 0:
        n = -n
    return n
if absolute_value(a) == absolute_value(b):
    print("The absolute values of", a, "and", b, "are equal.")
else:
    print("The absolute values of", a, "and", b, "are different.")
```

Python Functions:

- Step 1: Declare the function with the keyword **def** followed by the function name.
- Step 2: Write the arguments inside the opening and closing parentheses of the function, and end the declaration with a colon.
- Step 3: Add the program statements to be executed
- Step 4: End the function with/without return statement.

Syntax :

```
def function_name(parameters):  
    """doc string"""  
    statement(s)
```

ex:

```
def userDefFunction (arg1, arg2, arg3 ...):  
    program statement1  
    program statement3  
    program statement3  
    ....  
    return
```

There are two types of functions

1. Built-in functions - Functions that are built into Python.
2. User-defined functions - Functions defined by the users themselves.

Advantages of functions :

- ✓ User-defined functions are reusable code blocks; they only need to be written once, then they can be used multiple times.
- ✓ These functions are very useful, from writing common utilities to specific business logic.
- ✓ The code is usually well organized, easy to maintain, and developer-friendly.
- ✓ A function can have dif types of arguments & return value.

ex : `def disp():`
 `print("hi ratan sir")`
 `print("hi anu")`

 `disp()` # function calling

ex: To specify no body of the function use **pass statement**.
 `def disp():`
 `pass`

 `disp()`

ex: one function is able to call more than one function.
 `def happyBirthday(person):`
 `print("Happy Birthday dear ",person)`

 `def main():`
 `happyBirthday('ratan')`
 `happyBirthday('anu')`

 `main()`

ex: inner functions : declaring the function inside the another function.
 `def ex4():`
 `var_outer = 'foo'`
 `def inner():`
 `var_inner = 'bar'`
 `print (var_outer)`
 `print (var_inner)`

 `inner()` #calling of inner function
 `print (var_outer)`

 `ex4()`

ex:

- ✓ inside the inner function to represent outer function variable use **nonlocal** keyword.
- ✓ inside the function to represent the global value use **global** keyword.

`def ex4():`
 `var_outer = 'ratan'`
 `def inner():`
 `nonlocal var_outer`

```
var_outer="anu"  
print (var_outer)
```

```
inner() #calling of inner function  
print (var_outer)
```

```
ex4()
```

```
ex:  name='ratan'  
def ex4():  
    var_outer = 'ratan'  
    def inner():  
        nonlocal var_outer  
        global name  
        name="ratanit"  
        var_outer="anu"  
        print (var_outer)  
  
    inner() #calling of inner function  
    print (var_outer)
```

```
ex4()  
print(name)
```

function vs arguments:-

1. default arg
2. required arg
3. keyword argument
4. variable argument

Default arguments:

When we call the function if we are not passing any argument the default value is assigned.

```
ex :  defempdetails(eid=1,ename="anu",esal=10000):  
        print ("Emp id =", eid)  
        print ("Emp name = ", ename)  
        print ("Empsal=", esal)  
        print("*****")
```

```
empdetails()
empdetails(222)
empdetails(333,"durga")
empdetails(111,"ratan",10.5)
```

ex :

```
def defArgFunc( empname, emprole = "Manager" ):
    print ("Emp Name: ", empname)
    print ("Emp Role ", emprole)

print("Using default value")
defArgFunc("Ratan")
print("*****")
print("Overwriting default value")
defArgFunc("Anu", "CEO")
```

Required arguments:

Required arguments are the mandatory arguments of a function. These argument values must be passed in correct number and order during function call.

```
Def show(a,b):      #function declaration
show(10,20)         #function calling
```

ex 1:

```
def reqArgFunc( empname):
    print ("Emp Name: ", empname)

reqArgFunc("ratan")
```

ex:

```
def add(a,b):
    print(a+b)

add(10,20)
add(10.5,20.4)
add("ratan","anu")
add(10,10.5)
add("ratan",10)
```

ex 2:

```
def addition(x,y):
    print x+y

a,b=10,20
addition(100,200)
addition(a,b)
```

none :-

- ✓ *none is a special constant in python to represent absence of value or null value.*
- ✓ *None is not a **0, false, []***
- ✓ *Void functions that don't return anything will return a none object automatically.*

Case 1: Test.py

```
def add():
```

```
    a=10
```

```
    b=20
```

```
    c=a+b
```

```
x = add()
```

```
print x
```

G:\>python Test.py

None

case 2: Test.py

```
def add(a):
```

```
    if (a % 2) == 0:
```

```
        return True
```

```
x = add(3)
```

```
print x
```

G:\>python Test.py

None

Keyword arguments / named arguments:

The keywords are mentioned during the function call along with their corresponding values. These keywords are mapped with the function arguments so the function can easily identify the corresponding values even if the order is not maintained during the function call.

Using the Keyword Argument, the argument passed in function call is matched with function definition on the basis of the name of the parameter.

```
def empdetails(name,role):  
    empdetails ( name = "ratan", role = "Manager")           #function calling  
    empdetails (role = "developer", name = "anu")             #function calling
```

```
ex:  def msg(id,name):  
        print id  
        print name
```

```
msg(id=111,name='ratatn')  
msg(name='anu',id=222)
```

```
ex:  def disp(eid,ename):  
        print(eid,ename)
```

```
disp(eid=111,ename="anu")  
disp(ename="ratan",eid=222)  
disp(333,"durga")  
disp(444,ename="sravya")  
disp(eid=555,"aaa") #SyntaxError: positional argument follows keyword argument
```

```
ex:  def emp(eid,ename="ratan",esal=10000):  
        print(eid,ename,esal)
```



```
emp(111)
emp(222,ename="anu")
#emp(ename="anu",222) #error SyntaxError: positional argument follows keyword
argument
emp(222,esal=100,ename="anu")
```

- ✓ In the first case, when msg() function is called passing two values i.e., id and name the position of parameter passed is same as that of function definition and hence values are initialized to respective parameters in function definition. This is done on the basis of the name of the parameter.
- ✓ In second case, when msg() function is called passing two values i.e., name and id, although the position of two parameters is different it initialize the value of id in Function call to id in Function Definition. same with name parameter. Hence, values are initialized on the basis of name of the parameter.

Variable number of arguments:

This is very useful when we do not know the exact number of arguments that will be passed to a function. Or we can have a design where any number of arguments can be passed based on the requirement.

```
def varlengthArgs(*varargs):
    varlengthArgs(10,20,30,40)           #function calling
```

ex:

```
def disp(*var):
    for i in var:
        print("var arg=",i)
```

```
disp()
disp(10,20,30)
disp(10,20.3,"ratan")
```

Function return type:-

Example 1:

```
def addition(x,y):
    return a+y
```

```
a,b=10,20
sum=addition(a ,b)
print (sum)
```

ex : If the functions not returns the value but if we are trying to hold the values it returns none as a default value.

```
def add():  
    a=10  
    b=20  
    c=a+b
```

```
x = add()  
print x
```

```
G:\>python Test.py  
None
```

Ex:

```
def hello():  
    print('Hello!')
```

```
def print_welcome(name):  
    print('Welcome,', name)
```

```
def area(width, height):  
    return width * height
```

```
def positive_input(prompt):  
    number = float(input(prompt))  
    while number <= 0:  
        print('Must be a positive number')  
        number = float(input(prompt))  
    return number
```

```
name = input('Your Name: ')  
hello()  
print_welcome(name)  
print('To find the area of a rectangle, enter the width and height below.')  
w = positive_input('Width: ')  
h = positive_input('Height: ')  
  
print('Width =', w, ' Height =', h, ' so Area =', area(w, h))
```

Ex:

```
def factorial(n):
```

```

if n <= 1:
    return 1
return n * factorial(n - 1)

n = int(input("enter a number to perform factorial:"))
print("%d factorial is ="%(n),factorial(n))

```

Names can only contain upper/lower case digits (A-Z, a-z), numbers (0-9) or underscores _;
 Names cannot start with a number;
 Names cannot be equal to reserved keywords:

Python control flow statements

If-else statement :

- ✓ if the condition **true** if block executed.
- ✓ if the condition **false** else block executed.

Syntax :

```

if(condition):
    Statement(s)
else :
    statement(s)

```

ex:

```

a=10
if(a>10):
    print("if body")
else:
    print("else body")

```

ex : In python 0=false 1=true

```

if 0:
    print("hi ratan")
else:
    print("hi anu")

```

ex: In python Boolean constants start with uppercase character. (**True,False**)

```

if True:
    print("true body")
else:
    print("false body")

```

ex:

```
year = 2000
if year % 4 == 0:
    print("Year is Leap")
else:
    print("Year is not Leap")
```

ex: conditional expression

a=10

```
if(a>10):
    print("Ratan")
else:
    print("Anu")
```

print("Ratan") if a>10 else print("Anu")

```
x = 10 if a>10 else 20
print(x)
```

elif statement :

- ✓ The keyword 'elif' is short for 'else if'
- ✓ An **elif** sequence is a substitute for the switch or case statements found in other languages

Syntax:

```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

ex:

```
a=200
if a==10:
    print("ratan")
elif a==20:
    print("anu")
```

```
elif a==30:
    print("durga")
else:
    print("sravya")
```

ex:

```
number = 23
guess = int(input("Enter an integer : "))
if guess == number:
    print("Congratulations, you guessed it.")
elif guess < number:
    print("No, it is a little higher number")
else:
    print("No, it is a little lower number")
print("rest of the app")
```

ex:

```
x = int(input("Please enter an integer: "))
if x < 0:
    print ("x is negative value")
elif x % 2:
    print ("your values positive and odd")
else:
    print ("your value is positive and even ")
print("process is done")
```

for loop :

- ✓ Used to print the data **n** number of times based on conation.
- ✓ If you do need to iterate over a sequence of numbers, use the built-in function `range()`.

*syntax: for <temp-variable> in <sequence-data>:
statement(s)*

range() function :

<code>range(10)</code>	0 1 2 3 4 5 6 7 8 9
<code>range(5, 10)</code>	5 through 9
<code>range(0, 10, 3)</code>	0, 3, 6, 9

Syntax:

```
for iterator_name in range(10):
    ...statements...
```

```
for iterator_name in range(start,end):
    ...statements...
```

```
for iterator name in range(start,stop,increment):
```

...statements...

ex:

```
for x in range(5):  
    print("ratanit:",x)
```

```
for x in range(5,10):  
    print("ratanit:",x)
```

```
for x in range(5,10,3):  
    print("ratanit:",x)
```

```
for x in range(10,5,-3):  
    print("ratanit:",x)
```

ex :

```
for x in range(-5):  
    print("ratanit:",x)
```

```
for x in range(-10,-5):  
    print("ratanit:",x)
```

```
for x in range(-10,-5,3):  
    print("ratanit:",x)
```

```
for x in range(-5,-10,-2):  
    print("ratanit:",x)
```

For Loops with else block :

ex: *else block is always executed if the for loop terminated normally.*

```
for i in range(1, 5):  
    print(i)  
else:  
    print('The for loop is over')
```

else block is not executed in three cases

case 1: if the exception raised in for loop else block not executed.

```
for x in range(10):  
    print("ratan world",x)  
    print(10/0)  
else:  
    print("else block")
```

case 2: In loop when we use break statement the else block not executed.

```
for x in range(10):
```

```
print("ratan sir",x)
if(x==4):
    break
else:
    print("else block")
```

case 3: when we use `os._exit(0)` the virtual machine shutdown, the else block not executed.

```
import os
for x in range(6):
    print("ratanit:",x)
    os._exit(0)
else:
    print("else block")
```

ex: write a program to print addotion of 1-100 numbers.

```
sum=0
for i in range(1,100):
    sum=sum+i
print sum
```

ex: write a program to find even numbers form 1-20.

```
for i in range(1,10):
    if i%2==0:
        print("even number=",i)
    else:
        print("odd number",i)
```

ex:

```
# Iterating over a List data
print("List Iteration")
l = ["ratan", "anu", "sunny"]
for i in l:
    print(i)
```

Iterating over a tuple

```
print("Tuple Iteration")
t = ("aaa", "bbb", "ccc")
for i in t:
    print(i)
```

Iterating over a String

```
print("String Iteration")
s = "ratanit"
for i in s :
    print(i)
```

Iterating over dictionary

```
print("Dictionary Iteration")
d = {111:"rattan",222:"anu"}
for l in d :
    print("%s %d" % (l, d[l]) )
```

ex:

```
words = ["cat", "apple", "ratanit", "four"]
for w in words:
    print(w, len(w))
```

```
words = ["cat", "apple", "ratanit", "four"]
for w in words[1:3]:
    print(w, len(w))
```

While loop:

```
while <expression>:
    Body
```

ex :

```
a=0
while(a<10):
    print ("hi ratan sir")
    a=a+1
```

ex: **else** is always executed when while loop terminated normally.

```
a=0
while(a<10):
    print ("hi ratan sir",a)
    a=a+1
else:
    print("else block after while");
    print("process done")
```

else block is not executed in three cases

case 1: if the exception raised in while loop else block not executed.

```
while True:
    print(10/0)
else:
    print("while loop terminated normally...")
```

case 2: In loop when we use break statement the else block not executed.

```
a=0
while(a<10):
    a=a+1
    if(a==3):
        break
    print("RATANIT:",a)
else:
    print("else block")
```

case 3: when we use os._exit(0) the virtual machine shutdown, the else block not executed.

```
import os
while(True):
    print("RATANIT:")
    os._exit(0)
else:
    print("else block")
```

ex: The below example represent infinite times.

```
while(True):
    print ("hi ratan sir")
```

ex: write the program to repeat the loop until you entered valid credentials(username&pass).

```
while True:
    eid = input("Enter emp id")
    ename=input("enter emp name")
    if(ename=="ratan"):
        print("login success")
        break
    else:
        print("login fail try with valid name")
```

ex: while 1:

```
n = input("Please enter 'hello':")
if n.strip() == 'hello':
    break
else:
```

```
print("u entered wrong input")
```

ex:

```
while True:
    n = input("enter some name")
    if(n=='exit'):
        break
    elif(len(n)<3):
        print("name is very small...")
    print("you entered good name....")
```

ex:

```
num=25
while True:
    guess = int(input("enter guess number:"))
    if(num==guess):
        print("Congratulations....")
        break
    elif num>guess:
        print("your value is lower")
    elif num<guess:
        print("your value is higher")
    print("process done")
```

Break vs. continue:

- ✓ Break is used to stop the execution.
- ✓ Continue used to skip the particular iteration.

ex: for i in range(1,10):
 if(i==4):
 break
 print(i)

ex: for i in range(1,10):
 if(i==4):
 continue
 print(i)

ex: for letter in 'ratan':
 if letter == 'a' or letter == 'r':
 continue
 print ('Current Letter :', letter)

ex: for letter in 'ratanit':
 if letter == 'a' or letter == 'x':
 continue
 elif letter=='i':
 break
 print ('Current Letter :', letter)

ex:

```
for x in range(1,10):
```

```
if(x%2==0):  
    print("Even value:",x)  
    continue  
print("Odd value:",x)
```

Pass statement:

- ✓ The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action.
- ✓ Pass is an empty statement in python.
- ✓ If you want declare the function now without implementation, but you want provide the implementation in future use pass statement.

```
while True:  
    pass
```

```
class MyEmptyClass:  
    pass
```

```
def disp1():  
    pass
```

```
if False:  
    pass
```

ex:

```
politicleParties=['tdp','ysrcp','congress','bjp']  
electionYear=["2014","2019","2005","2001"]  
countryStatus=["worst","developing","developed"]  
corruptionStatus=["Max","Normal","Min"]  
  
for party in politicleParties:  
    year=input("enter a year")  
  
    if year in electionYear:  
        if year == "2014":  
            print("tdp Wins!")  
            print("Country status: "+countryStatus[2])  
            print("Corruption Status: "+corruptionStatus[0])  
            break  
  
        elif year == "2019":  
            print("ysrcp Won")  
            print("Country status: "+countryStatus[0])
```

```
print("Corruption Status: "+corruptionStatus[0])
break

elif year == "2005":
    print("congress won!")
    print("Country status: "+countryStatus[0])
    print("Corruption Status: "+corruptionStatus[0])
    break

elif year == "2001":
    print("bjp won!")
    print("Country status: "+countryStatus[0])
    print("Corruption Status: "+corruptionStatus[0])
    break
else:
    print("Wrong year of election!")
```

Data types in Python

❖ Numbers

- *Int / float/complex : type*
- *Describes the numeric value & decimal value*
- *These are immutable modifications are not allowed.*

❖ Boolean

- *bool : type*
- *represent True/False values.*
- *0 =False & 1 =True*
- *Logical operators and or not return value is Boolean*

✓ Strings

- *str : type*
- *Represent group of characters*
- *Declared with in single or double or triple quotes*
- *It is immutable modifications are not allowed.*

✓ Lists

- ✓ *list : type*
- ✓ *group of heterogeneous objects in sequence.*
- ✓ *This is mutable modifications are allowed*
- ✓ *Declared with in the square brackets []*

✓ Tuples

- ✓ *tuple : type*
- ✓ *group of heterogeneous objects in sequence*
- ✓ *this is immutable modifications are not allowed.*
- ✓ *Declared within the parenthesis ()*

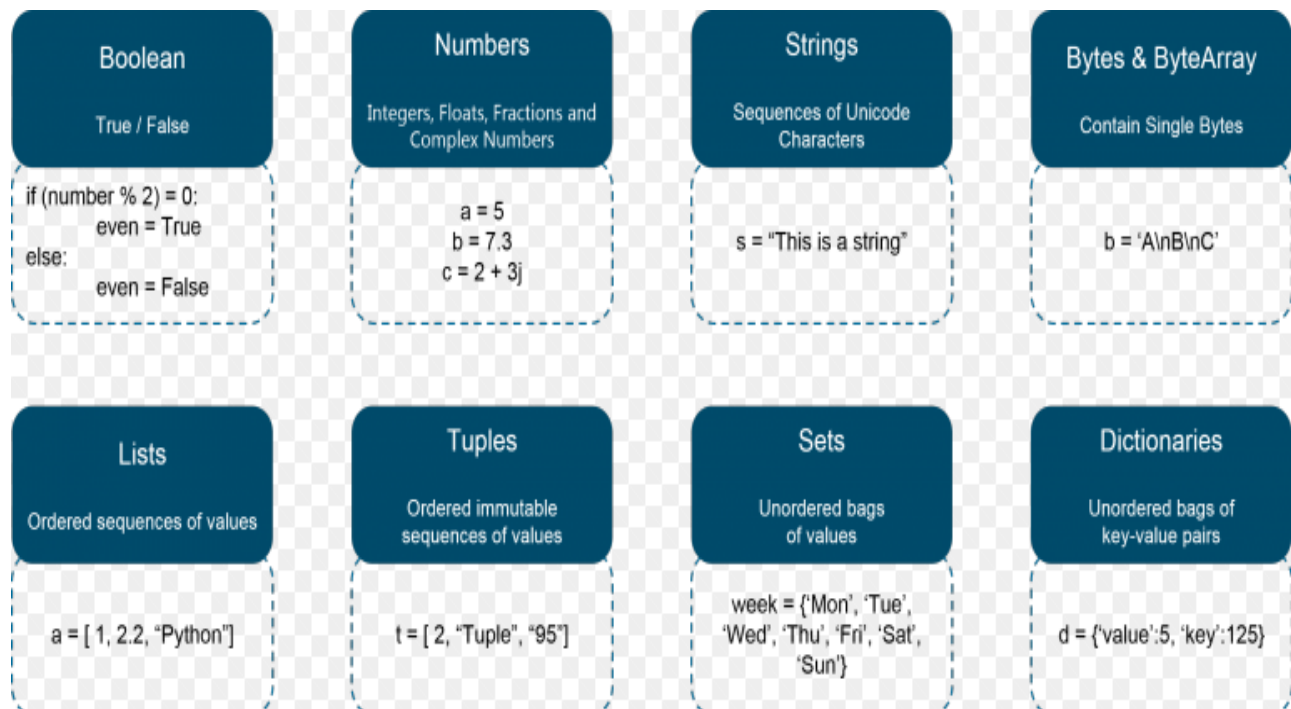
✓ Sets

- ✓ *set : type*
- ✓ *group of heterogeneous objects in unordered*
- ✓ *this is mutable modifications are allowed*
- ✓ *declared within brases { }*

✓ Dictionaries

- ✓ *dict : type*
- ✓ *it stores the data in key value pairs format.*
- ✓ *Keys must be unique & value*
- ✓ *It is mutable modifications are allowed.*
- ✓ *Declared within the curly brases {key:value}*

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	



Boolean data type :- (bool)

- ✓ true & false are result values of comparison operation or logical operation in python.
- ✓ true & false in python is same as 1 & 0 1=true 0=false
- ✓ except zero it is always True.
- ✓ while writing true & false first letter should be capital otherwise error message will be generated.
- ✓ Comparison operations are return Boolean values.

ex:

```
a = bool(1)
b = bool(0)
c = bool(10)
d = bool(-5)
e = int(True)
f = int(False)
print("a: ", a, " b: ", b, " c: ", c, " d: ", d, " e: ", e, " f: ", f)
```

ex:

```
T,F = True, False
print ("T: ", T, " F: ", F)
print ("T and F: ", T and F) #False
print ("T and F: ", F and T) #False
print ("T and T: ", T and T) #True
print ("F and F: ", F and F) #False
print ("not T: ", not T) # False
print ("not F: ", not F) # True
print ("T or F: ", T or F) # True
print ("T or F: ", F or T) # True
print ("T or T: ", T or T) # True
print ("F or F: ", F or F) # False
```

ex:

```
print (1 == 1) #true
print (5 > 3) #true
print (True or False) #true
print (3 > 7) #false
print (True and False) #false
print ("ratan"=="ratan") #true
print ('a'=="a") #true
print (1==True) #true
print (False==0) #true
print (True+True) #2
print (False+False) #2
```

Strings Data type

- ✓ A string is a list of characters in order enclosed by single quote or double quote.
- ✓ Python string is immutable modifications are not allowed once it is created.
- ✓ String index starts from 0, it support both forward and backward indexing.

Ex 1: string data representation.

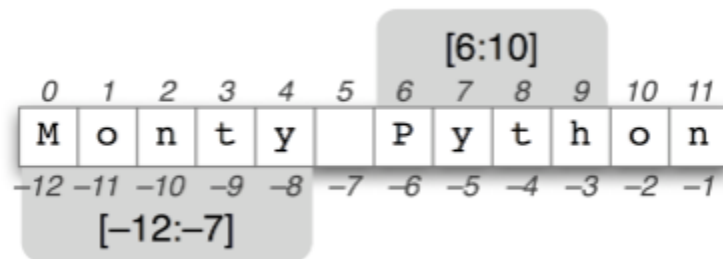
```
s1="ratan"
print(s1)
```

```
s2='anu'
print(s2)
```

Ex -2: indexing both forward & backward.

Slice Notation

<string_name>[startIndex : endIndex]	: Start to end
<string_name>[: endIndex]	: No start so it will start from 0 index
<string_name>[startIndex :]	: No end so it will take up to end



```
str="ratanit"
print(str[3])           #a
print(str[1:3])         #at
print(str[1:4:2])       #aa
print(str[3:])          #anit
print(str[:4])          #ratan
print(str[:])           #ratanit
#print(str[20])         IndexError: string index out of range

print(str[-3])
print(str[-6:-4])
print(str[-3:])
print(str[:-5])
print(str[:])
#print(str[-9])         IndexError: string index out of range
```


Ex-3: len() strip() Functions

```
name1=" ratan "  
print(len(name1))  
print(name1.strip())  
print(len(name1.strip()))  
  
name2="@@@@ratan####"  
print(name2.lstrip("@"))  
print(name2.rstrip("#"))  
print(name2.lstrip("@").rstrip("#"))
```

ex - 4:

- ✓ To print the memory address use **id()** function.
- ✓ **==** , **!=** Operator will check the data comparison.
- ✓ To check the memory address use **is** & **is not** operator. If two reference variables are pointing to same memory it returns true otherwise false.

```
name1="ratan"  
name2='anu'  
name3='ratan'  
  
print(id(name1))  
print(id(name2))  
print(id(name3))  
  
print(name1 is name2)  
print(name1 is name3)  
print(name1 is not name2)  
print(name1 is not name3)  
  
print(name1==name2)  
print(name1==name3)  
print(name1!=name2)  
print(name1!=name3)  
  
# list data memory comparison  
l1=[10,20,30]  
l2=[10,20,30]  
print(id(l1))  
print(id(l2))  
print(l1 is l2)  
print(l1 is not l2)  
print(l1==l2)
```

ex-5:

count() : To find data occurrences

in: "in" operator return true if a character or the entire substring is present in the specified string, otherwise false.

not in: "not in" operator return true if a character or entire substring does not exist in the specified string, otherwise false.

```
s1="ratanit"
print("ratan" in s1)
print("durga" in s1)
print("ratan" not in s1)
print("durga" not in s1)

s2="ratanratan"
print(s2.count('a'))
print(s2.count('ratan'))
print(s2.count('a',2,7))
print(s2.count('ratan',2,7))
```

Ex - 6:

String Formatting with the % Operator:

%d int **%s** string **%f/%g** floating point

```
x = "apples"
y = "lemons"
a=10
b=10.5
print( "In the basket are %s %d and %s %f " % (x,a,y,b))
```

String Formatting with the { } Operators:

```
eid,ename,esal=111,'ratan',10000.45

print("Emp id=%d Emp name=%s Emp sal=%g"%(eid,ename,esal))
print("Emp eid={} Emp name={} emp sal={}".format(eid,ename,esal))
print("Emp eid={0} Emp name={1} emp sal={2}".format(eid,ename,esal))
```

Ex-7: Concatenation & Replication

```
s1="ratan"
s2='anu'
s3=s1+s2
print(s3)
```

```
s="ratan"
ss=s*3
print(ss)
```

*Note: In python it is possible to concat only similar type of data; if we are trying to combine different types of data we will get **TypeError**.*

Ex-8: Relational Operators:

All the comparison operators i.e., (<,>,<=,>=,==,!=,<>) are also applicable to strings. The Strings are compared based on dictionary Order.

```
print("ratan">"anu")
print("ratan"<"anu")
print("ratan">="Ratan")
print("Ratan"<="ratan")
print("ratan"=="ratan")
print("Ratan"!="anu")
```

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Method**Functionality**

<code>s.find(t)</code>	index of first instance of string t inside s (-1 if not found)
<code>s.rfind(t)</code>	index of last instance of string t inside s (-1 if not found)
<code>s.index(t)</code>	like <code>s.find(t)</code> except it raises <code>ValueError</code> if not found
<code>s.rindex(t)</code>	like <code>s.rfind(t)</code> except it raises <code>ValueError</code> if not found
<code>s.join(text)</code>	combine the words of the text into a string using s as the glue
<code>s.split(t)</code>	split s into a list wherever a t is found (whitespace by default)
<code>s.splitlines()</code>	split s into a list of strings, one per line
<code>s.lower()</code>	a lowercased version of the string s
<code>s.upper()</code>	an uppercased version of the string s
<code>s.title()</code>	a titlecased version of the string s
<code>s.strip()</code>	a copy of s without leading or trailing whitespace
<code>s.replace(t, u)</code>	replace instances of t with u inside s

ex-9 : upper() , lower() , capitalize() , replace()

```
text = "hi ratan sir python is very good"
print ("upper =>", text.upper())
print ("lower =>", text.lower())
print ("capitalize =>", text.capitalize())
print ("join =>", "+" .join(text.split()))
print ("replace =>", text.replace("python", "java"))
```

ex-10 : enumerate() functions

- ✓ The enumerate() function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.
- ✓ Split () function used to split the data, the default splitting character is space.

```
str2="ho si.r how r u"
print (str2.split()) #default splitting character is space
print (str2.split("."))
```

```
str4 = 'ratanit'
# enumerate()
print(tuple(enumerate(str4)))
print(list(enumerate(str4)))
```

ex-11 : startswith() & endswith() functions.

```
string1="Welcome to ratanit";
print (string1.endswith("ratanit"))
print (string1.endswith("to",2,16))
print (string1.startswith("Welcome"))
print (string1.startswith("come",3,10))
```

ex -12 : index() & find() & swapcase() functions

- ✓ When we use find function, the element is not present it return -1
- ✓ When we use index function, the element is not present it will generate **ValueError**.

```
str="Welcome to ratanit";
print (str.find("ratanit"))
print(str.find("anu"))

print (str.index("come"))
print (str.index("t",10,15))
#print(str.index("anu")) ValueError: substring not found

string1="Welcome To Ratan It"
print (string1.swapcase())
```

ex-13 : isalnum() , isalpha() , isdigit() functions

```
str="Welcome to ratanit"
print (str.isalnum()) #false
str1="Python36"
print (str1.isalnum()) #true
string2="HelloPython"
print (string2.isalpha()) #true
string3="This is Python3.1.6"
print (string3.isalpha()) #false
string4="HelloPython";
print (string4.isdigit()) #false
string5="98564738"
print (string5.isdigit()) #true
```

ex-14 : islower() , isupper() , isspace() functions

```
string1="Ratanit";
print (string1.islower()) #false
string2="ratanit"
print (string2.islower()) #true
string3="ratanit";
print (string3.isupper()) #false
string4="RATANIT"
print (string4.isupper()) #true
string5="WELCOME TO WORLD OF PYT"
print (string5.isspace()) #false
string6=" ";
print (string6.isspace()) #true
```

Assignment-1 Count the number of **a** occurrences in given string without using count function.

```
count=0
for i in "ratanit":
    if(i=="a"):
        count=count+1
print("a charcater occur:",count)
```

Assignment-2 : Count number of **ratan** occurrences in given string without using count function.

```
s="ratan anu ratan durga ratan"
words = s.split()
```

```
count=0
for i in words:
    if(i=="ratan"):
        count=count+1
print(count)
```

Assignment 3 : python program perform sorting of words in given input String

```
mystr = input("Enter a string: ")
# breakdown the string into a list of words
words = mystr.split()
```

```
# sort the list
words.sort()
for word in words:
    print(word)
```

```
# reverse sort the list
words.sort(reverse=True)
for word in words:
    print(word)
```

Errors in Strings data type:

NameError:

```
str = "hi sir"
print (substring(2,4))
```

G:\>python first.py

NameError: name 'substring' is not define

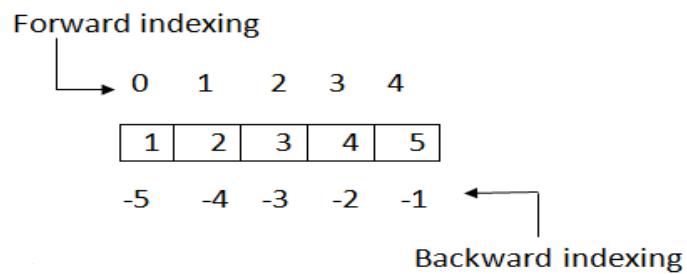
IndexError: str = "hi sir" print(str[10]) # IndexError: string index out of range

TypeError: print (10+"ratan") # TypeError: unsupported operand type(s) for +: 'int' and 'str'

*******String completed : Than you*******

List data type: (list)

- ✓ List is used to store the group of values.
- ✓ A list can be composed by storing a sequence of different type of values separated by commas.
`<list_name> = [value1,value2,value3,...,valuen]`
- ✓ List is **mutable** object so we can do the manipulations.
- ✓ A python list is enclosed between square `[]` brackets.
- ✓ In list insertion order is preserved it means in which order we inserted element same order output is printed.
- ✓ List duplicate objects are allowed.
- ✓ The list contains forward indexing & backward indexing.

**ex : List data**

```
data1=[1,2,3,4]           # list of integers
data2=['x','y','z']       # list of String
data3=[12.5,11.6];        # list of floats
data4=['ratan',10,56.4,'a'] # list with mixed datatypes
```

```
print (data1)
print (data2)
print (data3)
print (data4)
print(type(data1))
```

ex: Empty list creation.

```
list1= []
print(list1)
```

```
list2 = list()
print(list2)
```

```
list3 = list("ratan")
print(list3)
```

ex: forward & backward indexing

```
# -5 -4 -3 -2 -1
```

```
l1=[10,20,30,40,50]
# 0 1 2 3 4
```

```
print(l1[2])
print(l1[2:4])
print(l1[1:4:2])
print(l1[2:])
print(l1[:2])
print(l1[:])
#print(l1[8])      IndexError: list index out of range
```

```
print(l1[-3])
print(l1[-3:-1])
print(l1[-3:])
print(l1[:-3])
print(l1[:])
#print(l1[-9])      IndexError: list index out of range
```

ex: **printing list data by using for loop.**

```
l = ["ratan", "anu", "durga", "sunny"]
for x in l:
    print(x)
```

```
for x in l[1:3]:
    print(x)
```

```
for x in l[1:]:
    print(x)
```

```
for x in l[:3]:
    print(x)
```

```
for x in l[:]:
    print(x)
```

```
l1= ["mango", "blueberry", "apple"]
for x,y in enumerate(l1):
    print x,y)
```

ex: == is , is not in , not in

- ✓ `==` operator will check the data in the list not memory address returns Boolean value .
- ✓ To check the memory address use `is` & `is not` operator it returns Boolean value.
- ✓ To check the data is available or not use `in` & `not in` operators returns Boolean value.

```
l1 = [10,20,30]
l2=[40,50,60]
l3=l1
l4=[10,20,30]
```

```
print(id(l1))
print(id(l2))
print(id(l3))
print(id(l4))
```

```
print(l1 is l2)
print(l1 is l3)
print(l1 is not l2)
print(l1 is not l3)
```

```
print(l1==l2)
print(l1==l4)
print(l1!=l2)
print(l1!=l4)
```

```
print(10 in l1)
print(100 in l2)
print(10 not in l1)
print(100 not in l2)
```

ex 5: **unpacking the list data**

```
l1 = [10,"ratan",10.5]
a,b,c=l1
print(a,b,c)
print(type(a),type(b),type(c))
```

```
l1 = [10,"ratan",10.5,30]
a,b,c=l1
ValueError: too many values to unpack (expected 3)
```

ex: **Nested List data**

```
my_list = ["mouse", [1,2,3], ['a','b'], "anu"]
```

```
print(my_list)
```

```
# unpack the data
```

```
a,b,c=my_list
```

```
print(a)
```

```
print(b)
```

```
print(c)
```

```
print(type(a),type(b),type(c))
```

```
matrix = [[1,2,3],[4,5,6],[7,8,9]]
```

```
print(matrix[1])
```

```
print(matrix[1][1])
```

```
print(matrix[2][0])
```

ex:

```
x = [i for i in range(10,20)]
```

```
print(x)
```

```
y = [i*i for i in range(10)]
```

```
print(y)
```

```
z = [i+1 for i in range(10,20,2)]
```

```
print(z)
```

ex: list data is **mutable** it allows modifications.

- ✓ The **list.insert(i,x)** method takes two arguments, where *i* = index *x*= item .
- ✓ The method **list.append(x)** will add an item (*x*) to the end of a list.
- ✓ we can use **list.copy()** to make a copy of the list.
- ✓ We can combine the list by using **+** operator.
- ✓ If we want to combine more than one list, we can use the **list.extend(L)** method.

adding the data by using append() function

```
animal = ['rat','tiger']  
animal.append('cat')  
print(animal)
```

#adding the data by using insert()

```
animal = ['rat','tiger']  
animal.insert(2,'lion')  
animal.insert(0,'cow')  
animal.insert(6,'cat')  
print(animal)
```

#adding the data by using + operator

```
l1=[1,2,3]  
l2=[4,5,6]  
l3 = l1+l2  
print(l3)
```

#adding same data multiple times (Replication)

```
list_data=[10,20,30]  
x=list_data*2  
print(x)
```

#copy the data by using copy method

```
fruit=["apple","orange","grapes","banana"]  
fruitcopy = fruit.copy()  
print(fruitcopy)
```

#adding one list data into another list

```
a=[10,20,30]  
b=[10,40,50]  
a.extend(b)  
print (a)
```

ex: Inserting the data in list by using index.

```
animal = ['rat','tiger']
```

```
animal[1]="lion"  
print(animal)
```

```
animal[1:3]=["cat","rat","dog"]  
print(animal)
```

```
animal[2:4]=["ant"]  
print(animal)
```

```
animal[1:3]="Puppy"  
print(animal)
```

```
animal[2]=10  
print(animal)
```

```
animal[2:4]=100  
print(animal)                                TypeError: can only assign an iterable
```

ex:

- ✓ **List.index()** used to print the index of element. If the element is not available generates error : `ValueError: 'banana' is not in list`.
- ✓ The `reverse()` function is used to reverse the order of items.
- ✓ We can use the `list.sort()` method to sort the items in a list.
- ✓ The `list.count(x)` method will return the number of times the value `x` occurs within a specified list.

```
fruit=["apple","orange","grapes","orange"]  
print(fruit.index("orange"))  
print(fruit.index("orange",2))  
#print(fruit.index("banana"))                #ValueError: 'banana' is not in list
```

```
fruit.reverse()  
print(fruit)
```

```
fruit=["apple","orange","apple","banana"]  
print(fruit.count("apple"))  
fruit.sort()  
print(fruit)
```

ex:

- ✓ When we need to remove an item from a list, we'll use the **list.remove(x)** method which removes the first item in a list whose value is equivalent to x.
- ✓ If you pass an item in for x in list.remove() that does not exist in the list, you'll receive the following error: list.remove(x): x not in list
- ✓ We can use the **list.pop([i])** method to return the item at the given index position from the list and then remove that item.
- ✓ The square brackets around the i for index tell us that this parameter is optional, so if we don't specify an index (as in fish.pop()), the last item will be returned and removed.
- ✓ The del statement can also be used to remove slices from a list or clear the entire list
- ✓ By using clear() function we can clear the all the elements of the list.

Removing elements by using remove()

```
a=[10,20,30]
a.remove(10)
print(a)
#a.remove('ratan')    #ValueError: list.remove(x): x not in list
```

#Removing element by using pop() function

```
fruit=["apple","orange","grapes","banana","orange"]
print(fruit.pop())
print(fruit.pop(0))
print(fruit)
#print(fruit.pop(10)) # IndexError: pop index out of range
```

deleting element by using del

```
a = [10,20,30,40,50]
del a[0]
print(a)
del a[2:4]
print(a)
del a[:1]
print(a)
del a[1:]
print(a)
del a[: ]
print(a)
```

#clearing element by using clear() function

```
fruit=["apple","orange","apple","banana"]
print(fruit)
fruit.clear()
print(fruit)
```

ex: **Appending data to new list**

```
l1=[['a',2],['b',4],['c',6]]
```

```
#creating empty list
```

```
l2=list()
```

```
l3=list()
```

```
for i,j in l1:
```

```
    l2.append(i)
```

```
    l3.append(j)
```

```
print(l2)
```

```
print(l3)
```

ex: Adding data into list

```
l1=[10,20,30,40,50,60]
```

```
l2=[]
```

```
l3=[]
```

```
for x in l1:
```

```
    if x<30:
```

```
        l2.append(x)
```

```
    elif x>30:
```

```
        l3.append(x)
```

```
print(l2)
```

```
print(l3)
```

ex:

```
line = "hi ratan sir how are you"
```

```
words = line.split()
```

```
print(words)
```

```
print(type(words))
```

```
x = [[w.upper(),w.lower(),len(w)] for w in words]
```

```
print(x)
```

```
y = ['AnB','ABN','and','bDE']
```

```
print(sorted([i.lower() for i in y]))
```

```
print(sorted([i.lower() for i in y],reverse=True))
```

output :

```
['hi', 'ratan', 'sir', 'how', 'are', 'you']
```

```
<class 'list'>
```

```
[[['HI', 'hi', 2], ['RATAN', 'ratan', 5], ['SIR', 'sir', 3], ['HOW', 'how', 3], ['ARE', 'are', 3], ['YOU', 'you', 3]]
```

```
['abn', 'anb', 'and', 'bde']
```

```
['bde', 'and', 'anb', 'abn']
```

ex:

```
def append_to_sequence (myseq):  
    myseq += (9,9,9)  
    return myseq  
  
tuple1 = (1,2,3) # tuples are immutable  
list1 = [1,2,3] # lists are mutable  
tuple2 = append_to_sequence(tuple1)  
list2 = append_to_sequence(list1)  
print ('tuple1 = ', tuple1) # outputs (1, 2, 3)  
print ('tuple2 = ', tuple2) # outputs (1, 2, 3, 9, 9, 9)  
print ('list1 = ', list1) # outputs [1, 2, 3, 9, 9, 9]  
print ('list2 = ', list2) # outputs [1, 2, 3, 9, 9, 9]
```

Multidimensional Lists:

ex:

```
l = ["ratan" * 2] * 3
print(l)

print(l[0])
print(l[0][0])
print(l[0][1])

l[0].append("durga")
print(l)

lists = [[]] * 3
print(lists)
lists[0].append("durga")
print(lists)
```

*The reason is that replicating a list with * doesn't create copies, it only creates references to the existing objects. The *3 creates a list containing 3 references to the same list of length two. Changes to one row will effect in all rows, which is almost certainly not what you want.*

ex:

```
a = ["ratan"]*3
for x in range(3):
    a[x] = ["ratan"]*2

print(a)
print(a[0])
a[0].append("durga")
a[1].append("anu")
a[2].append("sunny")
print(a[0])
print(a[1])
print(a[2])
print(a)

w, h = 2, 3
l = ["ratan" * w for i in range(h)]
print(l)
```

Note: This generates a list containing 3 different lists of length two.

Tuple data type : (tuple)

- ✓ Tuple can store both homogeneous & heterogeneous data.
- ✓ Insertion order is preserved it means in which order we inserted data same order output is printed.
- ✓ Tuple allows duplicate objects.
- ✓ The tuple contains forward indexing & backward indexing.
- ✓ This is immutable modifications are not allowed.
- ✓ Declared within the parenthesis ()

Ex-1: Basic example: # comments are to understand the application

```
t1 = (10,20,30)      # tuple of integers (homogenous data)
print(t1)
```

```
t2 = (10,20.5,"ratan") # heterogeneous data
print(t2)
```

```
t3 = (10,20,10,20,30) # duplicates are allowed
print(t3)
```

```
t4 = (10)
print(type(t4))      #<class 'int'>
```

```
t5 = (10,)
print(type(t5))      #<class 'tuple'>
```

```
t6 = ()
print(t6)            # empty tuple
```

```
t7 = tuple()
print(t7)            # empty tuple
```

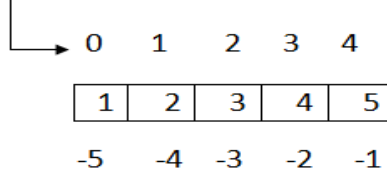
```
t8 = tuple("ratan")  # every character will become element of the tuple
print(t8)
```

when we declare the tuple parenthesis optional but recommended :

```
t1 = 10,20,30        valid : but not recommended
t2 = (10,20,30)       valid : recommended
```

Ex-2 : Indexing both forward & backward

Forward indexing



Backward indexing

```

t1=(10,20,30,40,50)
print(t1[0])
print(t1[2:4])
print(t1[1:4:2])
print(t1[:4])
print(t1[2:])
print(t1[:])
print(t1[2:10])
#print(t1[10])      IndexError: tuple index out of range

print(t1[-3])
print(t1[-4:-2])
print(t1[:-3])
print(t1[-3:])
print(t1[:])
#print(t1[-9])      IndexError: tuple index out of range

```

Ex 3: unpacking data

```

t = (10,20.5,"ratan")
a,b,c=t
print(a,b,c)
print(type(a),type(b),type(c))

# unpacking only single element
m = ( 'ratan', 'anu', 'durga')
y = m[1]
print(y)

t1 = (10,20,30)
a,b=t1
print(a,b)      ValueError: too many values to unpack (expected 2)

```

ex 4: Nested tuple

```
# 0      1
t = ((10,20) , (30,40,50))
# 0 1    0 1 2
```

```
print(t[0])
print(t[1])
```

```
print(t[1][1])
print(t[0][1])
print(t[1][2])
```

```
# unpacking nested tuple
a,b = t
print(type(a))
print(type(b))
```

ex 5: print the data by using for loop

```
t = (10,20,30,40)
for x in t:
    print(x)
```

```
for x in t[1:3]:
    print(x)
```

```
for x in t[1:1]:
    print(x)
```

```
t1 = ((10,20),(30,40))
for x,y in t1:
    print(x,y)
```

for loop can print only iterable data

```
for x in t[2]:
    print(x)                TypeError: 'int' object is not iterable
```

All the sub tuples must have same size

```
t2 = ((10,20),(30,40,50))
for x,y in t2:
    print(x,y)              ValueError: too many values to unpack (expected 2)
```

Ex 6: Adding multiple elements in tuple

```
l = [x for x in range(10)]          # working with list data
print(l)

t1 = tuple(x for x in range(10))     # working with tuple
print(t1)

t2 = tuple(x+267 for x in range(10,26))
print(t2)

t3 = tuple(x*89 for x in range(10,99,4))
print(t3)
```

Ex-7: Comparison operations

- ✓ *Printing memory address by using : id() function*
- ✓ *Is , is not : used for memory comparison*
- ✓ *== != : used for data comparison*
- ✓ *In , not in : checking data is available or not.*

```
t1 = (10,20,30)
t2 = (40,50,60)
t3 = t1
t4 = (10,20,30)
```

```
print(id(t1))
print(id(t2))
print(id(t3))
```

memory comparison

```
print(t1 is t2)
print(t1 is t3)
print(t1 is not t2)
print(t1 is not t3)
```

data comparison

```
print(t1==t2)
print(t1==t3)
print(t1==t4)
print(t1!=t2)
print(t1!=t3)
print(t1!=t4)
```

checking data available or not

```
print(10 in t1)
print(100 in t1)
```

```
print(10 not in t1)
print(100 not in t1)
```

Ex 8 : concatenation & replication

```
t1 = (10,20,30)
t2 = (40,50,60)
t3 = t1+t2          # after concat the data assigning to new reference variable
print(t3)
```

```
t4 = t1*3
print(t4)
```

```
t1 = (10,20,30)
t2 = (40,50,60)
t1 = t1+t2          # after concat the data assigning to same reference variables
print(t1)           # here the old reference variable pointing to new memory
                    # old object without reference so, that object destroyed.
```

Ex 9: conversion process

```
t = (10,20,30)      # initially : we have tupe

l = list(t)          # convert tuple into list : do the modifications
l.append(40)
l.insert(2,66)
print(l)

t1 = tuple(l)        # after modifications convert list into tuple
print(t1)

tup = ('r','a','t','a','n') # joining the data
str1 = '+'.join(tup)
str2 = ''.join(tup)
print(str1)
print(str2)
```

ex-10: Appending data & copy the data

```
t1 = ("HELLO", 5, [], True)
t1[2].append(100)
print(t1)
print(id(t1))

from copy import deepcopy
t2 = ("HELLO", 5, [], True)
t2= deepcopy(t2)
t2[2].append(50)
```

```
print(id(t2))
print(t2)
```

Ex-11: count(), index() len () function

```
t = (10,10,10,20,30,40)
print(t.count(10))

print(t.index(10))
print(t.index(10,2))
print(t.index(10,2,4))
# print(t.index(10,3,5))      ValueError: tuple.index(x): x not in tuple
```

Ex 12: min & max values only on homogenous.

```
t1 = (10,20,30)
print(min(t1))
print(max(t1))

t2 = ("ratan","durga","anu")
print(min(t2))
print(max(t2))

t3 = [10,"ratan"]      # min & max not possible in heterogeneous data
print(min(t3))
print(max(t3))      TypeError: '<' not supported between instances of 'str' and 'int'
```

ex 13 : Assignment example

```
t = ((1,2),("ratan","anu"),(4,5))
l1 = []
l2 = []
for x,y in t:
    l1.append(x)
    l2.append(y)

print(l1)
print(l2)
```

Ex-14 : sorting the data by using sort() & sorted() function.

sort() function : present in only list : l.sort() : general sorting purpose

sorted() function : present in both list & tuple : print(sorted(l)) : To generate the report

```
l1 = [4,2,5,7]
print(sorted(l1))
l2 = [4,2,5,7]
print(sorted(l2,reverse=True))

t1 = (10,4,5,100)
print(sorted(t1))
t1 = (10,4,5,100)
print(sorted(t1,reverse=True))

t = (30,1,5,98)
l = list(t)           # conversion of tuple to list
l.sort()              # sorting the list data by using sort function
t1 = tuple(l)         # converting list to tuple
print(l)
print(t1)

t = (30,1,5,98,"ratan")
print(sorted(t))      # TypeError: '<' not supported between instances of 'str' and 'int'
```

Ex 15: Assignment: understand the example: write the output

```
txt = "hi rattan sir what about python"
words = txt.split( )      # default splitting character space
t = [ ]
# taking list of tuple data : [(length,word),(length,word)]
for word in words:
    t.append((len(word), word))
t.sort(reverse=True)      # data in descending order
#t.sort()                 :# data in Ascending order
print(t)

# taking only list of words
res1 = [ ]
for length,word in t:
    res1.append(word)
print (res1)
# taking only length of words
res2 = [ ]
for length, word in t:
    res2.append(length)
print (res2)
```

***** Tuple completed : Thank you *****

Set Data Type

- ✓ Set is used to store the group of unique object (duplicates are not allowed).
- ✓ Declare the set data by using curly braces { }.
- ✓ Set is mutable allows modifications.
- ✓ Set insertion order is not preserved, indexing is not supported.

ex-1:

set of integers

```
my_set = {1, 2, 3}
print(my_set)
print(type(my_set))
```

set of mixed datatypes

```
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

set duplicates not allowed

```
my_set = {1,2,3,1,2,3}
print(my_set)
print(len(my_set))
```

#creates empty set : but it is empty dictionary but not set

```
s={}
print(s)
print(type(s))          # <class, 'dict'>
```

#creates empty set

```
s=set()
print(type(s))          #<class 'set'>
```

- ✓ In set it is possible to insert only immutable data : String , Number , Tuple
- ✓ In set it is not possible to insert mutable data like set,list ,dict , if we are trying to insert mutable data we will get Type error.

○ `s = { [1,2,3], "ratan" }` `TypeError: unhashable type: 'list'`

○ `s = { {1,2,3}, "ratan" }` `TypeError: unhashable type: 'set'`

○ `s=set()`
`s.add({"ratan":111})`
`print(type(s))` `TypeError: unhashable type: 'dict'`

ex-2: set is mutable allows modifications.

set of integers

```
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)
```

adding list and set

```
my_set = {1, 2, 3}
my_set.update([4,5], {1,6,8})
print(my_set)
```

#copy the set data

```
my_set = {1,2,3}
a= my_set.copy()
print(a)
```

#concatenation & replication not possible

```
s1={10,20,30} s2={30,40,50}
s3=s1*2      Type Error: unsupported operand type(s) for *: 'set' and 'int'
s3 =s1+s2    Type Error: unsupported operand type(s) for +: 'set' and 'set'
```

ex:

- ✓ A particular item can be removed from set using functions , `discard()` and `remove()`.
 - `discard()`** : it removes the data if the data is not available no error .
 - `remove()`** : it removes the data if the data is not present we will get **keyError**
- ✓ we can remove and return an item using the **`pop()`** function. Set unordered, there is no way of determining which item will be popped. It is completely arbitrary.
- ✓ We can also remove all items from a set using `clear()` function.

```
my_set = {1,2,3,4,5}
my_set.discard(4)
my_set.discard(7)    # if the data not present : no error
print(my_set)
```

```
my_set = {1,2,3,4,5}
my_set.remove(5)
#my_set.remove(6)    #KeyError: 6 if the key not present
print(my_set)
```

#pop takes random element

```
my_set = {1,2,3,4,5}
print(my_set.pop())
print(my_set.pop())
print(my_set)
```

Removes all elements

```
my_set = {1,2,3,4,5}
```

```
my_set.clear()
```

```
print(my_set)
```

ex 3:

- ✓ To print the id of the dictionary **id()** function, it print memory address.
- ✓ To check the memory address use **is** & **is not** operator.
 - If two references are pointing to same memory returns true otherwise false.
- ✓ **==** operator will check the data in set not memory address.
 - If the data same returns true otherwise false.
- ✓ To check the data is available or not use **in** & **not** in operators.
 - If the data present in set return true otherwise returns false.

```
s1={10,20,30,40}
```

```
s2={10,20,30,40}
```

```
s3={"ratan","anu"}
```

```
s4=s3
```

```
print(id(s1))
```

```
print(id(s2))
```

```
print(id(s3))
```

```
print(id(s4))
```

```
print(s1 is s2)
```

```
print(s3 is s4)
```

```
print(s1 is not s2)
```

```
print(s3 is not s4)
```

```
print(s1==s2)
```

```
print(s1==s3)
```

```
print(s1!=s2)
```

```
print(s1!=s3)
```

```
print(10 in s1)
```

```
print(100 in s1)
```

```
print(10 not in s1)
```

```
print(100 not in s1)
```

ex: unpacking the set data

```
s1 = {10,"ratan",(10,20)}  
a,b,c=s1  
print(a,b,c)  
print(type(a),type(b),type(c))
```

```
s2 = {1,2,3}  
a,b=s2  
print(a,b)
```

ValueError: too many values to unpack (expected 2)

ex: Iterating data by using for loop.

```
s = {"apple","banana"}  
for fruit in s:  
    print(fruit)
```

```
for letter in set("apple"):  
    print(letter)
```

ex: Adding group of values into set.

```
x = {i for i in range(10,20)}  
print(type(x))  
print(x)
```

```
y = {i*i for i in range(10)}  
print(type(y))  
print(y)
```

```
z = {i for i in range(10,20,2)}  
print(type(z))  
print(z)
```

ex: creating new set by adding list data with some condition.

```
l1=[10,20,30,40,50]  
l2=[1,2,3,4,5]
```

```
s = set()  
for x in l1:  
    if(x>30):  
        s.add(x)
```

```
for y in l2:  
    if(y<3):  
        s.add(y)
```

```
print(s)
```

ex: **Eliminating duplicates from list**

```
l1=["ratan",10,20,10,20,"ratan",10,20,"anu"]
```

```
s1 = set(l1)
```

```
print(s1)
```

```
l2 = list(s1)
```

```
print(l2)
```

ex: **#creating set by passing more than one list data**

```
l1 = ['ratan', 'anu', 'durga']
```

```
l2 = ['Jack', 'Sam', 'Susan']
```

```
l3 = ['Jane', 'Jack', 'Susan']
```

```
engineers = set(l1+l2+l3)
```

```
print(engineers)
```

#creating set by using list data

```
engineers = set(['ratan', 'anu', 'durga'])
```

```
programmers = set(['Jack', 'Sam', 'Susan'])
```

```
managers = set(['Jane', 'Jack', 'Susan'])
```

```
employees = engineers | programmers | managers
```

```
print(employees)
```

ex:

```
a= set("ratan")
```

```
print(a)
```

```
b= set("ratansoft")
```

```
print(b)
```

```
print(b-a)    # letters in b but not in a
```

```
print(a|b)    # letters in a or b or both
```

```
print(a&b)    # letters in both a and b
```

```
print(a^b)    # letters in a or b but not both
```

```
basket1 = {'orange', 'apple', 'pear', 'banana'}
```

```
basket2 = {'pear', 'orange', 'banana'}
```

```
print(basket1-basket2)
```

```
print(basket1|basket2)
```

```
print(basket1&basket2)
```

```
print(basket1^basket2)
```

ex: **Checking predefined functions of Set data type.**

```
x = dir(set)
print(x)
```

Dictionary data type

- ✓ List, tuple, set data types are used to represent individual objects as a single entity.
- ✓ **To store the group of objects as a key-value pairs use dictionary.**
- ✓ Declare the dictionary data within the curly braces, each key separated with value by using : (colon) & each element is separated with comma.
- ✓ A dictionary is a data type similar to arrays, but works with keys & values instead of indexes.
- ✓ **The keys must be unique keys but values can be duplicated.**
- ✓ The values in the dictionary can be of any type while the keys must be immutable like numbers, tuples or strings.
- ✓ Dictionary keys are case sensitive- Same key name but with the different case are treated as different keys in Python dictionaries.

```
d = {"ratan":1, "RATAN":2}
```

ex-1:

```
phonebook = {}
print(phonebook)
phonebook["ratan"] = 9355775660
phonebook["durga"] = 9476655551
print(phonebook)
```

```
phonebook1 = { "ratan" : 935577566, "anu" : 936677884}
print(phonebook1)
```

ex -2:

empty dictionary

```
my_dict = {}
print(my_dict)
```

empty dictionary

```
x = dict()
print(x)
```

dictionary with integer keys

```
my_dict = {1: 'apple', 2: 'ball'}
print(my_dict)
```

keys are case sensitive

```
d = {"ratan":1, "RATAN":2}
print(d)
```

dictionary with mixed keys

```
my_dict = {'name': 'John', 1: [2, 4, 3]}
```

```
print(my_dict)
```

```
d1 = {[1,2,3]:"ratan"} #TypeError: unhashable type: 'list'
```

ex-3 :

we can read the data from dictionary in two ways

- a. By using keys : If the key is not present generate **KeyError**
- b. By using get method : If the key not present return **None**

```
friends = {'tom':'111-222-333','jerry':'666-33-111'}  
print(friends)
```

```
print(friends['tom'])  
print(friends['jerry'])
```

```
print(friends.get("tom"))  
print(friends.get('jerry'))
```

```
print(friends.get("ratan")) #none  
print(friends['ratan']) #KeyError: 'ratan'
```

ex-4:

- ✓ To get all the keys from dictionary use : **keys()** function.
- ✓ To get all the values from dictionary use : **values()** function.
- ✓ To print the data in sorting order use : **sorted()** function.

```
d1={1:"ratan",2:"anu",4:"durga",3:"aaa"}
```

```
print(d1.keys())  
print(d1.values())
```

```
print(list(d1.keys()))  
print(list(d1.values()))
```

```
print(tuple(d1.keys()))  
print(tuple(d1.values()))
```

```
print(set(d1.keys()))  
print(set(d1.values()))
```

```
print(sorted(d1.keys()))  
print(sorted(d1.values()))
```

ex-5:

- ✓ To print the id of the dictionary **id()** function, it print memory address.
- ✓ To check the memory address use **is** & **is not** operator.
 - If two references are pointing to same memory returns true otherwise false.
- ✓ **==** operator will check the data in dictionary(key based) not memory address.
 - If the data same returns true otherwise false.
- ✓ To check the data is available or not use **in** & **not** in operators.
 - **in** and **not in** operators check whether key exists in the dictionary but no values.
 - If the data present in dictionary return true otherwise returns false.

```
d1 = {1:"ratan",2:"anu"}
d2 = {3:"aaa",4:"bbb"}
d3=d1
d4={1:"ratan",2:"anu"}
```

```
print(id(d1))
print(id(d2))
print(id(d3))
```

```
print(d1 is d2)
print(d1 is d3)
```

```
print(d1 is not d2)
print(d1 is not d3)
```

```
print(d1==d2)
print(d1==d4)
```

```
print(d1!=d2)
print(d1!=d4)
```

```
print(1 in d1)
print("ratan" in d1)
```

```
print(1 not in d1)
print("ratan" not in d1)
```

ex-6: printing data by using for loop

```
d1 = {1:"ratan",2:"anu",3:"durga"}
```

```
for i in d1:
    print(i,d1[i])
```

```
print(d1.items())
```

```

for k,v in d1.items():
    print(k,v)
ex-7:  del[key]      :      It removes the item based on input key.  : no key : KeyError
       pop[key]     :      It removes the item based on input key.  : no key : KeyError
       popitem()    :      It removes last element of the dictionary. : no item : KeyError
       clear()      :      It removes all items from the dictionary.

```

```

phonebook = {"ratan":938477566,"anu":938377264,"durga":947662781,"sunny":909090909}
del phonebook["durga"]
phonebook.pop("anu")
phonebook.popitem() # it removes last element of the dictionary
print(phonebook)
phonebook.clear()
print(phonebook)

```

```

# del phonebook["xxx"]  KeyError: 'xxx'
#phonebook.pop("xxx")  KeyError: 'xxx'
#phonebook.popitem()  KeyError: 'popitem(): dictionary is empty'

```

ex-8:

```

mydict = {"ratan":28,"anu":25,"durga":30}
#Updating data
mydict["durga"]=35
print(mydict)

```

#Adding one dictionary data into another

```

d1={"1":"ratan",2:"anu"}
d2={"3": "durga",4:"aaa"}
d1.update(d2)
print(d1)

```

#creating new dictionary

```

d1 = {"1":"ratan",3:"anu"}
d2 = {"2":"aaa",4:"bbb"}
x = {**d1,**d2}
print(x)

```

#Copying Dictionary data

```

d1={"1":"ratan",2:"anu",3:"durga",4:"aaa"}
d2 = d1.copy()
print(d2)

```

#length of the dictionary

```

d1 = {"1":"ratan",2:"anu",3:"durga"}
print(len(d1))

```



```
print(len(d1.keys()))
print(len(d1.values()))
```

ex-9:

creating new dictionary by using list data

```
l1=[10,20,30,40]
l2=["ratan","anu","durga","aaa"]
x = zip(l1,l2)
d = dict(x)
print(d)
```

creating new dictionary by using tuple data

```
l1=(10,20,30,40)
l2=("ratan","anu","durga","aaa")
x = zip(l1,l2)
d = dict(x)
print(d)
```

creating new dictionary by using set data

```
l1={10,20,30,40}
l2={"ratan","anu","durga","aaa"}
x = zip(l1,l2)
d = dict(x)
print(d)
```

ex-11:

```
d1={10:"ratan",20:"anu"}
d2={1:"aaa",2:"bbb"}
```

```
l1 = list(d1.keys())
l2 = list(d2.values())
```

```
x = zip(l1,l2)
d = dict(x)
print(d)
```

ex-12:

#creating dictionary with list of keys with same value

```
keys = ["bird", "plant", "fish"]
d = dict.fromkeys(keys, 5)
print(d)
```

#creating dictionary by using items

```
pairs = [("cat", "meow"), ("dog", "bark"), ("bird", "chirp")]
```

```
lookup = dict(pairs)
print(lookup)
print(lookup.items())
```

ex:

```
v1 = int(2.7) # 2
v2 = int(-3.9) # -3
v3 = int("2") # 2
v4 = int("11", 16) # 17, base 16
v5 = long(2)
v6 = float(2) # 2.0
v7 = float("2.7") # 2.7
v8 = float("2.7E-2") # 0.027
v9 = float(False) # 0.0
vA = float(True) # 1.0
vB = str(4.5) # "4.5"
vC = str([1, 3, 5]) # "[1, 3, 5]"
vD = bool(0) # False; bool fn since Python 2.2.1
vE = bool(3) # True
vF = bool([]) # False - empty list
vG = bool([False]) # True - non-empty list
vH = bool({}) # False - empty dict; same for empty tuple
vI = bool("") # False - empty string
vJ = bool(" ") # True - non-empty string
vK = bool(None) # False
vL = bool(len) # True
```

Python operators

- ✓ Python Arithmetic Operators.
- ✓ Python Comparison/relational Operators
- ✓ Python Logical Operators
- ✓ Python Assignment Operators
- ✓ Python Identity Operators
- ✓ Python Membership Operators
- ✓ Python Bitwise Operators

✓ Arithmetic operators

Arithmetic Operators

Operator	Meaning	Example
+	Addition	$4 + 7 \longrightarrow 11$
-	Subtraction	$12 - 5 \longrightarrow 7$
*	Multiplication	$6 * 6 \longrightarrow 36$
/	Division	$30 / 5 \longrightarrow 6$
%	Modulus	$10 \% 4 \longrightarrow 2$
//	Quotient	$18 // 5 \longrightarrow 3$
**	Exponent	$3 ** 5 \longrightarrow 243$

```

ex :   x ,y= 15,4
        # Output: x + y = 19
        print('x + y =',x+y)

        # Output: x - y = 11
        print('x - y =',x-y)

        # Output: x * y = 60
        print('x * y =',x*y)

        # Output: x / y = 3.75
        print('x / y =',x/y)

        # Output: x // y = 3
        print('x // y =',x//y)

        # Output: x ** y = 50625
        print('x ** y =',x**y)

```

Python Mixed-Mode Arithmetic

The calculation which done both integer and floating-point number is called mixed-mode arithmetic. When each operand is of a different type.

$9/2.0 \rightarrow 4.5$

✓ **Relational operators**

Relational Operators

Operators	Meaning	Example	Result
<	Less than	$5 < 2$	False
>	Greater than	$5 > 2$	True
<=	Less than or equal to	$5 <= 2$	False
>=	Greater than or equal to	$5 >= 2$	True
==	Equal to	$5 == 2$	False
!=	Not equal to	$5 != 2$	True

$x = 5$

$y = 2$

Output: $x > y$ is False

`print('x > y is', x > y)`

Output: $x < y$ is True

`print('x < y is', x < y)`

Output: $x == y$ is False

`print('x == y is', x == y)`

Output: $x != y$ is True

`print('x != y is', x != y)`

Output: $x >= y$ is False

`print('x >= y is', x >= y)`

Output: $x <= y$ is True

`print('x <= y is', x <= y)`

✓ **Logical operator :**

Logical operators

```
>>> a, b, c = 10, 20, 30
```

```
>>> (a > b) and (b < c)  
False
```

```
>>> (a < b) and (b < c)  
True
```

```
>>> (a > b) or (b < c)  
True
```

Operator	Description
a and b	Logical AND If both operands are True then it returns True
a or b	Logical OR If one of the operands is True then it returns True
not	Logical NOT

Example :

```
x = True
```

```
y = False
```

```
# Output: x and y is False  
print('x and y is',x and y)
```

```
# Output: x or y is True  
print('x or y is',x or y)
```

```
# Output: not x is False  
print('not x is',not x)
```

✓ **Assignment operator :**

Operator	Description
=	$x=y$, y is assigned to x
+=	$x+=y$ is equivalent to $x=x+y$
-=	$x-=y$ is equivalent to $x=x-y$
=	$x=y$ is equivalent to $x=x*y$
/=	$x/=y$ is equivalent to $x=x/y$
=	$x=y$ is equivalent to $x=x**y$

Assignment operators in Python

Operator	Example	Equivalent to
=	<code>x = 5</code>	<code>x = 5</code>
+=	<code>x += 5</code>	<code>x = x + 5</code>
-=	<code>x -= 5</code>	<code>x = x - 5</code>
*=	<code>x *= 5</code>	<code>x = x * 5</code>
/=	<code>x /= 5</code>	<code>x = x / 5</code>
%=	<code>x %= 5</code>	<code>x = x % 5</code>
//=	<code>x //= 5</code>	<code>x = x // 5</code>
**=	<code>x **= 5</code>	<code>x = x ** 5</code>
&=	<code>x &= 5</code>	<code>x = x & 5</code>
=	<code>x = 5</code>	<code>x = x 5</code>
^=	<code>x ^= 5</code>	<code>x = x ^ 5</code>
>>=	<code>x >>= 5</code>	<code>x = x >> 5</code>
<<=	<code>x <<= 5</code>	<code>x = x << 5</code>

✓ **Identity operators**

Operator	Meaning
is	True if the operands are identical (refer to the same object)
is not	True if the operands are not identical (do not refer to the same object)

```
x1 , y1 = 5,5  
x2 , y2 = "ratan","ratan"  
x3 , y3= [1,2,3],[1,2,3]
```

```
print(id(x1))      #20935504  
print(id(y1))      #20935504  
print(x1 is y1)  
print(x1 is not y1)
```

```
print(id(x2))      #21527296  
print(id(y2))      #21527296  
print(x2 is y2)  
print(x2 is not y2)
```

```
print(id(x3))      #45082264  
print(id(y3))      #45061624  
print(x3 is y3)  
print(x3 is not y3)
```

ex: a=10
 b=15

```
x=a  
y=b  
z=a
```

```
print(x is y)  
print(x is a)
```

```
print(y is b)
print(x is not y)
print(x is not a)
print(x is z)
```

✓ **Membership operators:**

in: "in" operator return true if a character or the entire substring is present in the specified string, otherwise false.

not in: "not in" operator return true if a character or entire substring does not exist in the specified string, otherwise false.

```
str1="ratanit"
str2="durgasoft"
str3="ratan"
str4="durga"
print(str3 in str1)           # True
print(str4 in str2)           # True
print(str3 in str2)           # False
print("ratan" in "ratanit")   #true
print("ratan" in "durgasoft") #False
```

```
print(str3 not in str1)       # False
print(str4 not in str2)       # False
print(str3 not in str2)       # True
print("ratan" not in "ratanit") # false
print("ratan" not in "anu")    # true
```


✓ **Bitwise operator: -**

Operator	Description
	Perform binary OR operation
&	Perform binary AND operation
~	Perform binary XOR operation
^	Perform binary one's Complement operation
<<	Left shift operator, left side operand bit is moved left by numeric number specified in right side
>>	Right shift operator, left side operand bit is moved right by numeric number specified in right side

& operator :**print(3&7)**

0011

0111

0011

print(15&15)

1111

1111

1111

print(9&6)

1001

0101

0000

print(0&0)

0000

0000

0000

/ operator :**print(3/7)**

0011

0111

0111

print(15/15)

1111

1111

1111

print(9/6)

1001

0101

1111

print(0/0)

0000

0000

0000

Python Class concept

- ✓ *Class is a logical entity grouping functions and data as a single unit, where as object is the physical entity represent memory.*
- ✓ *Class is a blue print it decides object creation.*
- ✓ *Based on single class possible to create multiple objects but every object occupies memory.*
- ✓ *In python define the class by using class keyword.*

```
class MyClass:  
    pass
```

Python 2.7 : *our class not extends any other classes*

```
class MyClass:
```

Python 3.x : *our class is by default child class of object*

The three declarations are valid & same:

```
class MyClass:  
class MyClass():  
class MyClass(object):
```

Class elements: *The class contains*

Variables

Methods (functions)

```
Constructors    :    __init__()  
__str__()  
__del__()
```

Note : *To represent the method,constructor , __str__ (),__del__ () is belongs to particular class must pass self as a first argument.*

Note : ***self** represents the instance of the class.*

Ex - 1:

In python 2.7 by default our class not extending object class

```
class MyClass1:
    pass
class MyClass2():
    pass
class MyClass3(object):
    pass

print(issubclass(MyClass1,object))    # False
print(issubclass(MyClass2,object))    # False
print(issubclass(MyClass3,object))    # True
```

python 3.x : our class is default super class is object

```
class MyClass1:
    pass
class MyClass2():
    pass
class MyClass3(object):
    pass

print(issubclass(MyClass1,object))    # True
print(issubclass(MyClass2,object))    # True
print(issubclass(MyClass3,object))    # True
```

Ex-2: Declaring Functions inside the class.

```
class MyClass:
    def disp1(self):
        print("Good Morning")

    def disp2(self,name):
        print("Good Evening:",name)

c = MyClass()
c.disp1()
c.disp2("Ratan")
```

Ex - 3 : Declaring variables inside the class

- ✓ Inside the class to use class variables always use : **self**
- ✓ Outside of the class to access class variables use : **object name**

```
class MyClass():  
    a,b=10,20  
    def add(self):  
        print(self.a+self.b)  
  
    def mul(self):  
        print(self.a*self.b)
```

```
c = MyClass()  
c.add()  
c.mul()  
print(c.a+c.b)  
print(c.a*c.b)
```

Ex - 4: local variables vs. class variables vs. global variables

```
a,b=100,200  
class MyClass():  
    a,b=10,20  
    def add(self,a,b):  
        print(a+b)  
        print(globals()['a']+globals()['b'])  
        print(self.a+self.b)  
  
    def mul(self,a,b):  
        print(a*b)  
        print(globals()['a']+globals()['b'])  
        print(self.a*self.b)
```

```
c = MyClass()  
c.add(3,3)  
c.mul(4,4)
```

Ex-5: There are two formats of object creation,

- Named object
- Name less object

```
class MyClass:
    def disp(self):
        print("Good Morning")
```

```
#named object approach
c = MyClass()
c.disp()
```

```
#Name less object approach
MyClass().disp()
```

Ex-6:

- ✓ **For the single class possible to create more than object, every object occupies memory.**

```
class MyClass():
    def disp(self):
        print("Good morning")
```

```
#first object creation
c1 = MyClass()
c1.disp()
#second object creation
c2 = MyClass()
c2.disp()
```

- ✓ **Modifications done by one object not reflected on another object.**

```
class MyClass():
    name="ratan"

#first object creation
c1 = MyClass()
c1.name="durga"
print(c1.name)      # durga
#second object creation
c2 = MyClass()
print(c2.name)      # ratan
```

Ex-7: Instance methods & static method.

- ✓ Access the instance methods by using object name. Represent by using **self**.
- ✓ Access the static methods by using class-name. Here self not required.
- ✓ Represent method is a static by using @staticmethod qualifier.

```
class MyClass():
    def disp1(self):
        print("Good morning")

    @staticmethod
    def disp2():
        print("Good evening")
    @staticmethod
    def disp3(name):
        print("Good Night:",name)

#Accessing instance method
c = MyClass()
c.disp1()

# Accessing static method
MyClass.disp2()
MyClass.disp3("ratan")
```

Ex-8 : Object ID:

- ✓ Every object has an identity, a type and a value. An object's identity never changes once it has been created, you may think of it as the object's address in memory.
- ✓ To get the id of the object use id() function it returns an integer representing its identity.
- ✓ The 'is' , 'is not' operators compares the identity of two objects.

```
class MyClass():
    pass

c1 = MyClass()
c2 = MyClass()
c3 = c1
print(id(c1))
print(id(c2))
print(id(c3))

print(c1 is c2)      #False
print(c1 is c3)      #True
print(c1 is not c2)   # True
print(c1 is not c3)   # False
```

Ex -9 : conversion of local variables to class level variables

```
class MyClass():
    def values(self,val1,val2):
        print(val1)
        print(val2)
        #conversion of local to class-values
        self.val1=val1
        self.val2=val2

    def add(self):
        print(self.val1+self.val2)

    def mul(self):
        print(self.val1*self.val2)

c = MyClass()
c.values(3,3)
c.add()
c.mul()
```

ex: Constructor: **`__init__()`**

- ✓ Constructors are used to write the logics these logics are executed during object creation.
- ✓ Constructors are used to initialize the values to variables during object creation.
- ✓ The constructor arguments are local variables.
- ✓ To make the local variables to global variables use self keyword. (self.eid=eid)

ex:

```
class MyClass:
    def __init__(self):
        print("constructor")
```

```
a = MyClass()
```

ex: constructor arguments are local variables

```
class MyClass:
    a,b=10,20
    def __init__(self,a,b):
        print(a+b)
        print(self.a+self.b)

    def add(self,a,b):
        print(a+b)
        print(self.a+self.b)
```

```
c = MyClass(2,2)
c.add(3,3)
```

ex: Conversion process : conversion of local to class level values

```
class MyClass:
    def __init__(self, val1, val2):
        print(val1)
        print(val2)
        #make local to class level values
        self.val1=val1
        self.val2=val2

    def add(self):
        print(self.val1+self.val2)

    def mul(self):
        print(self.val1*self.val2)
```

```
c = MyClass(10,20)
c.add()
c.mul()
```

ex :

```
class Student:
    def __init__(self, rollno, name):
        self.rollno = rollno
        self.name = name

    def displayStudent(self):
        print ("rollno : ",self.rollno )
        print("name: ",self.name)
```

```
student1 = Student(111, "ratan")
student1.displayStudent()
```

```
student2 = Student(222, "anu")
student2.displayStudent()
```

ex :

```
class Greeting:
    msg = 'morning'
    def __init__(self, name):
        self.name = name;

r = Greeting('ratan')
a = Greeting('anu')
```



```
print( r.msg , r.name)
print (a.msg , a.name)
```

ex: `__init__()` executed when we create the object.
`__str__()` executed when we print reference variable it returns always String only.

case 1:

```
class MyClass:
    pass
c = MyClass()
print(c)          # <MyClass object at 0x7f89d1115438>
```

case 2:

```
class Test:
    def __str__(self):
        return "ratanit.com"
t = Test();
print(t)          #ratanit.com
```

case 3:

```
class Test:
    def __str__(self):
        return 10
t = Test()
print(t)          #TypeError: __str__ returned non-string (type int)
```

ex:

```
class Emp:
    def __init__(self,eid,ename,esal):
        self.eid=eid
        self.ename=ename
        self.esal=esal

    def __str__(self):
        return "emp id=%d Emp name=%s Emp sal=%g"%(self.eid,self.ename,self.esal)

e1 = Emp(111,"ratan",100000.45)
print(e1)

e2 = Emp(111,"anu",200000.46)
print(e2)
```

```
ex : class Pet(object):
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def getName(self):
        return self.name
    def getSpecies(self):
        return self.species

a = Pet("durga", "human")
print("Name=", a.getName())
print("species=", a.getSpecies())
```

```
ex:
class Emp:
    def setEid(self, eid):
        self.eid = eid
    def setEname(self, ename):
        self.ename = ename

    def getEid(self):
        return self.eid
    def getEname(self):
        return self.ename;

class Client:
    e = Emp()
    e.setEid(111)
    e.setEname("ratan")

    print(e.getEid())
    print(e.getEname())
```

```
ex : class Employee:
    empCount = 0
```

```

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

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

def __str__(self):
    return "Name : {0} Salary: {1}".format(self.name,self.salary)

emp1 = Employee("ratan", 20000)
print(emp1)
emp2 = Employee("anu", 5000)
print(emp2)

emp2.displayCount()

```

Destructors in python:

- ✓ To destroy the object use **del**
- ✓ Destructors are called when an object gets destroyed, it is opposite to constructors.
- ✓ When we destroy the object the **__del__()** function executed.
- ✓ When multiple reference variables are pointing to same object , if all reference variable count will become zero then only **__del__()** will be executed.

ex:

```

class Vehicle:
    def __init__(self):
        print("Vehicle object created")

    def __del__(self):
        print("Vehicle object destroyed")

v = Vehicle()
del v

```

ex:

```

class MyClass:
    def __del__(self):

```

```
print("object destroyed")
```

```
c1 = MyClass()
```

```
c2= MyClass()
```

```
del c1
```

```
del c2
```

ex: When multiple reference variables are pointing to same object, if all reference variable count will become zero then only `__del__()` will be executed.

```
class MyClass:
```

```
    def __del__(self):
```

```
        print("object destroyed")
```

```
c1 = MyClass()
```

```
c2= c1
```

```
c3 = c1
```

```
del c1
```

```
del c2
```

```
del c3
```

ex: If any exceptions are raised in `__del__()` these are ignored object destroyed

```
class Vehicle:
```

```
    def __del__(self):
```

```
        print("Vehicle object destroyed")
```

```
        print(10/0)
```

```
v = Vehicle()
```

```
del v
```

```
E:\>python first.py
```

```
Vehicle object destroyed
```

```
Exception ignored in: <bound method Vehicle.__del__ of <__main__. Vehicle object at 0x000000B8087580B8>>
```

```
ZeroDivisionError: division by zero
```

ex:

```
class MyClass:
```

```
    a=10
```

```
    def m1(self):
```

```
        print("m1 function")
```

```
def __init__(self):
    print("constructor")
def __str__(self):
    return "ratanit.com"
def __del__(self):
    print("object destroyed....")

c = MyClass()          # constructor __init__() executed
print(c.a)             # variables are printed
c.m1();                # method executed
print(c)               # __str__() executed
del c                  # __del__() executed

ex:
class Customer:
    def __init__(self,name,bal=0.0):
        self.name=name
        self.bal=bal

    def deposit(self,amount):
        self.bal=self.bal+amount

    def withdraw(self,amount):
        if amount>self.bal:
            raise RuntimeError("withdraw amount is more than balance")
        else:
            self.bal=self.bal-amount

    def remaining(self):
        return self.bal;

c = Customer("ratan",10000)
damt = int(input("enter amount to deposit"))
c.deposit(damt)

amt = int(input("enter amount to withdraw"))
c.withdraw(amt)

print(c.remaining())
```

Class Attribute:

Hasattr() : To check the attribute present in the class or not. If the attribute present in the class it returns true otherwise false.

```
class Student:
    def __init__(self, rollno, name):
        self.rollno = rollno
        self.name = name
```

```
Student1 = Student(111, "ratan")
Student2 = Student(222, "anu")
print(hasattr(Student1, "rollno"))
print(hasattr(Student1, "name"))
print(hasattr(Student1, "age"))
```

Ex:

```
class Student:
    def __init__(self, rollno, name):
        self.rollno = rollno
        self.name = name
```

```
Student1 = Student(111, "ratan")
```

```
print(hasattr(Student1, "age"))
setattr(Student1, "age", 25)
print(hasattr(Student1, "age"))
print(getattr(Student1, "age"))
delattr(Student1, "age")
print(hasattr(Student1, "age"))
```

Inheritance

✓ The process of acquiring properties from parent to child class is called inheritance.

Classic class declaration not inheriting from any class.

```
class MyClass:                                # Don't inherit from anything.
```

While new-style classes inherit from either object or some other class.

```
class MyClass(object):                        # Inherit from object, new-style class.
```

Single inheritance: is when each class inherits from exactly one parent, or super class

```
>class A(object): pass                       object
> class B(A): pass                           |
                                           A
```

Multilevel inheritance

```
>class A(object): pass                       object
> class B(A): pass                           |
> class C(B): pass                           A
                                           |
                                           B
                                           |
                                           C
```

hierarchical inheritance

```
>class A(): pass                             A
> class B(A): pass                           / \
> class C(A): pass                           B  C
```

multiple inheritance :

at least one of the classes involved inherits from two or more super classes. Here's a particularly simple example:

```
> class A(object): pass                       object
> class B(A): pass                           |
> class C(A): pass                           A
> class D(B, C): pass # multiple superclasses / \
                                           B  C
                                           \ /
                                           D
```

ex:

```
class Parent:
    pass
class Child(Parent):
    pass

p = Parent()
c = Child()

print(isinstance(p,object))
print(isinstance(p,Parent))
print(isinstance(c,Child))
print(isinstance(c,object))
print(isinstance(p,Child))

print(issubclass(Parent,object))
print(issubclass(Child,object))
print(issubclass(Child,Parent))
print(issubclass(Parent,Child))
```

ex:

```
class Parent:
    def m1(self):
        print("Parent class m1")

class Child(Parent):
    def m2(self):
        print("Child class m2")

p = Parent()
p.m1()

c = Child()
c.m1()
c.m2()
```

ex:

```
class A:
    def m1(self):
        print("m1 of A called")

class B(A):
    def m2(self):
        print("m2 of B called")

class C(B):
```



```
def m3(self):  
    print("m3 of C called")  
  
c = Child()  
c.m1()  
c.m2()  
c.m3()
```

ex:

In below example 1-arg constructor not present in B class so parent class(A) constructor will be executed.

```
class A:  
    def __init__(self,name):  
        print("A class cons")  
        self.name=name  
  
class B(A):  
    def disp(self):  
        print(self.name)  
  
b = B("ratan")  
b.disp()
```

ex:

In below example 1-arg constructor present in B class so class B constructor will be executed.

```
class A:  
    def __init__(self,name):  
        print("A class cons")  
        self.name=name  
  
class B(A):  
    def __init__(self,name):  
        print("B class cons")  
        self.name=name  
    def disp(self):  
        print(self.name)  
  
b = B("ratan")  
b.disp()
```

ex:

```
class Parent:
    def m1(self):
        print("parent m1()")

class Child(Parent):
    def m1(self):
        print("child m1()")
    def disp(self):
        self.m1()          # Current class function calling
        super().m1()       # parent class function calling

c = Child()
c.disp()
```

ex:

```
class Parent:
    a,b=10,20

class Child(Parent):
    def disp(self):
        print(self.a+self.b)

c = Child()
c.disp()
```

ex:

```
class A(object):
    def save(self):
        print("class A saves")
class B(A):
    def save(self):
        print("B saves stuff")
        super().save()
        #A.save(self) # call the parent class method too
class C(A):
    def save(self):
        print("C saves stuff")
        A.save(self)

class D(B, C):
    def save(self):
        print ("D saves stuff")
        # make sure you let both B and C save too
        B.save(self)
        C.save(self)

d = D()
d.save()
```

Ex :

```
class A(object):
    def save(self):
        print("class A saves")
class B(A):
    def save(self):
        print("B saves stuff")
        super(B, self).save()
class C(A):
    def save(self):
        print("C saves stuff")
        super(C, self).save()
class D(B, C):
    def save(self):
        print ("D saves stuff")
        super(D, self).save()

d = D()
d.save()
```

ex:

```
class A:
    def m(self):
        pass

class B(A):
    def m(self):
        print("m of B called")
        super().m()
class C(A):
    def m(self):
        print("m of C called")
        super().m()

class D(B,C):
    pass
D().m()
```

Ex:

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")
```

Case -1 : **m of B called**

```
class D(B,C):
    pass
D().m()
```

Case-2 : **m of C called**

```
class D(C,B):
    pass
D().m()
```

Ex: `class A:`
 `def m(self):`
 `print("m of A called")`

`class B(A):`
 `pass`

`class C(A):`
 `def m(self):`
 `print("m of C called")`

`class D(B,C):`
 `pass`

`x = D()`
`x.m()`
 output : **"m of C called"**

Ex: `class A:`
 `def m(self):`
 `print("m of A called")`

`class B(A):`
 `def m(self):`
 `print("m of B called")`

`class C(A):`
 `pass`

`class D(C,B):`
 `pass`

`x = D()`
`x.m()`
 output : **"m of B called"**

ex:

`class A:`
 `def __init__(self):`
 `print("A.__init__")`

`class B(A):`
 `def __init__(self):`
 `print("B.__init__")`
 `super().__init__()`

`class C(A):`
 `def __init__(self):`
 `print("C.__init__")`
 `super().__init__()`

`class D(B,C):`
 `def __init__(self):`
 `print("D.__init__")`
 `super().__init__()`

`A()`
`B()`
`C()`
`D()`

Ex:

```
class A:  
    def m(self):  
        print("m of A called")
```

```
class B(A):  
    def m(self):  
        print("m of B called")
```

```
class C(A):  
    def m(self):  
        print("m of C called")
```

```
class D(B,C):  
    def m(self):  
        B.m(self)  
        C.m(self)  
        A.m(self)  
D().m()
```

Ex:

```
class A:  
    def m(self):  
        print("m of A called")
```

```
class B(A):  
    def m(self):  
        print("m of B called")  
        A.m(self)
```

```
class C(A):  
    def m(self):  
        print("m of C called")  
        A.m(self)
```

```
class D(B,C):  
    def m(self):  
        print("m of D called")  
        B.m(self)  
        C.m(self)  
D().m()
```

Ex:

```
class Person:
```

```
    def __init__(self, personName, personAge):  
        self.name = personName  
        self.age = personAge
```

```
    def showName(self):  
        print(self.name)
```

```
    def showAge(self):  
        print(self.age)
```

```
class Student:
```

```
    def __init__(self, studentId):  
        self.studentId = studentId
```

```
    def getId(self):  
        return self.studentId
```

```
class Resident(Person, Student):
```

```
    def __init__(self, name, age, id):  
        Person.__init__(self, name, age)  
        # super().__init__(name,age)  # Another way to call super class members  
        Student.__init__(self, id)
```

```
# Create an object of the subclass
```

```
resident1 = Resident('John', 30, '102')
```

```
resident1.showName()
```

```
resident1.showAge()
```

```
print(resident1.getId())
```

ex:

class A:

```
def __init__(self):  
    self.name = 'John'  
def getName(self):  
    return self.name
```

class B:

```
def __init__(self):  
    self.name = 'Richard'  
def getName(self):  
    return self.name
```

class C(A,B):

```
def __init__(self):  
    A.__init__(self)  
    B.__init__(self)
```

```
def getName(self):  
    return self.name
```

C1 = C()

print(C1.getName())

ex:

class Person(object):

```
def __init__(self, first, last):  
    self.firstname = first  
    self.lastname = last  
def Name(self):  
    return self.firstname + " " + self.lastname
```

class Employee(Person):

```
def __init__(self, first, last, staffnum):  
    super().__init__(first, last)  
    #Person.__init__(self, first, last)  
    self.staffnumber = staffnum  
def GetEmployee(self):  
    return self.Name() + ", " + self.staffnumber
```

x = Person("ratan", "addanki")

y = Employee("ratan", "addanki", "111")

print(x.Name())

print(y.GetEmployee())

The `__init__` method of our `Employee` class explicitly invokes the `__init__` method of the `Person` class. We could have used `super` instead. `super().__init__(first, last)` is automatically replaced by a call to the superclasses method, in this case `__init__`:

```
def __init__(self, first, last, staffnum):  
    super().__init__(first, last)  
    self.staffnumber = staffnum
```

ex :

```
class Person:  
    def __init__(self, first, last):  
        self.firstname = first  
        self.lastname = last  
  
    def __str__(self):  
        return self.firstname + " " + self.lastname  
  
class Employee(Person):  
    def __init__(self, first, last, id):  
        super().__init__(first, last)  
        self.id = id  
  
x = Person("ratan", "addanki")  
y = Employee("ratan", "addanki", "111")  
  
print(x)  
print(y)
```

ex: **We have overridden the method `__str__` from Person in Employee**

```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def __str__(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, id):
        super().__init__(first, last)
        self.id = id
    def __str__(self):
        return super().__str__() + " " + self.id

x = Person("ratan", "addanki")
y = Employee("ratan", "addanki", "111")

print(x)
print(y)
```

Example : **method overriding**

```
class Animal:
    def eat(self):
        print ('Animal Eating...')

class Dog(Animal):
    def eat(self):
        print ('Dog eating...')

d = Dog()
d.eat()
```

Exception handling

Two distinguishable kinds of errors:

1. syntax errors
2. Exceptions.

- ✓ Syntax error are nothing but normal mistakes done by developer while writing the code.
- ✓ An exception is an error that happens during the execution of a program.

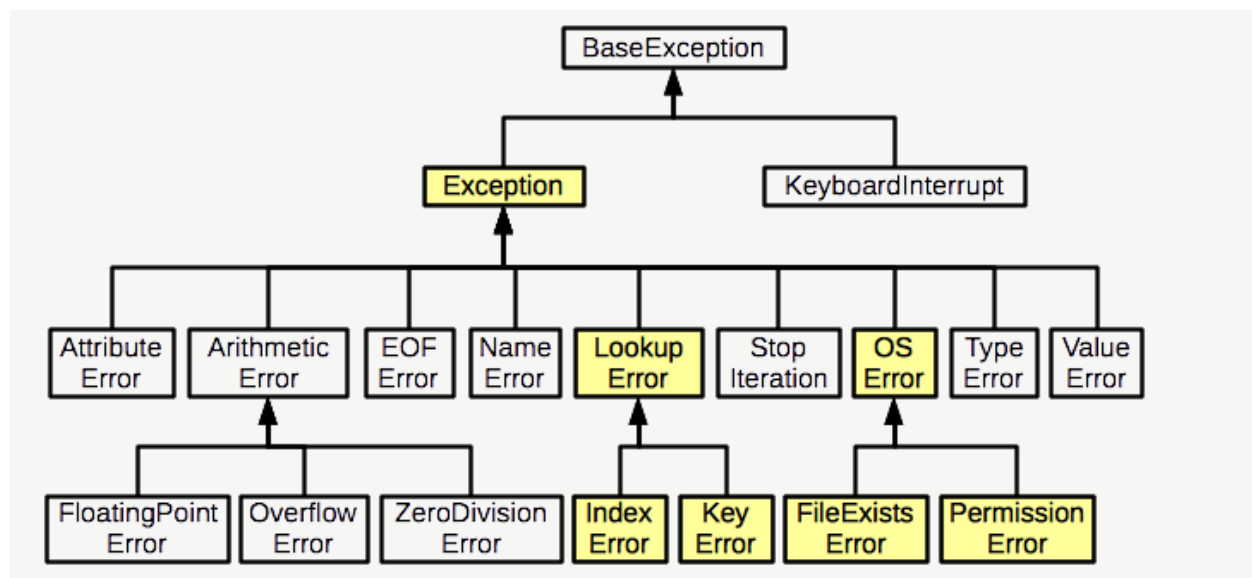
Whenever the exception raised in program,

- Program terminated abnormally
- Rest of the application not executed.

- ✓ To overcome above problem We can handle the exceptions by using **try-except** blocks.

Advantages of Exception handling:

- Program terminated normally.
- Rest of the application executed.



ZeroDivisionError	:	Occurs when a number is divided by zero.
NameError	:	It occurs when a name is not found. It may be local or global.
IndentationError	:	If incorrect indentation is given.
IOError	:	It occurs when Input Output operation fails.
EOFError	:	occurs when end of file is reached and yet operations are being performed
IndexError	:	Raised when a sequence subscript is out of range.
ImportError	:	Raised when an import statement fails to find the module definition
KeyError	:	Raised when a dictionary key is not found in the set of existing keys.
MemoryError	:	Raised when an operation runs out of memory
OSError	:	It is raised when a function returns a system-related error
SyntaxError	:	Raised when the parser encounters a syntax error.

Exception handling hierarchy:

```

BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
        |   +-- BufferError
        |   +-- ArithmeticError
        |       |   +-- FloatingPointError
        |       |   +-- OverflowError
        |       |   +-- ZeroDivisionError
        |   +-- AssertionError
        |   +-- AttributeError
        |   +-- EnvironmentError
        |       |   +-- IOError
        |       |   +-- OSError
        |           |   +-- WindowsError (Windows)
        |           |   +-- VMSError (VMS)
        |   +-- EOFError
        |   +-- ImportError
        |   +-- LookupError
        |       |   +-- IndexError
        |       |   +-- KeyError
        |   +-- MemoryError
        |   +-- NameError
        |       |   +-- UnboundLocalError
        |   +-- ReferenceError
        |   +-- RuntimeError
        |       |   +-- NotImplementedError
        |   +-- SyntaxError
        |       |   +-- IndentationError
        |       |   +-- TabError
        |   +-- SystemError
        |   +-- TypeError
        |   +-- ValueError
        |       |   +-- UnicodeError
        |           |   +-- UnicodeDecodeError
        |           |   +-- UnicodeEncodeError
        |           |   +-- UnicodeTranslateError
    +-- Warning
        +-- DeprecationWarning
        +-- PendingDeprecationWarning
        +-- RuntimeWarning
        +-- SyntaxWarning
        +-- UserWarning
        +-- FutureWarning
        +-- ImportWarning
        +-- UnicodeWarning
        +-- BytesWarning

```

Exceptions handling by using **try-except** blocks:

Syntax:

```
try:
    Exceptional code (it may or may not raise an exception)
except :
```

Code runs when exception raised in try block

- ✓ The try block contains exceptional code it may raise an exception or may not.
- ✓ If the exception raised in try block the matched except block will be executed.

ex-1

Application without try-except block

```
print("Hello")
print(10/0)
print("rest of the app.....")
```

output :

```
Hello
ZeroDivisionError: division by zero
```

Disadvantages:

- Program terminated abnormally.
- Rest of the application not executed.

Application with try-except block

```
try:
    a=10/0
except ArithmeticError:
    print (10/2)
print("rest of the Application")
```

output :

```
5
rest of the Application
```

Advantages:

- Rest of the application executed.
- Program terminated normally.

ex-2:

- ✓ If the except block is not matched so program terminated abnormally.
- ✓ In below example try block raise ZeroDivisionError but except block not matched(TypeError)

```
try:
    print(10/0)
except TypeError as a:
    print("Exception raised:",a)
print("Rest of the App")
```

ex-3: If no exception occurs in try block then code under except clause will be skipped.

```
try:
    print("ratanit")
except ArithmeticError as a:
    print("Exception raised:",a)
```

ex 4: in between try-except block statement declaration not possible.

```
try:
    print("ratanit")
except:
```

ex-5:

- ✓ If the exception raised in other than try it is always abnormal termination.
- ✓ If the exception raised in except block it is abnormal termination.
- ✓ In below example exception raised in except block it is abnormal termination.

```
print("ratan sir python is good")
try:
    n = int(input("enter a number:"))
    print(10/n)
except ArithmeticError as a:
    print(10/0)
print("rest of the app")
```

ex-6:

- ✓ Once the exception raised in try, the remaining code of the try block not executed.
- ✓ In try block it is always recommended to take exceptional code.

```
try:
    print(10/0)
    print("ratan")
    print("anu")
except ZeroDivisionError as a:
    print(10/2)
print("rest of the app")
```

Default except & else blocks:**Syntax :**

```
try:
    exceptional code

except Exception-1:
    execute code if exception raised

except Exception-2:
    execute code if exception raised
....
except Exception-N:
    execute code if exception raised

except :
    code execute if other except blocks are not matched.

else:
    this block is executed when there is no exception in try block.
```

- ✓ If the exception raised in try block the matched except block will be executed.
- ✓ **The except blocks are not matched then the default except block will be executed.**
- ✓ The else block will be executed if no exception in try block.

ex-6 :

Case 1: If the except blocks are not matched then default except block will be executed.

```
try:
    print(10/0)
except TypeError as a:
    print("ratanit")
except:
    print("durgasoft")
    print("rest of the app")
```

case 2: The default except block must be last statement.

```
try:
    print(10/0)
except:
    print("ratanit")
except TypeError as a:
    print(10/2)
print("rest of the app")
SyntaxError: default 'except:' must be last
```

ex-7: try with multiple except blocks

```

try:
    n = int(input("enter a num:"))
    print(10/n)
except ZeroDivisionError as a:
    print("exception msg=",a)
except ValueError as a:
    print("exception msg=",a)
else:
    print("no exception in try")
print("Rest of the app")

```

enter a num:2	write the output
enter a num:0	write the output
enter a num:"ratan"	Write the output

ex-8: try with multiple except blocks

```

print("ratan sir python is good")
try:
    n = int(input("enter a number:"))
    print(10/n)
    print(10+"ratan")
except ArithmeticError as a:
    print("ratanit.com")
except TypeError as e:
    print("operations are not supported",e)
except:
    print("durgasoft")
print("rest of the app")

```

write the output:

enter a num	:	5
enter a num	:	0
enter a num	:	"ratan"

ex-8: single except block handling multiple exceptions.

```

try:
    n = int(input("enter a num:"))
    print(10/n)
except (ZeroDivisionError,ValueError) as a:
    print("exception msg=",a)
else:
    print("no exception in try")
print("Rest of the app")

```

enter a num:2	write the output
enter a num:0	write the output
enter a num:"ratan"	Write the output

ex-9:

- ✓ The top level root class for all exceptions is : **BaseException**
- ✓ The root class can handle all exceptions (**Exception , BaseException**)

```
try:
    n = int(input("enter a num:"))
    print(10/n)
except BaseException as a:
    print("exception msg=",a)
else:
    print("no exception in try")
print("Rest of the app")
```

enter a num:2	write the output
enter a num:0	write the output
enter a num:"ratan"	Write the output

Note: the below all declarations can handle all exceptions, all are valid & same

```
except (ZeroDivisionError,ValueError) as a:
except BaseException as a:
except Exception as a:
except:
```

ex : When we declare more than one except blocks , the order must be child to parent, if we are declaring parent to child we will get error message

Case 1: valid: except block order is Child to Parent

```
try:
    n = int(input("enter a number:"))
    print(10/n)
except ZeroDivisionError:
    print(10/5)
except BaseException:
    print(10/2)
print("rest of the app")
```

Case 2: Invalid : Because the except block order is parent to child

```
except BaseException:
    print(10/2)
except ZeroDivisionError:
    print(10/5)
error: bad except clause order : BaseException class is ancestor class of ZeroDivisionError
```

ex: We learn from the below result that the function catches the exception.

```
def m1():
    n = int(input("enter a num:"))
    print(10/n)

try:
    m1()
except (ArithmeticError, ValueError) as a:
    print("exception raised=", a)
else:
    print("no exception")
print("Rest of the app")

enter a num : 0
enter a num : 5
enter a num : "ratan"
```

ex: In method try block there may be a chance of two exceptions ZeroDivisionError, ValueError but here one exception handled inside the function, another exception handle outside of the function.

```
def f():
    try:
        n = int(input("Enter a num:"))
        print(10/n)
    except ZeroDivisionError as e:
        print("got it in the function :-)", e)

try:
    f()
except ValueError as e:
    print("got it out of function:-)", e)
```

ex:

```
def m1():
    n = int(input("enter a number:"))
    print(10/n)
def m2():
    print(10+"ratan")

try:
    m1()
    m2()
except ArithmeticError as a:
    print("ratanit.com")
except:
    print("durgasoft")
else:
    print("no exception")
print("rest of the app")
```

Finally block :

- ✓ The try block contains exceptional code it may raise an exception or may not.
- ✓ If the exception raised in try block the matched except block will be executed.
- ✓ If the except blocks are not matched then default except block will be executed.
- ✓ The default except block must be last statement.
- ✓ The else block is executed if there is no exception in try block.
- ✓ Finally block is always executed irrespective of try-except blocks.
- ✓ The finally block is used to write the resource releasing code.

```
try:
    <exceptional code>
except <ExceptionType1>:
    <handler1>
except <ExceptionTypeN>:
    <handlerN>
except:
    <handlerExcept>
else:
    <process_else>
finally:
    <process_finally>
```

Ex:

```
try:
    n = int(input("enter a number"))
    print(10/n)
except ArithmeticError as e:
    print("ratanit.com")
else:
    print("no xception")
finally:
    print("finally block")
```

ex:

```
try:
    print(10+"ratan")
except ArithmeticError as e:
    print("ratanit.com")
finally:
    print("finally block")
```

ex:

```
try:
    print("try block")
finally:
    print("finally block")
```

ex : Invalid : else must be with presence of except block.

```
try:
    print("try block")
else:
    print("no exception")
finally:
    print("finally block")
```

In two cases finally block is not executed

- ✓ if the control is not entered in try block
- ✓ when we use `os._exit(0)`

case 1: The control is not entered in try block so finally block is not executed.

```
print(10/0)
try:
    print("try block")
finally:
    print("finally block")
```

case 2: when we use `os._exit(0)` virtual machine is shutdown so finally block is not executed.

```
import os
try:
    print("try block")
    os._exit(0)
finally:
    print("finally block")
```

ex: If the try,except , finally block contains exceptions : it display finally block exception

```
try:
    print(10/0)
except ArithmeticError as e:
    print("ratan"+10)
finally:
    s="ratan"
    print(s[10])
```

ex: If the try,except , finally block contains return statement : it display finally block return

```
def m1():
    try:
        return 10
    except ArithmeticError as e:
        return 20
    finally:
        return 30

print(m1())
```

```
ex :   try:
        num1, num2 = eval(input("Enter two numbers, separated by a comma : "))
        result = num1 / num2
        print("Result is", result)
    except ZeroDivisionError:
        print("Division by zero is error !!")
    except SyntaxError:
        print("Comma is missing. Enter numbers separated by comma like this 1, 2")
    except:
        print("Wrong input")
    else:
        print("No exceptions")
    finally:
        print("This will execute no matter what")
```

case 1: Enter two numbers, separated by a comma : 10,2

Result is 5.0

No exceptions

This will execute no matter what

Case 2: Enter two numbers, separated by a comma : 10,0

Division by zero is error !!

This will execute no matter what

Case 3: Enter two numbers, separated by a comma : 10.6

Wrong input

This will execute no matter what

Nested try-except block:

- ✓ In outer try block if there is no exception then outer else block will be executed.
- ✓ In inner try block if there is no exception then inner else block will be executed.

```
Ex:   try:
        print("outer try block")
        n = int(input("enter a number"))
        print(10/n)
    try:
        print("inner try")
        print("anu"+"ratan")
```

```

        except TypeError:
            print("ratanit.com")
        else:
            print("inner no exception")
    except ArithmeticError:
        print(10/5)
    else:
        print("outer no excepiton")
    finally:
        print("finally block")

```

ex:

```

n = int(input("enter a number:"))
try:
    print("outer try block")
    try:
        print("Inner try block")
        print(10/n)
    except NameError:
        print("Inner except block")
    finally:
        print("Inner finally block")
except ZeroDivisionError:
    print("outer except block")
else:
    print("else block execute")
finally:
    print("outer finally block")
print("Rest of the Application")

```

enter a number: 0
 outer try block
 Inner try block
 Inner finally block
 outer except block
 outer finally block
 Rest of the Application

enter a number: 2
 outer try block
 Inner try block
 5.0
 Inner finally block
 else block execute
 outer finally block
 Rest of the Application

ex :

```

st-1
st-2
try:
    st-3
    st-4
    try:
        st-5
        st-6
    except:
        st-7
        st-8

```

```

except:
    try:
        st-9
        st-10
    except:
        st-11
        st-12
else:
    st-13
    st-14
finally:
    st-15
    st-16
st-17
st-18

```

case 1: No exception in above example.

Case 2: Exception raised in st-2

Case 3: exception raised in st-3 the except block is matched.

Case 4: exception raised in st-4 the except block is not matched.

Case 5: exception raised in st-5 the except block is matched.

Case 6: exception raised in st-6 the inner except block is not matched but outer except block is matched .

Case 7 : Exception raised in st-5 the except block is matched while executing except block
exception raised in st-7 the inner except block executed while executing inner except
block exception raised in st-9 , the inner except block executed while executing inner
except exception raised in st-11.

Case 8 : Exception raised in st-6 the except block is matched while executing except block
exception raised in st-8 the inner except block executed while executing inner except
block exception raised in st-10 , the inner except block executed while executing inner
except exception raised in st-12.

Case 9 : exception raised in st-13

Case 10 : exception raised in st-15

Case 11 : exception raised in st-18

There are two types of exceptions.

1. predefined exception : `ArithmeticError`
2. user defined exceptions : `InvalidAgeError`

- ✓ By using raise keyword we can raise predefined exceptions & user defined exceptions.
- ✓ By using raise keyword it is not recommended to raise predefined exceptions because predefined exceptions contains some fixed meaning(don't disturb the meaning).

raise `ArithmeticError`("name is not good")

- ✓ By using raise keyword it is recommended to raise user defined exceptions.

raise `InvalidAgeError`("age is not good")

raise keyword :

- ✓ To raise your exceptions from your own methods you need to use raise keyword.
 raise ExceptionClassName("Your information")

ex :

```
try:
    raise NameError("Hello")
except NameError as e:
    print ("An exception occurred=",e)
```

ex1:

```
def status(age):
    if age < 0:
        raise ValueError("Only positive integers are allowed")

    if age>22:
        print("eligible for mrg")
    else:
        print("not eligible for mrg try after some time")
```

```
try:
    num = int(input("Enter your age: "))
    status(num)
finally:
    print("finally block....")
```

Enter your age: 23

finally block....

ValueError: Only positive integers are allowed

```
Ex:    def status(age):
        if age < 0:
            raise ValueError("Only positive integers are allowed")
        if age>22:
            print("eligible for mrg")
        else:
            print("not eligible for mrg try after some time")

    try:
        num = int(input("Enter your age: "))
        status(num)
```



```
except ValueError:
    print("Only positive integers are allowed you .....")
finally:
    print("finally block....")
```

User defined exceptions :

```
ex : class NegativeAgeException(RuntimeError):
    def __init__(self, age):
        super().__init__()
        self.age = age

    def status(age):
        if age < 0:
            raise NegativeAgeException("Only positive integers are allowed")
        if age > 22:
            print("Eligible for mrg")
        else:
            print("not Eligible for mrg....")

    try:
        num = int(input("Enter your age: "))
        status(num)
    except NegativeAgeException:
        print("Only positive integers are allowed")
    except:
        print("something is wrong")
```

```
ex : class YoungException(Exception):
    def __init__(self, age):
        self.age = age

class OldException(Exception):
    def __init__(self, age):
        self.age = age

age = int(input("Enter Age:"))
if age < 18:
```

```
        raise YoungException("Plz wait some time ")
    elif age>65:
        raise TooOldException("Your age too old")
    else:
        print("we will find one girl soon")
```

Enter Age: 12

Check the output

Enter Age: 89

Check the output

```
ex : class TooYoungException(Exception):
      def __init__(self,age):
          self.age=age

      class TooOldException(Exception):
          def __init__(self,age):
              self.age=age

      try:
          age=int(input("Enter Age:"))
          if age<18:
              raise YoungException("Plz wait some time ")
          elif age>65:
              raise TooOldException("Your age too old")
          else:
              print("we will find one girl soon")
      except YoungException as e:
          print("Plz wait some time ")
      except OldException as e:
          print("Your age too old ")
```

File Handling in Python

- ✓ Python can be used to read and write data. Also it supports reading and writing data to Files.
- ✓ File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

Generally we divide files in two categories,

- Text file
- Binary file.

Text file:

Text files are contains simple text(character data).

Ex: .html , .c , .cpp , .javaetc

Binary files:

Binary files contain binary data which is only readable by computer.

Ex : video files ,audio files , ,jpg file ,pdf filesetc

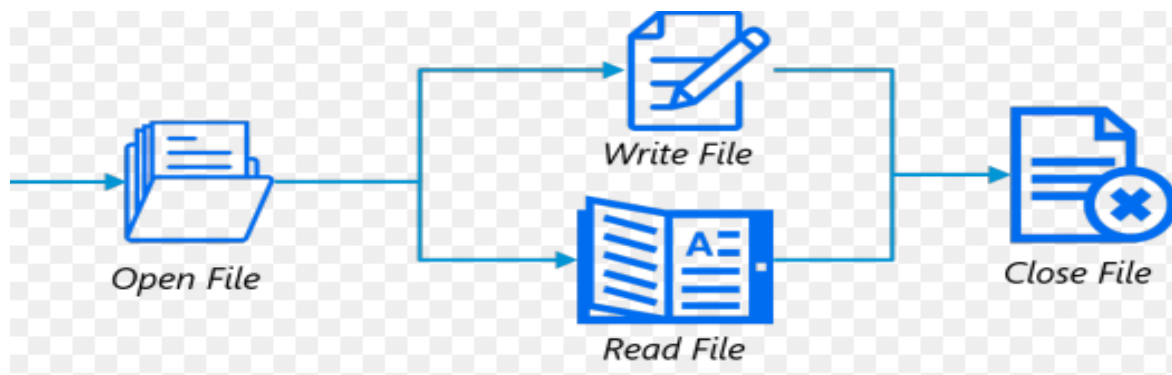
- ✓ File handling in Python does not require any predefined modules.
- ✓ The File object provides basic functions to manipulate files.
- ✓ In Python, file processing takes place in the following order.

To perform file handling, we need to perform these steps:

Open File

Read / Write File

Close File



Step 1: open the file**Opening a File:**

Before working with Files you have to open the File. To open a File, Python built in function `open()` is used. It returns an object of File which is used with other functions.

The open function takes two arguments, the name of the file and the mode of operation.

`f = open (filename, mode)`

The default file operations is read mode

```
f = open("test.txt")           # open file in current directory
f = open("C:/Python33/ratan.txt") # specifying full path
```

step 2: write or read or append the data

Note: the default mode of the file is: Read

Writing to a File: `write()` method is used to write a string into a file.

Reading from a File: `read()` method is used to read data from the File.

The read functions contains different methods, `read()`, `readline()` and `readlines()`

```
read()      #return one big string
readline    #return one line at a time
readlines   #returns a list of lines
```

Append operations: used to append the data to existing file.

Step 3: close the file**Closing a File:**

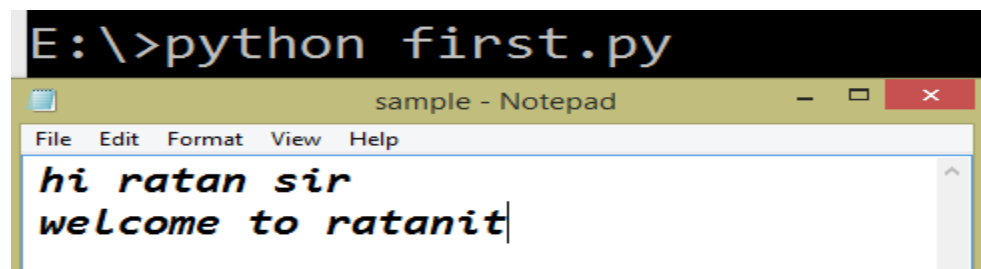
Once you are finished with the operations on File at the end you need to close the file. It is done by the `close()` method.

Mode	Description
r	Open a file for reading.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
r+	opens for reading and writing (cannot truncate a file)
w+	For writing and reading (can truncate a file)
a	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Open in text mode.
b	Open in binary mode.
x	Open for exclusive creation, failing if the file already

Ex-1: writing the data to file

"w" - Write - Opens a file for writing, creates the file if it does not exist
If the file is already available it will overwrite the data. All previous data are erased.

```
text = "hi ratan sir \n welcome to ratanit"  
f = open("sample.txt", "w")  
f.write(text)  
f.close()  
print("Operations are completed....")
```



Ex-2: To read the data from file , the file is mandatory otherwise we will get **IOError**.

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

```
try:  
    f1=open("sample.txt", "r")  
    s=f1.read()  
    print (s)  
except IOError as e:  
    print(e)  
print("rest of the application")
```

Ex 3: "a" - Append - Opens a file for appending, creates the file if it does not exist

case 1: file is already available : the data is appended to existing file.

```
f = open("sample.txt", "a")  
f.write("\n Good Morning ratan sir")  
f.close()  
print("operations are completed....")
```

case 2: file is not available : create the file write the data

```
f = open("ratan.txt", "a")  
f.write("Good Morning ratan sir")  
f.close()  
print("operations are completed....")
```

Ex-4: knowing file information

Name Returns the name of the file.

Mode Returns the mode in which file is being opened.

Closed Returns Boolean value. True, in case if file is closed else false.

```
f1 = open("sample.txt", "w")
print(f1)
print(f1.name)
print(f1.mode)
print(f1.closed)
f1.close()
print(f1.closed)
```

E:\>python first.py

```
<open file 'sample.txt', mode 'w' at 0x00BDD1D8>
sample.txt
w
False
True
```

Ex-5: when we declare the file by using with statement, the file is automatically closed once the execution completed.

```
with open("sample.txt", "w") as fh:
    fh.write("hi friends\nwelcome to ratanit!\n")
print(fh.closed)
```

Ex-6: Reading specific characters (5-char, 10-charetc)

reading all entire file data

```
f1=open("abc.txt", "r")
s=f1.read()
print (s)
f1.close()
```

#reading 10-chracters

```
f2=open("abc.txt", "r")
print(f2.read(10))
print(f2.read(5))
f2.close()
```

Note: In above example we are creating 2-file objects, when we create new file object the cursor is pointing to starting character of the file.

Note: first we are reading 10 characters, after that we are reading 5-characters (these are reading data from 11 characters onwards).

Ex-7: seek() :function move the cursor to specific location.

Case 1: in below example cursor reading the continues.

```
obj=open("abc.txt","w")
obj.write("Welcome to ratanit")
obj.close()
```

```
f1=open("abc.txt","r")
print(f1.read(5))
print(f1.read(5))
print(f1.read(5))
f1.close()
```

E:\>python first.py

Welco
me to
rata

Here after reading 5-char the cursor present in same location, then it is reading from 6-character onwards.

Case 2: in below example always we are moving cursor to starting position by using seek() function.

```
obj=open("abc.txt","w")
obj.write("Welcome to ratanit")
obj.close()
```

```
f1=open("abc.txt","r")
print(f1.read(5))
f1.seek(0)
print(f1.read(5))
f1.seek(0)
print(f1.read(5))
f1.close()
```

E:\>python first.py

Welco
Welco
Welco

Here after reading 5-char we are moving cursor to first location by using seek() function.

Observation:

```
fobj = open("sample.txt", 'w')
fobj.write('hi\n')
fobj.write('ratan\n')
fobj.write('sir\n')
fobj.write('how are you')
fobj.close()
```

Ex-8:

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the `close()` method to close a file.

```
f1=open("sample.txt","r")
print(f1.read())
```

```
f1 = open("ratan.txt","w")
f1.write("hi ratan sir")
f1.close()
```

Ex-9:

- ✓ When we read the data from file after reading the data the cursor is present in same location.
- ✓ To overcome above problem to move the cursor to particular location use **`seek()`** function.
- ✓ To know the current location of the file use **`tell()`** function.

reading entire file data

```
f = open("sample.txt","r")
s = f.read()
print (s)
```

read only first 2-characters

```
f.seek(0)
s1 = f.read(2)
print (s1)
```

after 5th character read only 3-characters

```
f.seek(5)
s1 = f.read(3)
print (s1)
```

```
print(f.tell())
f.close()
print ("operations are completed")
```

E:\>python first.py

hi ratan

hi

tan

8

Operations are completed

Ex-10:

we can read the data in three ways

1. *Read ()*
2. *Readline()*
3. *Readlines()*

#Read operations ; read complete String

```
f1 = open("abc.txt", "r")  
s = f1.read()  
print(s)  
f1.seek(0)
```

#To read one line at a time, use:

```
print (f1.readline())  
f1.seek(0)
```

#To read a list of lines use

```
print (f1.readlines())
```

Ex-11 : Assignment

read the data from abc.txt

abc.txt file first 5-characters write to : a.txt

abc.txt file first 10-characters write to : b.txt

abc.txt file first 15-characters write to : c.txt

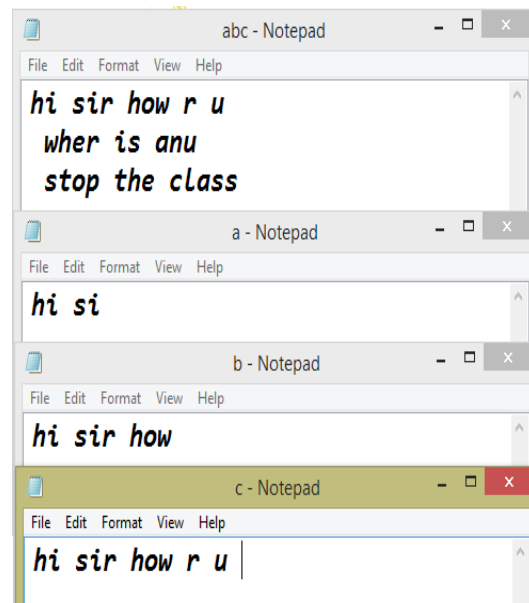
```
f = open("abc.txt", "r")
```

```
f1 = open("a.txt", "w")  
f1.write(f.read(5))
```

```
f2 = open("b.txt", "w")  
f.seek(0)  
f2.write(f.read(10))
```

```
f3 = open("c.txt", "w")  
f.seek(0)  
f3.write(f.read(15))
```

```
f1.close()  
f2.close()  
f3.close()  
f.close()  
print("operations are completed")
```



ex -12 :

Reading the data from one file, writing the data to another file.
In below example the for loop is reading the data line by line format.

```
f1 = open("sample.txt")
f2 = open("abc.txt", "w")

for line in f1:
    f2.write(line)
f1.close()
f2.close()
```

ex-13 :

"x" - Create - Creates the specified file, returns an error if the file exists

x : exclusive creation : **python 3.x**

if the file is not available then only it will create new file.

if the file is available we will get error : **FileExistsError: [Errno 17] File exists: abc.txt'**

```
f = open("abc.txt", 'x')
f.write("hi sir /n python is good")
```

```
f = open("acac.txt", 'r')
print(f.read())
f.close()
print("Operations are completed")
```

ex 14 : file operations module : os

- ✓ There is a module "os" defined in Python that provides various functions which are used to perform various operations on Files.
- ✓ To use these functions 'os' needs to be imported by using import keyword.

rename() : It is used to rename a file. It takes two arguments, *existing_file_name* and *new_file_name*.

remove() : It is used to delete a file. It takes one argument.

makedirs() : It is used to create a directory. A directory contains the files.

chdir() : It is used to change the current working directory.

getcwd() : It gives the current working directory.

rmdir() : It is used to delete a directory. It takes one argument which is the name of the directory.

Case 1: rename() remove() functions

```
import os
os.rename("sample.txt", "ratan.txt")
os.remove("Test.java")
```

case 2: import os

```
os.makedirs("new")
os.chdir("hadoop")
print(os.getcwd())
```

case 3: In order to delete a directory, it should be empty.

In case directory is not empty first delete the files.

```
import os
os.rmdir("new")
```

Ex-15:

- ✓ *r+* opening a file both reading & writing operations. (file is mandatory)
- ✓ While performing operations on the file if it is not available we will get error message.
- ✓ In this mode while performing write operations the data will be replaced with existing data.

```
f = open("sample.txt", "r+")
x = f.read()
print(x)
f.seek(0)
```

```
f.write("aaa")
f.seek(0)
x = f.read()
print(x)
```

E:\>python first.py

ratantit

aaaanit

ex-16:

- ✓ *W+* opening a file both read & writes operations. (file is optional)
- ✓ When we open the file, if the file is not available then it will create the file.
- ✓ When we open the file, if the file is available then it will erase the file data then creates empty file.

```
f = open("sample.txt", "w+")
f.write("aaa")
f.seek(0)
x = f.read()
print(x)
```

ex-18:

- ✓ *A+* opening a file both read & append mode. (file is optional)
- ✓ When we open the file, if the file is not available then it will create the file.
- ✓ When we open the file, if the file is available then the cursor is present in end of the file.

```
fh = open("sample.txt", "a+")
fh.write("\nHello World")
#read operations
fh.seek(0)
s = fh.read()
print(s)
fh.close()
```

***** **Completed : Thank you : Believes only practical Knowledge*******

Abstraction :

- ✓ The abstract class contains one or more abstract methods.
- ✓ The abstract method contains only method declaration but not implementation.
- ✓ In Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs) not possible to create the object of abstract classes.
- ✓ A class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

Ex:

```
from abc import ABC, abstractmethod
class Test(ABC):
    @abstractmethod
    def m1(self):
        pass

class Test1(Test):
    def m1(self):
        print("implementation here")

#t = Test() # abstract class object creation not allowed
t1 = Test1()
t1.m1()
```

ex:

```
from abc import ABCMeta, abstractmethod
class Animal(object):
    __metaclass__ = ABCMeta
    @abstractmethod
    def eat(self):
        pass
class Tiger(Animal):
    def eat(self):
        print("Tiger implementation ...")
Tiger().eat()
```

ex: *from abc import ABC, abstractmethod*

```
class Animal(ABC):
    @abstractmethod
    def eat(self):
        pass

class Tiger(Animal):
    def eat(self):
        print("Tiger implementation ...")

class Lion(Animal):
    def eat(self):
        print("Lion implementation here...")

t = Tiger()
t.eat()
l = Lion()
l.eat()
```

ex: *Abstract class constructor.*

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
        super().__init__()

    @abstractmethod
    def do_something(self):
        pass

class DoAdd(AbstractClassExample):
    def do_something(self):
        return self.value + 42

class DoMul(AbstractClassExample):

    def do_something(self):
        return self.value * 42

x = DoAdd(10)
y = DoMul(10)
```

```
print(x.do_something())
print(y.do_something())
```

ex:

ex 1:

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def eat1(self):
        pass
    @abstractmethod
    def eat2(self):
        pass
class Tiger(Animal):
    def eat1(self):
        print("Tiger implementation ...")
    def eat2(self):
        print("Tiger implementation ...")
```

```
t = Tiger()
t.eat1()
t.eat2()
```

ex 2:

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def eat1(self):
        pass
    @abstractmethod
    def eat2(self):
        pass
class Tiger(Animal):
    def eat1(self):
        print("Tiger implementation ...")
```

```
t = Tiger()
t.eat1()
TypeError: Can't instantiate abstract class Tiger with
abstract methods eat2
```

ex 3 :

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def eat1(self):
        pass
```

```
@abstractmethod
def eat2(self):
    pass
class Tiger(Animal):
    def eat1(self):
        print("Tiger implementation ...")
class lion(Tiger):
    def eat2(self):
        print("lion implementation ...")

t = lion()
t.eat1()
t.eat2()
```

ex:

The pass statement does nothing. It can be used when a statement is required syntactically but the while True:

```
pass ...
```

This is commonly used for creating minimal classes:

```
class MyEmptyClass:
```

```
    pass
```

```
def initlog(*args):
```

```
    pass # Remember to implement this!
```

Python working with Database

Mysql:

```

C:\Program Files (x86)\MySQL\MySQL Server 5.0\bin\mysql.exe
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.67-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database anu;
Query OK, 1 row affected (0.00 sec)

mysql> use anu;
Database changed
mysql> create table emp(eid int,ename varchar(30),esal int);
Query OK, 0 rows affected (0.17 sec)

mysql> desc emp;
+-----+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| eid   | int(11)| YES  |     | NULL    |       |
| ename | varchar(30)| YES  |     | NULL    |       |
| esal  | int(11)| YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.08 sec)

mysql> insert into emp values(111,'ratan',10000);
Query OK, 1 row affected (0.08 sec)

mysql> select * from emp;
+-----+-----+-----+
| eid | ename | esal |
+-----+-----+-----+
| 111 | ratan | 10000 |
+-----+-----+-----+
1 row in set (0.06 sec)

mysql> update emp set esal=esal+100 where esal>5000;
Query OK, 1 row affected (0.09 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from emp;
+-----+-----+-----+
| eid | ename | esal |
+-----+-----+-----+
| 111 | ratan | 10100 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> drop table emp;
Query OK, 0 rows affected (0.08 sec)

mysql> select * from emp;
ERROR 1146 (42S02): Table 'anu.emp' doesn't exist
mysql>

```


Ex-1: Table creation process.

```
import mysql.connector
try:
    con=mysql.connector.connect(host="localhost",user="root",password="root",database="ratan")
    print("Connection successfull")
    cursor=con.cursor()
    cursor.execute("create table emp(eid int,ename varchar(20),esal int)")
    print("table created successfully....")
    cursor.close()
    con.close()
except mysql.connector.Error as a:
    print("operations are fail....",a)
```

observation : if the emp table is already available we will get error message.

```
E:\>python first.py
Connection successfull
operations are fail.... 1050 (42S01): Table 'emp' already exists
```

observation: if the username or password wrong we will get error message

```
E:\>python first.py
operations are fail. 1045 (28000): Access denied for user 'root'@'localhost' (using password: YES)
```

observation : if the query is wrong we will get error message.

```
E:\>python first.py
Connection successfull
operations are fail.... 1064 (42000): You have an error in your SQL syntax;
```

Ex-2: Data insertion process.

```
import mysql.connector
try:
    con=mysql.connector.connect(host="localhost",user="root",password="root",database="ratan")
    print("Connection created successfully")
    cursor=con.cursor()
    cursor.execute("insert into emp values('%d','%s','%g')" %(111,"ratan",10000))
    cursor.execute("insert into emp values('%d','%s','%g')" %(222,"anu",20000))
    cursor.execute("insert into emp values('{0}','{1}','{2}')" .format(333,"durga",30000))
    print("values are inserted successfully....")
    con.commit()
    cursor.close()
    con.close()
except mysql.connector.Error as a:
    print("operations are fail....",a)
```

ex -3 : insertion of record in database table in different approach.

Case 1 : inserting one record.

```
cursor=con.cursor()
query = "insert into login_table values(%s,%s)"
val=('Ratan','anu')
cursor.execute(query,val)
print("values are inserted successfully....")
print(cursor.rowcount, "was inserted.")
con.commit()
```

case 2: inserting multiple records.

```
cursor=con.cursor()
query = "insert into login_table values(%s,%s)"
val=[('Ratan','anu'),('aaa','bbb'),('raju','rani')]
cursor.executemany(query,val)
print("values are inserted successfully....")
print(cursor.rowcount, "was inserted.")
con.commit()
```

ex: Table data updating process by using UPDATE command.

```
import mysql.connector
```

```
try:
```

```
con=mysql.connector.connect(host="localhost",user="root",password="root",database="ratan")
print("Connection created successfully")
mycursor=con.cursor()
mycursor.execute("update emp set esal=esal+{} where esal>{}".format(100,1000))
print("values are updated successfully....",mycursor.rowcount)
con.commit()
mycursor.close()
con.close()
```

```
except mysql.connector.Error as err:
```

```
print("operataions are fail.....",err)
```

ex: Table data deleting process by using DELETE command.

```
import mysql.connector
```

```
try:
```

```
con=mysql.connector.connect(host="localhost",user="root",password="root",database="ratan")
print("Connection created successfully")
mycursor=con.cursor()
mycursor.execute("delete from emp where eid=%d"%(111))
print("deleted records....",mycursor.rowcount)
con.commit()
mycursor.close()
con.close()
```

```
except mysql.connector.Error as err:
```

```
print("operataions are fail.....",err)
```

ex: Application with try-except-finally**syntax :**

try:
 exceptional code : it may or may not raise an exception
except:
 this code executed when the exception raised in try block
finally:
 to release the resources.

```
import mysql.connector
conn=None
mycursor=None
```

```
try:
    con=mysql.connector.connect(host="localhost",user="root",password="root",database="ratan")
    print("Connection created successfully")
    mycursor=con.cursor()
    mycursor.execute("delete from emp where esal>%d"%(10000))
    print("deleted records....",mycursor.rowcount)
    con.commit()
except mysql.connector.Error as err:
    print("operataions are fail.....",err)
finally:
    if mycursor is not None :
        mycursor.close()
    if con is not None:
        con.close()
print("Application completed Successfully.....")
```

Note : To release the resources use finally block because the finally block is always executed, irrespective of try-except block.

Note : Before releasing the resources it is always recommended to check resource available or not.

Note : when we create the resources(connection , cursor) within the try block the scope is only within try block, so it is recommended to declare the resources outside the try block.

Note : when we declare the resources outside of the try block then we can access that resources in three blocks(try-except-finally).

ex : fetching (select) data from database.

```
import mysql.connector
try:
    con=mysql.connector.connect(host="localhost",user="root",password="root",database="ratan")
    print("Connection created successfully")
    mycursor=con.cursor()

    mycursor.execute("SELECT * FROM login_table")
    myresult = mycursor.fetchall()
    for x in myresult:
        print(x)

    con.commit()
    mycursor.close()
    con.close()
except mysql.connector.Error as a:
    print("operations fail....",a)
```

Fetching selected columns from the database table:

```
mycursor.execute("select eid,ename from emp")
myresult = mycursor.fetchall()
```

Reading first row of the table :

```
mycursor.execute("SELECT * FROM customers")
myresult = mycursor.fetchone()
```

ex: selecting the data from database table.

```
import mysql.connector
try:
    con=mysql.connector.connect(host="localhost",user="root",password="root",database="ratan")
    print("Connection created successfully")
    mycursor=con.cursor()
    mycursor.execute("select * from emp")
    results = mycursor.fetchall()
    # Fetch all the rows
    for row in results:
        eid = row[0]
        ename = row[1]
        esal = row[2]
        # Now print fetched result
        print ("eid=%d ,ename=%s ,esal=%g"%(eid,ename,esal))
    con.commit()
    mycursor.close()
    con.close()
except mysql.connector.Error as a:
    print("operations are fail....",a)
```

Example :

fibonacci.py

```
def fib(n): # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b
```

first.py**case 1:**

```
import fibo
fibo.fib(100)
```

case 2:

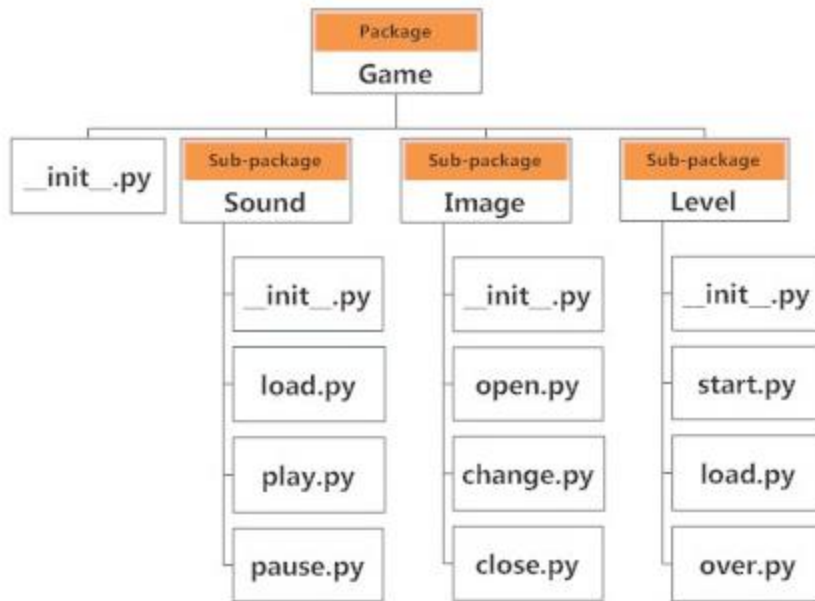
```
from fibo import fib
fib(100)
```

case 3:

```
from fibo import *
fib(100)
```

Packages & Modules

- ✓ If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a script.
- ✓ As your program gets longer, you may want to split it into several files for easier maintenance. To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a module
- ✓ As a directory can contain sub-directories and files, a Python package can have sub-packages and modules. **Python has packages for directories and modules for files.**
- ✓ definitions from a module can be imported into other modules or into the main module. A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. **Any Python file can be referenced as a module.**



- ❖ By using import statement we can import one module into another module .
`import <module-name>` **#Access the members by using module name**
- ❖ Form module if you want import specific data use below command.
`from <module-name> import < name1, name2, ... nameN >`
- ❖ Form module if you want import all the elements use below command.
`from <module-name> import *` **# Access the members directly**

ex-1:

#arithmetic.py

```
def add(x, y):
    print(x + y)
def mul(x, y):
    print(x * y)
```

#client.py

#Approach-1

```
import arithmetic
arithmetic.add(3,4)
arithmetic.mul(3,4)
```

#Approach-2

```
from arithmetic import add
from arithmetic import mul
#from arithmetic import add,mul
add(4,4)
```

```
mul(4,4)
```

#Approach-3

```
from arithmetic import *  
add(5,5)  
mul(5,5)
```

Approach 1 : *Importing module so access the functions inside the module by using module-name*

Approach-2: *Access specific functions of the module without using module name.*

Approach -3 : *Access all functions of the particular module without using module-name.*

ex 2:

A.py

```
def disp1():  
    print("A disp1()")  
def disp2():  
    print("A disp2()")
```

B.py

```
def show1():  
    print("good morning")  
def show2():  
    print("good evening")
```

client.py : Access the both A,B module data.

#Approach-1

```
import A  
import B
```

```
A.disp1()
A.disp2()
```

```
B.show1()
B.show2()
```

#Approach-2

```
from A import disp1,disp2
from B import show1,show2
```

```
disp1()
disp2()
```

```
show1()
show2()
```

#Approach-3

```
from A import *
from B import *
```

```
disp1()
disp2()
```

```
show1()
show2()
```

ex-3:**A.py**

```
def disp():
    print("A disp()")
```

B.py

```
def disp():
    print("good morning")
```

client.py : Access the both A,B module data.**#Approach-1**

```
import A
import B
A.disp()
B.disp()
```

#Approach-2

```
from A import disp
disp()
```

A module function executed


```
from B import disp
disp()                                # B module function executed
```

```
from A import disp
from B import disp
disp()                                # B module function executed
```

```
from B import disp
from A import disp
disp()                                # A module function executed
```

#Approach-3

```
from A import *
from B import *
disp()                                # B module function executed
```

```
from B import *
from A import *
disp()                                # A module function executed
```

output :

E:\>python client.py

A disp()

good morning

A disp()

good morning

good morning

A disp()

good morning

A disp()

ex -4:

Fruit.py

```
class Fruits:
    def disp1(self):
        print("mango, banana")
```

Food.py

```
class Foods:
    def disp2(self):
        print("Biryani, chicken65")
```

client.py : Access Fruit.py & Food.py module data here .

#Approach-1

```
import Fruit
import Food
f = Fruit.Fruits()
f.disp1()
```

```
ff = Food.Foods()
ff.disp2()
```

#Approach-2

```
from Fruit import Fruits
from Food import Foods
f = Fruits()
f.disp1()
ff = Foods()
ff.disp2()
```

#Approach-3

```
from Food import *
from Fruit import *
f = Fruits()
f.disp1()
```

```
ff = Foods()
ff.disp2()
```

E:\>python client.py

Write the output here.

Ex-5:

first.py

```
class A:
    def m1(self):
        print("A m1() ")
```

```
class B:
    def m1(self):
        print("B m1()")
```

second.py

```
class Animal:
    def disp(self):
        print("Tiger...")
```

```
class Person:
    def disp(self):
        print("very dangerous...")
```

client.py : Access both first.py & second.py module data

#Approach-1

```
import first
import second
```

```
a = first.A()
a.m1();
b = first.B()
b.m1()
```

```
an = second.Animal()
an.disp()
p = second.Person()
p.disp()
```

#Approach-2

```
from first import A,B
from second import Animal,Person
a = A()
a.m1();
b = B()
b.m1()
an = Animal()
an.disp()
p = Person()
p.disp()
```

#Approach-3

```
from first import *
from second import *
A().m1()
B().m1()
Animal().disp()
Person().disp()
```

Ex-6:**Employee.py**

```
class Emp:
    def __init__(self, eid, ename):
        self.eid = eid
        self.ename = ename
    def disp(self):
        print("Emp id : ",self.eid , "Emp name: ",self.ename)
```

```
class MyClass:
    def disp(self):
        print("ratan sir good")
```

Student.py

```
class Stu:
    def __init__(self, rollno, name):
        self.rollno = rollno
```

```
self.name = name
```

```
def disp(self):  
    print ("rollno : ",self.rollno , "name: ",self.name)
```

```
class A:  
    def disp(self):  
        print("Anu is good")
```

client.py : Access both Employee.py & student.py module data

```
import Employee  
import Student
```

```
e = Employee.Emp(111,"ratan")  
e.disp()  
m = Employee.MyClass()  
m.disp()
```

```
s = Student.Stu(1,"anu")  
s.disp()  
c = Student.A()  
c.disp()
```

Ex-7:

```
Package-name :      anu-->anushka --->first.py  
    anu  
    |-->anushka  
    |-->first.py
```

first.py

```
def disp():  
    print("hi anu how r u")
```

package-name : ratan ---> A.py client.py

```
ratan  
    |-->A.py  
    |-->client.py
```

A.py

```
def eat():  
    print("I like biryani")
```

client.py

```
import A
A.eat()

import sys
sys.path.append("D:\\anu\\anushka")
import first
first.disp()
```

Ex-9:

sravani--> chandana ---> Animal.py class:Animals disp()
 A.py class Frut: disp()

pari---> kshit ---> mh --> frist.py : def eat1()
second.py : def eat2()
client.py
first data
second data
animal data
A data

Example :

```
import Fruits
c = dir(Fruits)
print(c)

from Fruits import MyFruits
c = dir(MyFruits)
print(c)
```

Working with two different packages :**Accessing Modules from Another Directory**

Modules may be useful for more than one programming project, and in that case it makes less sense to keep a module in a particular directory that's tied to a specific project.

If you want to use a Python module from a location other than the same directory where your main program is, you have a few options.

Appending Paths

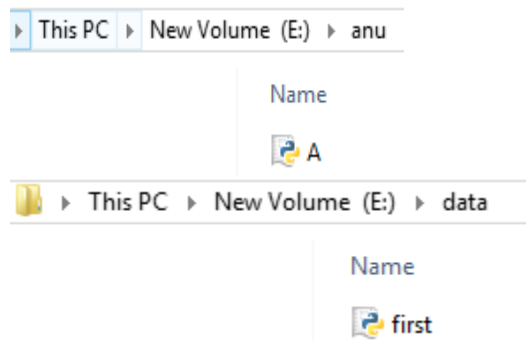
One option is to invoke the path of the module via the programming files that use that module. This should be considered more of a temporary solution that can be done during the development process as it does not make the module available system-wide.

To append the path of a module to another programming file, you'll start by importing the sys module alongside any other modules you wish to use in your main program file.

The `sys` module is part of the Python Standard Library and provides system-specific parameters and functions that you can use in your program to set the path of the module you wish to implement.

For example, let's say we moved the `hello.py` file and it is now on the path `/usr/sammy/` while the `main_program.py` file is in another directory.

In our `main_program.py` file, we can still import the `hello` module by importing the `sys` module and then appending `/usr/sammy/` to the path that Python checks for files.



First.py:

E:\data\first.py - EditPlus

```
import sys
sys.path.append('E:/anu')
```

```
import A
A.m1()
```

Predefined modules:

math module :

ex:

```
import math
content = dir(math)
print (content)
```

```
E:\data>python first.py
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Example :

```
import math
print(math.ceil(30.3))
print(math.fabs(10))
print(math.factorial(4))
print(math.floor(30.9))
print(math.pow(3,4))
print(math.sqrt(4))
print(math.sin(90))
print(math.cos(90))
print(math.pi)
print(math.e)
```

math.pi : The mathematical constant $\pi = 3.141592\dots$, to available precision.

math.e: The mathematical constant $e = 2.718281\dots$, to available precision.

math.cos(x):Return the cosine of x radians.

math.sin(x): Return the sine of x radians.

math.pow(x, y):Return x raised to the power y.

math.sqrt(x) : Return the square root of x.

math.ceil(x) :Return ceiling of x as a float, the smallest integer value greater than or equal to x

math.fabs(x) : Return the absolute value of x.

math.factorial(x) : Return x factorial. Raises ValueError if x is not integral or is negative.

math.floor(x) : Return the floor of x as a float, the largest integer value less than or equal to x.

ex:

```
from math import sqrt
print(sqrt(4))
```

os module :**ex:**

```
import os
content = dir(os)
print (content)
```

E:\data>python first.py

```
['F_OK', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXCL', 'O_NOINHERIT', 'O_RANDOM',
'O_RDONLY', 'O_RDWR', 'O_SEQUENTIAL', 'O_SHORT_LIVED', 'O_TEMPORARY', 'O_TEXT',
'O_TRUNC', 'O_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_NOWAITO', 'P_OVERLAY', 'P_WAI
```

```
T', 'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'TMP_MAX', 'UserDict', 'W_OK', 'X_OK',
'__Environ', '__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__',
'__copy_reg', '__execvpe', '__exists', '__exit__', '__get_exports_list', '__make_stat_result',
'__make_statvfs_result', '__pickle_stat_result', '__pickle_statvfs_result', 'abort', 'access', 'altsep',
'chdir', 'chmod', 'close', 'closerange', 'curdir', 'defpath', 'devnull', 'dup', 'dup2', 'environ', 'errno',
'error', 'execl', 'execle', 'execlp', 'execlpe', 'execv', 'execve', 'execvp', 'execvpe', 'extsep', 'fdopen',
'fstat', 'fsync', 'getcwd', 'getcwdu', 'getenv', 'getpid', 'isatty', 'kill', 'linesep', 'listdir', 'lseek', 'lstat',
'makedirs', 'mkdir', 'name', 'open', 'pardir', 'path', 'pathsep', 'pipe', 'popen', 'popen2', 'popen3',
'popen4', 'putenv', 'read', 'remove', 'removedirs', 'rename', 'renames', 'rmdir', 'sep', 'spawnl',
'spawnle', 'spawnv', 'spawnve', 'startfile', 'stat', 'stat_float_times', 'stat_result', 'statvfs_result',
'strerror', 'sys', 'system', 'tempnam', 'times', 'tmpfile', 'tmpnam', 'umask', 'unlink', 'unsetenv',
'urandom', 'utime', 'waitpid', 'walk', 'write']
```

Example:

```
import os
os.rename("sample.txt", "ratan.txt")
os.remove("ratan.txt")

os.mkdir("new")
os.chdir("data")
print(os.getcwd())
os.rmdir("new")
```

Example:

```
from os import remove
remove("ratan.txt")
```

random Module :

ex:

```
import random
c = dir(random)
print(c)
```

E:\data>python first.py

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'System
Random', 'TWOPI', 'WichmannHill', '_BuiltinMethodType', '_MethodType', '__all__'
, '__builtins__', '__doc__', '__file__', '__name__', '__package__', '_acos', '_c
eil', '_cos', '_e', '_exp', '_hashlib', '_hexlify', '_inst', '_log', '_pi', '_ra
ndom', '_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn', 'betava
riate', 'choice', 'division', 'expovariate', 'gammavariate', 'gauss', 'getrandbi
ts', 'getstate', 'jumpahead', 'lognormvariate', 'normalvariate', 'paretovariate'
, 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'tr
iangular', 'uniform', 'vonmisesvariate', 'weibullvariate']
```


ex:

```
import random

print(random.randint(1,100))
print(random.choice( ['red', 'black', 'green'])))

myList = [2, 10.5, False,"ratan","anu"]
print(random.choice(myList))

for i in range(3):
    print random.randrange(0, 101, 5)
```

ex:

```
from random import randint
print(randint(1,100))
```

time module :

Example :

```
import time
content = dir(time)
print (content)
```

output :

```
E:\data>python first.py
['__doc__', '__name__', '__package__', 'accept2dyear', 'altzone', 'asctime', 'clock', 'ctime', 'daylight', 'gmtime', 'localtime', 'mktime', 'sleep', 'strptime', 'struct_time', 'time', 'timezone', 'tzname']
```

Example :

```
import time
print("hi sir")
time.sleep(1)
print(time.strftime('%X %x %Z'))
```

```
print("hi sir")
```

Directive	Meaning	Notes
%a	Locale's abbreviated weekday name.	
%A	Locale's full weekday name.	
%b	Locale's abbreviated month name.	
%B	Locale's full month name.	
%c	Locale's appropriate date and time representation.	
%d	Day of the month as a decimal number [01,31].	
%H	Hour (24-hour clock) as a decimal number [00,23].	
%I	Hour (12-hour clock) as a decimal number [01,12].	
%j	Day of the year as a decimal number [001,366].	
%m	Month as a decimal number [01,12].	
%M	Minute as a decimal number [00,59].	
%p	Locale's equivalent of either AM or PM.	(1)
%S	Second as a decimal number [00,61].	(2)
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.	(3)
%w	Weekday as a decimal number [0(Sunday),6].	
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.	(3)
%x	Locale's appropriate date representation.	
%X	Locale's appropriate time representation.	
%y	Year without century as a decimal number [00,99].	
%Y	Year with century as a decimal number.	
%Z	Time zone name (no characters if no time zone exists).	
%%	A literal '%' character.	

sys module :

```
import sys
content = dir(sys)
print (content)
```

```
E:\data>python first.py
```

```
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__', '__stderr__',
'__stdin__', '__stdout__', '__clear_type_cache__', '_current_frames', '_getframe', '_git', 'api_version',
'argv', 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats', 'copyright', 'displayhook',
'dllhandle', 'dont_write_bytecode', 'exc_clear', 'exc_info', 'exc_type', 'excepthook', 'exec_prefix',
'executable', 'exit', 'flags', 'float_info', 'float_repr_style', 'getcheckinterval', 'getdefaultencoding',
'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof', 'gettrace',
'getwindowsversion', 'hexversion', 'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path',
'modules', 'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'py3kwarning',
'setcheckinterval', 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
'version', 'version_info', 'warnoptions', 'winver']
```

Regular Expression

- ✓ The term "regular expression", sometimes also called regex or regexp for string pattern matching.
- ✓ In python the regular expression are available in **re** module

Methods in Regular Expression:

```
re.match()  
re.search()  
re.findall()  
re.split()  
re.sub()  
re.compile()
```

ex: Normal string vs. raw string

A normal string is a string literal in which the normal escaping rules have consider.

For example, the string literal "\n" consists of one character escape sequence character (newline).

A raw string is a string literal (prefixed with an r) in which the normal escaping rules have been suspended so that everything is a literal.

Raw string can prefix with `r` or `R`.

For example, the string literal r"\n" consists of two characters: a backslash and a lowercase `n`. for ex, r"\" is a valid string literal consisting of two characters: a backslash and a double quote; r"\ is not a value string literal, even a raw string cannot ends with odd number of backslashes).

#normal string

```
print ('this\nexample')
print('this\twebsite')
print('C:\\Program Files')
```

#raw String

```
print (r'this\nexample')
print(r'this\twebsite')
print(r'C:\\Program Files')
```

*Ex: This method finds match if it occurs at start of the string
If the match is not available it returns None.*

re.match(pattern, string)

```
import re
result = re.match(r'ratan', 'ratan sir good')
print (result)
print (result.group(0))
print (result.start())
print (result.end())
```

```
result = re.match(r'durga', 'hi sir')
print (result) #None
```

```
E:\>python first.py
<_sre.SRE_Match object; span=(0, 5), match='ratan'>
```

ratan

0

5

None

* : zero or more

+ : one or more

? : zero or one

[abc] Only a, b, or c

[a-z] Characters a to z

[a-e] characters a to e

[^abc] Not a, b, nor c

[0-9] Numbers 0 to 9

[hc]?at matches **"at"**, "hat", and "cat".

[hc]*at matches **"at"**, "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on.

[hc]+at matches "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on, but not **"at"**.

```
import re
```

```
result = re.match(r'[a-z]+an', 'ratan sir good')
```

```
print (result.group())
```

```
result = re.match(r'[a,b,r]?', 'ratan sir good')
```

```
print (result.group())
```

```
result = re.match(r'[a-z]*', 'ratan sir good')
```

```
print (result.group())
```

```
E:\>python first.py
```

```
ratan
```

```
r
```

```
ratan
```

Ex:

*It is similar to match() but it doesn't restrict us to find matches at the beginning of the string only.
If the match is not available it returns None.*

re.search(pattern, string)

```
import re
result = re.search(r'ratan', 'hi ratan sir')
print(result)
print(result.group(0))
print(result.start())
print(result.end())
```

```
result = re.search(r'durga', 'hi ratan sir')
print(result)
```

```
E:\>python first.py
<_sre.SRE_Match object; span=(3, 8), match='ratan'>
ratan
3
8
None
```

{3} : exactly 3
{3,} : 3 or more
{3,5} : 3,4 or 5

Ex:

- ✓ *It helps to get a list of all matching patterns. It has no constraints of searching from start or end or middle.*
- ✓ *use re.findall() always, it can work like re.search() and re.match() both.*
- ✓ *findall() return the data in the form of list data type. If the data is not available it return empty list.*

re.findall (pattern, string)

```
import re
```

```
result = re.findall(r'ratan', 'hi ratan sir ratanit')
print(result)
print(tuple(result))
print(set(result))
```

```
result = re.findall(r'durga', 'hi ratan sir')
print(result)
```

```
E:\>python first.py
['ratan', 'ratan']
('ratan', 'ratan')
{'ratan'}
[]
```

Ex:

```
import re
```

```
pattern=re.compile('ratan')
```

```
result=pattern.findall('hi ratan sir welcome to ratanit')
print (result)
```

```
result2=pattern.findall('ratanit is good')
print (result2)
```

```
result3=pattern.findall('durgasoft is good')
print (result3)
```

```
E:\>python first.py
['ratan', 'ratan']
['ratan']
[]
```

```
import re
```

```
s = "rat mat bat cat durga"
```

```
x = re.findall("[rmbc]at",s)
```

```
y = re.findall("[a-f]at",s)
```

```
z = re.findall("[A-Z]at*+",s)
```

```
print(x)
```

```
print(y)
```

```
print(z)
```

```
E:\>python first.py
['rat', 'mat', 'bat', 'cat']
['bat', 'cat']
[]
```

ex: This methods helps to split string by the occurrences of given pattern.

`re.split(pattern, string, [maxsplit=0]):`

Method `split()` has another argument “maxsplit”. It has default value of zero.

```
import re
result=re.split(r'a','ratanit')
print(result)  #['r', 't', 'nit']
```

ex:

```
import re
result=re.split(r'a','ratanitratanit',maxsplit=2)
print(result)  #['r', 't', 'nitratanit']
```

ex:

It helps to search a pattern and replace with a new sub string. If the pattern is not found, string is returned unchanged.

`re.sub(pattern, repl, string):`

```
import re
result=re.sub(r'durga',r'ratan','durga world durga in India')
print(result)
```

```
result=re.sub(r'sunny',r'ratan','durga world durga in India')
print(result)
```

ex:

We can combine a regular expression pattern into pattern objects, which can be used for pattern matching. It also helps to search a pattern again without rewriting it.

```
import re
pattern=re.compile('ratan')
```

```
result=pattern.findall('hi ratan sir welcome to ratanit')
print (result)
```

```
result2=pattern.findall('ratanit is good')
print (result2)
```



```
result3=pattern.findall('durgasoft is good')
print (result3)
```

```
ex:
import re
result=re.findall(r'.','hi welcome to ratan it')
print (result)
```

```
ex: finding the index of the string
import re
s = "hi i like beer and beer is good"
```

```
for i in re.finditer("beer",s):
    x = i.span()
    print(x)
```

```
ex:
\d      Matches with digits [0-9]
[.].    Matches any single character in a square bracket
```

```
import re
nameage="Balu age is 40 Chiru age is 65"
```

```
age=re.findall(r"\d{1,3}",nameage)
names = re.findall(r"[A-Z][a-z]*",nameage)
```

```
my_dict={}
```

```
x=0
for name in names:
    my_dict[name]=age[x]
    x=x+1
```

```
print(my_dict)
```



Anchors	Quantifiers	Groups and Ranges
^ Start of string	* 0 or more	. Any character except new line (\n)
\A Start of string	+ 1 or more	(a b) a or b
\$ End of string	? 0 or 1	(...) Group
\Z End of string	{3} Exactly 3	(?:...) Passive Group
\b Word boundary	{3,} 3 or more	[abc] Range (a or b or c)
\B Not word boundary	{3,5} 3, 4 or 5	[^abc] Not a or b or c
\< Start of word		[a-q] Letter between a and q
\> End of word		[A-Q] Upper case letter between A and Q
Character Classes	Quantifier Modifiers	[0-7] Digit between 0 and 7
\c Control character	"x" below represents a quantifier	\n nth group/subpattern
\s White space	x? Ungreedy version of "x"	Note: Ranges are inclusive.
\S Not white space	Escape Character	Pattern Modifiers
\d Digit	\ Escape Character	g Global match
\D Not digit	Metacharacters (must be escaped)	i Case-insensitive
\w Word	^ [.	m Multiple lines
\W Not word	\$ { *	s Treat string as single line
\x Hexadecimal digit	(\ +	x Allow comments and white space in pattern
\O Octal digit) ?	e Evaluate replacement
POSIX	< >	U Ungreedy pattern
[[:upper:]] Upper case letters	Special Characters	String Replacement (Backreferences)
[[:lower:]] Lower case letters	\n New line	\$n nth non-passive group
[[:alpha:]] All letters	\r Carriage return	\$2 "xyz" in /^(abc(xyz))\$/
[[:alnum:]] Digits and letters	\t Tab	\$1 "xyz" in /^(?:abc)(xyz)\$/
[[:digit:]] Digits	\v Vertical tab	\$' Before matched string
[[:xdigit:]] Hexadecimal digits	\f Form feed	\$' After matched string
[[:punct:]] Punctuation	\xxx Octal character xxx	\$+ Last matched string
[[:blank:]] Space and tab	\xhh Hex character hh	\$& Entire matched string
[[:space:]] Blank characters	Sample Patterns	
[[:cntrl:]] Control characters	Pattern Will Match	
[[:graph:]] Printed characters	([A-Za-z0-9-]+)	Letters, numbers and hyphens
[[:print:]] Printed characters and spaces	(\d{1,2}\d{1,2}\d{4})	Date (e.g. 21/3/2006)
[[:word:]] Digits, letters and underscore	([^\s]+(?:\.(jpg gif png))\.\d2)	jpg, gif or png image
Assertions	(^[1-9]{1}\$ ^[1-4]{1}[0-9]{1}\$ ^50\$)	Any number from 1 to 50 inclusive
?= Lookahead assertion	(#[A-Fa-f0-9]{3}([A-Fa-f0-9]{3})?)	Valid hexadecimal colour code
?! Negative lookahead	((?=[a-z])(?=[A-Z]).{8,15})	String with at least one upper case letter, one lower case letter, and one digit (useful for passwords).
?<= Lookbehind assertion	(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})	Email addresses
?!= or ?<! Negative lookbehind	(<(/?(^>)+)\>)	HTML Tags
?> Once-only Subexpression		
?() Condition [if then]		
?() Condition [if then else]		
?# Comment		
	Note: These patterns are intended for reference purposes and have not been extensively tested. Please use with caution and test thoroughly before use.	

Ex:

```
import re
s = "rat mat bat cat durga"
```

```
x = re.findall("[rmbc]at",s)
```

```
for i in x:
    print(i)
```

ex:

```
import re
s = "rat mat bat cat durga"
```

```
x = re.findall("[rmbc]at",s)
y = re.findall("[a-f]at",s)
z = re.findall("[A-Z]at",s)
print(x)
print(y)
print(z)
```

```
for i in x:
    print(i)
```

ex:

```
import re
s = "rat mat bat cat"
```

```
x = re.findall("[a-k]at",s)
```

```
for i in x:
    print(i)
```

ex :

```
import re
s = "rat mat bat cat"
```

```
x = re.findall("[^a-k]at",s)
```

```
for i in x:
    print(i)
```

abc...	Letters
123...	Digits
\d	Any Digit

<code>\D</code>	Any Non-digit character
<code>.</code>	Any Character
<code>\.</code>	Period
<code>[abc]</code>	Only a, b, or c
<code>[^abc]</code>	Not a, b, nor c
<code>[a-z]</code>	Characters a to z
<code>[0-9]</code>	Numbers 0 to 9
<code>\w</code>	Any Alphanumeric character
<code>\W</code>	Any Non-alphanumeric character
<code>{m}</code>	m Repetitions
<code>{m,n}</code>	m to n Repetitions
<code>*</code>	Zero or more repetitions
<code>+</code>	One or more repetitions
<code>?</code>	Optional character
<code>\s</code>	Any Whitespace
<code>\S</code>	Any Non-whitespace character
<code>^...\$</code>	Starts and ends
<code>(...)</code>	Capture Group
<code>(a(bc))</code>	Capture Sub-group
<code>(.*)</code>	Capture all
<code>(abc def)</code>	Matches abc or def

Ex:

```
import re
```

```
s=" hi sir \n please stop \n the class"
print(s)
```

```
r = re.compile("\n")
x = r.sub(" ",s)
print(x)
```

ex:

```
import re
```

```
num = "12345"
print("Matches : ",len(re.findall("\d",num)))
print("Matches : ",len(re.findall("\D",num)))
```

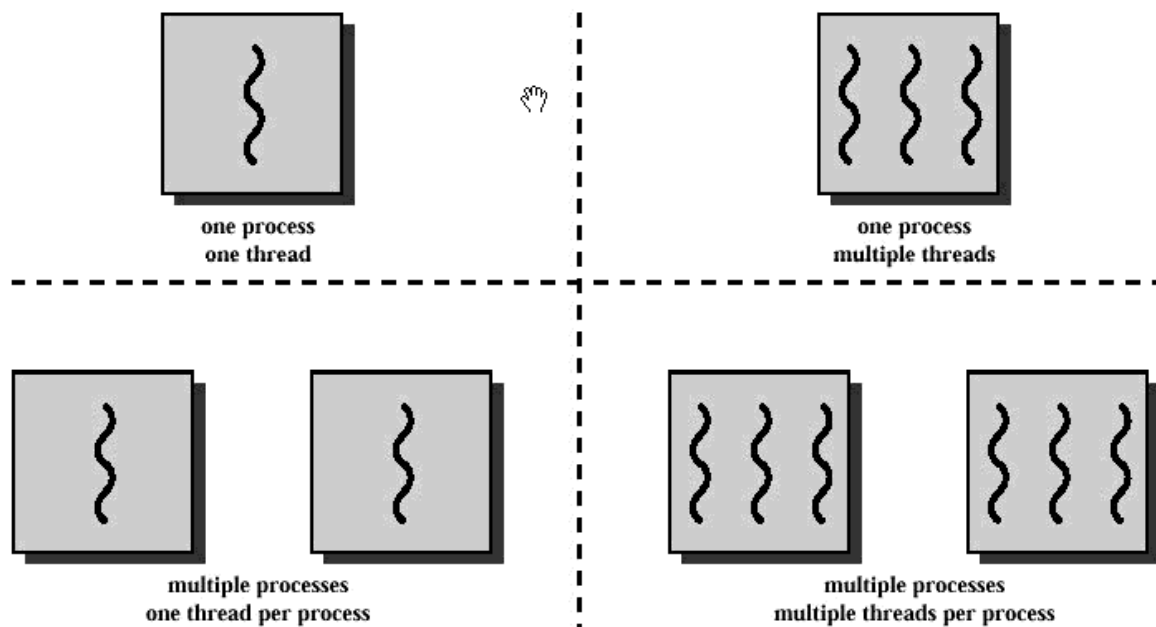
ex :

```
import re
```

```
num = "12345 6789 657 78 909090"  
print(len(re.findall("\d{5}",num)))  
print(len(re.findall("\d{4,7}",num)))
```

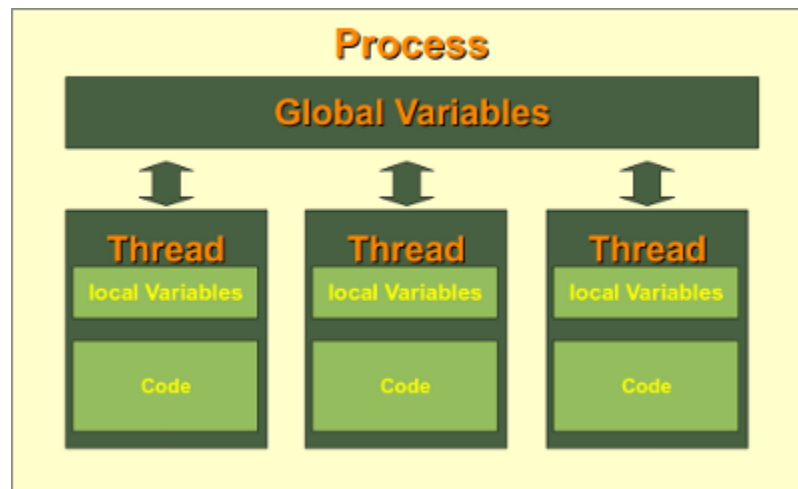
Multithreading

- ✓ Thread is a light weight task or small task of the application.
- ✓ Every thread is separate independent part of the same application(process).
- ✓ Threads exists inside a process. Multiple threads can exist in a single process.
- ✓ Executing more than one thread simultaneously is called multithreading.
- ✓ Threads are usually contained in processes. More than one thread can exist within the same process
- ✓ Threads allows a program to run multiple operations concurrently in the same process space.



The major difference between threads and processes is

1. Threads share the address space of the process that created it; processes have their own address.
2. Threads have direct access to the data segment of its process; processes have their own copy of the data segment of the parent process.
3. Threads can directly communicate with other threads of its process; processes must use interprocess communication to communicate with sibling processes.
4. Threads have almost no overhead; processes have considerable overhead.
5. New threads are easily created; new processes require duplication of the parent process.
6. Threads can exercise considerable control over threads of the same process; processes can only exercise control over child processes.
7. Changes to the main thread (cancellation, priority change, etc.) may affect the behavior of the other threads of the process; changes to the parent process does not effect child processes.

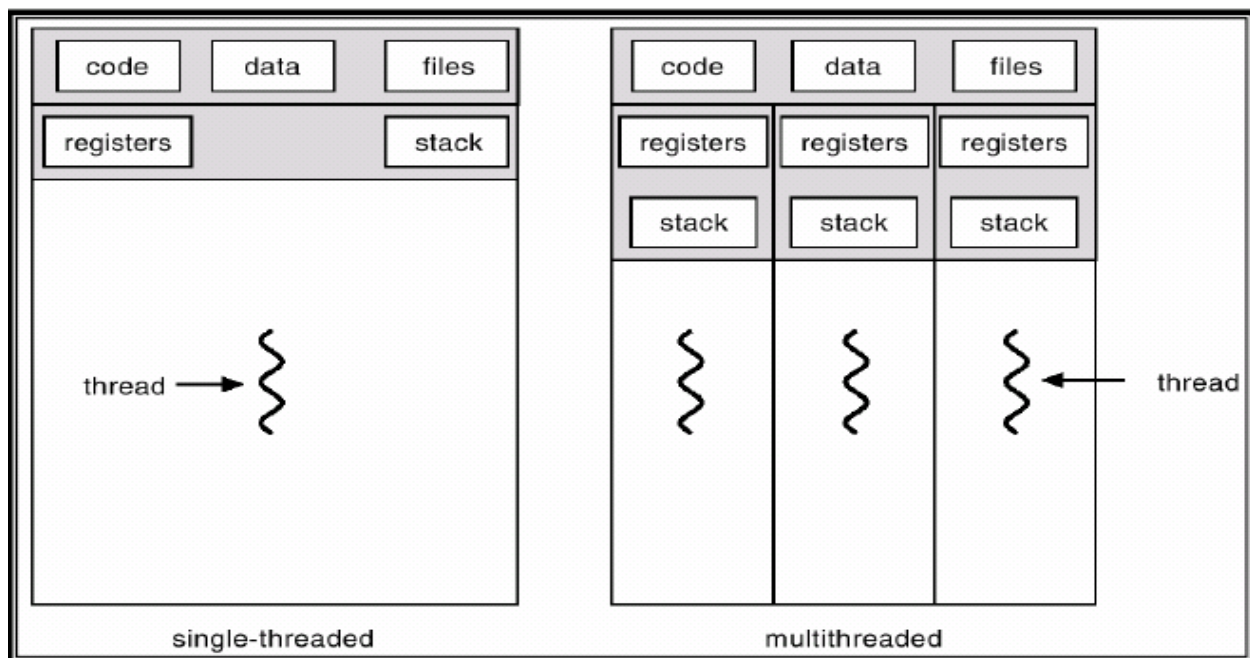


Advantages :

- ✓ Multiple threads will run simultaneously it improves performance.
- ✓ Used to develop gaming applications where the multiple objects are moving at a time.

There are two types of programming models

1. single threaded model
2. multithreaded model



There are two modules support multithreading concept.

1. thread (Deprecated)
2. threading

Note : thread, threading modules present in python 2.7 but python 3.x support only threading module.

ex:

```
# importing the threading module
import threading

def print_square(num):
    print("Square:", num * num)

def print_cube(num):
    print("Cube: ", num * num * num)

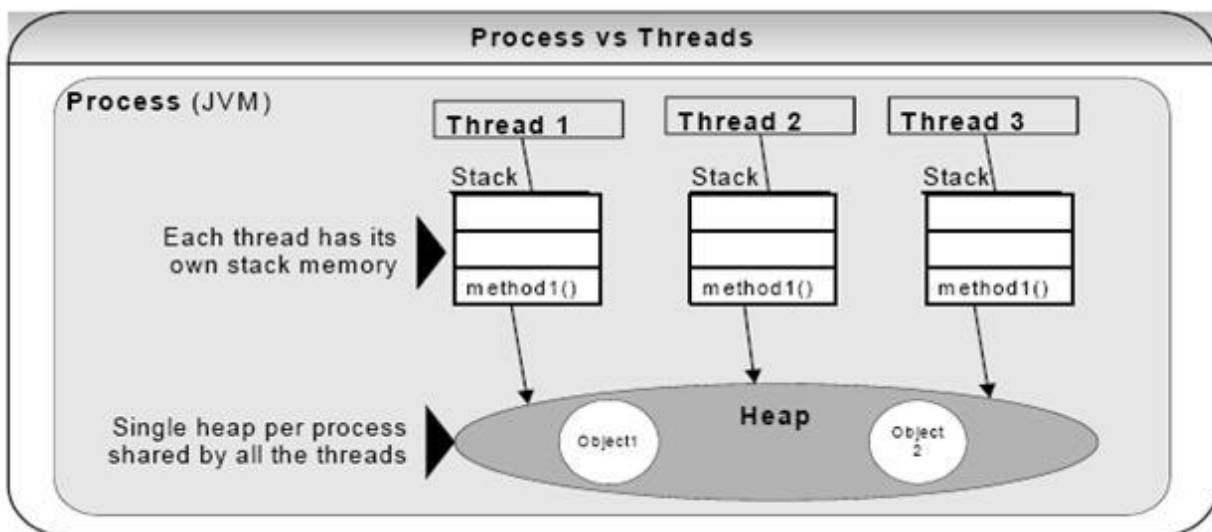
# creating thread
t1 = threading.Thread(target=print_square, args=(10,))
t2 = threading.Thread(target=print_cube, args=(10,))

# starting thread 1, thread2
t1.start()
t2.start()
```

ex: **from import statement**
from threading import Thread

```
def print_cube(num):
    print("Cube: {}".format(num * num * num))

t1 = Thread(target=print_cube, args=(4,))
t1.start()
```



ex : Thread names : The default name of threads starts from : Thread1 , thread2 ...thread n

```
import threading
def f1():
    print("thread starting : ",threading.currentThread().getName())

def f2():
    print("thread starting : ",threading.currentThread().getName())

t1 = threading.Thread(target=f1)
t2 = threading.Thread(target=f2)
t1.start()
t2.start()
```

```
E:\>python first.py
thread starting : Thread-1
thread starting : Thread-2
```

ex: Thread user defined names

- ✓ to get the current thread name use **threading.currentThread().getName()**
- ✓ to stop the execution use **sleep()** function of **time** module.

```
import threading
import time

def f1():
    print("thread starting : ",threading.currentThread().getName())
    time.sleep(1)
    print("thread-1 ending")

def f2():
    print("thread starting : ",threading.currentThread().getName())
    time.sleep(1)
    print("thread-2 ending")

t1 = threading.Thread(target=f1, name="t1")
t2 = threading.Thread(target=f2,name="t2")
t1.start()
t2.start()
```

ex: creating multiple Threads

```
import threading
import time
def f1():
    print("hi ratan sir staeted")
    time.sleep(3)
    print("hi ratan sir completed")

for i in range(1,4):
    t1 = threading.Thread(target=f1, name="t1")
    t1.start()
```

ex :

- ✓ **main Thread** : by default the application contains main thread. (python 3.x)
- ✓ **enumerate()** function return list of active threads.

```
import threading
import time
print(threading.main_thread())

def m1():
    for i in range(3):
        print("ratanit")
        time.sleep(1)

t1 = threading.Thread(target=m1)
t1.start()
print(threading.active_count())

for thread in threading.enumerate():
    print("Active Thread name is %s." % thread.getName())
```

ex: timer : To start the thread after specified time.

```
import threading

def delayed():
    print("I am printed after 3 seconds!")

thread = threading.Timer(3, delayed)
thread.start()
```

ex:

- ✓ The daemon threads are executed at background to give the support to foreground threads.
- ✓ Once the main thread completes its execution all daemon threads are automatically stopped.
- ✓ All threads are by default non-daemon, to set the daemon nature to thread use **setDaemon(true)** by default it is false.

```
import threading
import time
```

```
def daemon():
    print("daemon thread started")
    time.sleep(5)
    print("daemon thread completed")
```

```
def non_daemon():
    print("non-daemon thread started")
    time.sleep(1)
    print("non-daemon thread completed")
```

```
t = threading.Thread(name='non-daemon', target=non_daemon)
d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)
```

```
d.start()
t.start()
```

ex:

```
# importing the threading module
import threading
import time
```

```
def thread1():
```

```

for i in range(1,10):
    print("Thread-1 running:",i)
    time.sleep(1)

def thread2():
    for i in range(1,10):
        print("Thread-2 running :",i)
        time.sleep(1)

# creating thread
t1 = threading.Thread(target=thread1)
t2 = threading.Thread(target=thread2)
t1.start()
t2.start()

```

case 1: Observation

- ✓ join() : function is used to stop the thread until completion of current thread.
- ✓ Join(2) : used to stop the thread 2-sec after that both threads are running parallel.

```

# creating thread
t1 = threading.Thread(target=thread1)
t2 = threading.Thread(target=thread2)
t1.start()
t1.join()
#t1.join(2)
t2.start()
print("Rest of the app")

```

case 2: Observation

```

# creating thread
t1 = threading.Thread(target=thread1)
t2 = threading.Thread(target=thread2)
t1.start() # starting thread 1
t1.join()
t2.start() # starting thread 2
t2.join()
print("Rest of the app")

```

ex:

```

import threading
t1 = threading.Thread()
print(t1)
t1.start()

```

```
print(t1)
```

```
<Thread(Thread-1, initial)>  
<Thread(Thread-1, stopped 139849493174016)>
```

ex:

```
import threading  
t1 = threading.Thread()  
t1.start()  
t1.start()
```

```
E:\>python first.py  
raise RuntimeError("threads can only be started once")  
RuntimeError: threads can only be started once
```

ex:

```
import logging  
import threading  
import time
```

```
logging.basicConfig(filename='log.txt',level=logging.DEBUG)  
def worker():  
    logging.debug('worker Starting {}'.format(threading.currentThread().getName()))  
    time.sleep(2)  
    logging.debug('worker Exiting {}'.format(threading.currentThread().getName()))  
  
def my_service():  
    logging.debug('worker Starting {}'.format(threading.currentThread().getName()))  
    time.sleep(3)  
    logging.debug('worker Exiting {}'.format(threading.currentThread().getName()))  
  
t = threading.Thread( target=my_service,name="my service")  
w = threading.Thread(target=worker,name="worker-1")  
w2 = threading.Thread(target=worker,name="worker-2") # use default name  
t.start()  
w.start()  
w2.start()
```

ex : **Creating Thread by sub-classing thread class**

- ✓ Write the logics of threads by overriding run() function.
- ✓ Start the thread by using start() function of thread class.

```
import threading
class MyThread(threading.Thread):
    def run(self):
        for i in range(10):
            print("ratanit:",i)

t = MyThread()
t.start()
```

ex: **multiple threads are performing same task.**

```
import threading
class MyThread(threading.Thread):
    def run(self):
        for i in range(3):
            print("ratanit:",i)

t1 = MyThread()
t1.start()
t2 = MyThread()
t2.start()
t3 = MyThread()
t3.start()
```

ex : **Different threads are perform different tasks**

```
from threading import Thread
import threading
class MyThread1(Thread):
    def run(self):
        print("Ratanit :",threading.currentThread().getName())

class MyThread2(Thread):
    def run(self):
        print("DurgaSoft :",threading.currentThread().getName())

class MyThread3(Thread):
    def run(self):
        print("anusoft :",threading.currentThread().getName())

MyThread1().start()
MyThread2().start()
MyThread3().start()
```

Output :

```
Ratanit : Thread-1
DurgaSoft : Thread-2
anusoft : Thread-3
```

Observation : giving user defined names to Threads

```
MyThread1(name="ratan").start()
MyThread2(name="durga").start()
MyThread3(name="anu").start()
```

Output :

Ratanit : ratan
DurgaSoft : durga
anusoftware : anu

It is highly recommended to store complete application flow and exceptions information to a file.

This process is called logging.

The main advantages of logging are:

Depending on type of information, logging data is divided according to the following 6 levels in Python.

table

- 1. CRITICAL==>50==>Represents a very serious problem that needs high attention*
- 2. ERROR==>40==>Represents a serious error*
- 3. WARNING==>30==>Represents a warning message ,some caution needed.it is alert to the programmer*
- 4. INFO==>20==>Represents a message with some important information*
- 5. DEBUG==>10==>Represents a message with debugging*

To perform logging,first we required to create a file to store messages and we have to specify which level messages we have to store.

We can do this by using basicConfig() function of logging module.

logging.basicConfig(filename='log.txt',level=logging.WARNING)

The above line will create a file log.txt and we can store either WARNING level or higher level messages to that file.

After creating log file,we can write messages to that file by using the following methods.

logging.debug(message)

logging.info(message)

logging.warning(message)

logging.error(message)

logging.critical(message)

ex:

```
import logging
logging.basicConfig(filename='log.txt',level=logging.WARNING)
print("Logging Module Demo")
logging.debug("This is debug message")
logging.info("This is info message")
logging.warning("This is warning message")
logging.error("This is error message")
logging.critical("This is critical message")
```

Note:

*In the above program only WARNING and higher level messages will be written to log file.
If we set level as DEBUG then all messages will be written to log file.*

Ex:

```
import logging
logging.basicConfig(filename='ratan.txt',level=logging.INFO)
logging.info("Request processing started:")
try:
    x=int(input("Enter First Number: "))
    y=int(input("Enter Second Number: "))
    print(x/y)
except ZeroDivisionError as msg:
    print("ZeroDivisionError occurred")
    logging.exception(msg)
except ValueError as msg:
```

```
print("Enter only integer values")
logging.exception(msg)
logging.info("Request processing completed:")
```

ex:

```
def squareIt(x):
    return x**x
```

```
assert squareIt(2)==4,"The square of 2 should be 4"
assert squareIt(3)==9,"The square of 3 should be 9"
assert squareIt(4)==16,"The square of 4 should be 16"
print(squareIt(2))
print(squareIt(3))
print(squareIt(4))
```

ex:

```
def squareIt(x):
    return x*x
assert squareIt(2)==4,"The square of 2 should be 4"
assert squareIt(3)==9,"The square of 3 should be 9"
assert squareIt(4)==16,"The square of 4 should be 16"
print(squareIt(2))
print(squareIt(3))
print(squareIt(4))
```


Lambda expressions, Filter, Map, Reduce

- ✓ *Lambda function will simplify the expressions, reduce the length of the code.*

Syntax : **Lambda** arguments_list: expression

ex:

```
>>> f = lambda x, y : x + y
>>> f(1,1)
2
```

- ✓ *This function can have any number of arguments but only one expression, which is evaluated and returned.*
- ✓ *Lambda functions are used along with built-in functions like filter(), map() etc.*
- ✓ *A lambda function is a small anonymous function(name less function)to perform operations.*

Ex-1: function coding vs. lambda coding

```
def m1(x):
    print(2*x)
m1(10)
```

```
a = lambda x:x*2
print(a(10))
```

Ex-2: function coding vs. lambda coding

```
def m1(x,y):
    print(x*y)
m1(10,20)
```

```
a = lambda x,y : x*y
print(a(10,20))
```

```
b = lambda x,y : x if x>y else y
print(b(3,4))
```

ex 3: **Filter** : used to filter the data by applying conditions.
 `filter(function_to_apply, input)`
 where **input** : list , tuple , string, set...

Filtering even elements from the list : **by using function code vs. by using lambda code**

```
l1 = [10,3,5,2,20]
def m1(x):
    if x%2==0:
        return True
    else:
        return False

print(list(filter(m1,l1)))

print(list(filter(lambda x:x%2==0,l1)))      # printing output list format
print(tuple(filter(lambda x:x%2==0,l1)))     #printing output tuple format
```

Ex-4 : filter the "rattan" from given list. **by using function code vs. by using lambda code**

```
l1 = ["ratan","anu","ratan","durga"]
def m1(x):
    if x=="ratan":
        return True
    else:
        return False
print(list(filter(m1,l1)))

print(list(filter(lambda x : x=="ratan",l1)))
print(tuple(filter(lambda x : x=="ratan",l1)))

l2 = list(filter(lambda x : x=="ratan",l1))
print(l2)
```

Ex 5: Map : applies a function to all the items in an input_list.

```
map(function_to_apply, list_of_inputs)
```

where list_of_inputs : list , tuple , string , set...

Perform multiplication of 5 to every element of the list :

by using function code vs. by using lambda code

```
l1 = [2,45,6,3,7]
```

```
def m1(x):
```

```
    return x*5
```

```
print(list(map(m1,l1)))
```

```
print(list(map(lambda x : x*5,l1)))
```

output list format data

```
print(tuple(map(lambda x : x*5,l1)))
```

output tuple format data

Ex- 6: concat "it" to every element of the list : **by using function code vs. by using lambda code**

```
l1 = ["ratan","durga","sravya"]
```

```
def m1(x):
```

```
    return x+'it'
```

```
print(list(map(m1,l1)))
```

```
print(list(map(lambda x : x+'it',l1)))
```

```
print(tuple(map(lambda x : x+'it',l1)))
```

Ex 7 : performing operations on multiple inputs

```
l1 = [1,2,3,4]
```

```
l2 = [10,20,30,40]
```

```
l3 = [100,200,300,400]
```

```
print(list(map(lambda x,y: x+y,l1,l2)))
```

```
print(tuple(map(lambda x,y,z: x+y+z,l1,l2,l3)))
```

```
E:\>python first.py
```

```
[11, 22, 33, 44]
```

```
(111, 222, 333, 444)
```

Ex-8: *printing words length of the given string.*

```
s = "hi ratan sir how are you"
words = s.split()
print(words)
print(list(map(lambda x :len(x), words)))
```

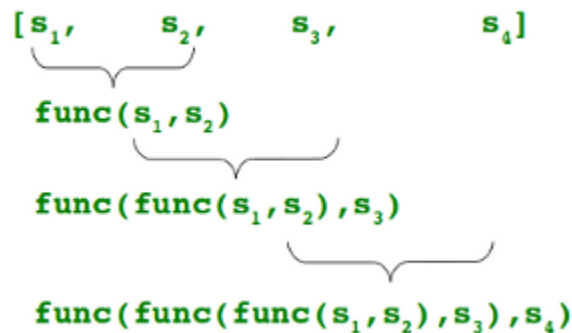
```
E:\>python first.py
```

['hi', 'ratan', 'sir', 'how', 'are', 'you']
[2, 5, 3, 3, 3, 3]

Ex -9: Reduce : used to performing some computation on a list and returning the result

Reduce function present in **functools** module, so must import the module

Reduce function internal implementation



Zip File creation

Ex:

```
import zipfile
zf = zipfile.ZipFile('ratan.zip', mode='w')
try:
    zf.write('ratan.txt')
    zf.write('log.txt')
finally:
    zf.close()
print("zip file is ready")
```

ex:

```
import zipfile

for filename in [ 'ratan.txt', 'ratan.zip', 'example.zip' ]:
    print ('%s %s' % (filename, zipfile.is_zipfile(filename)))
```

```
E:\>python first.py
ratan.txt False
ratan.zip True
example.zip False
```


Important examples:

*ex 1: # itertools.permutations() generates permutations
for an iterable. Time to brute-force those passwords*

```
import itertools  
for p in itertools.permutations('ABCD'):  
    print(p)
```

```
for p in itertools.permutations('12'):  
    print(p)
```

```
for p in itertools.permutations('123'):  
    print(p)
```

ex-2:

Numpy

- ✓ *Numpy is the core library for scientific computing in Python.*
- ✓ *NumPy is an acronym for "Numeric Python" or "Numerical Python".*
- ✓ *SciPy (Scientific Python) is often mentioned in the same breath with NumPy. SciPy extends the capabilities of NumPy with further useful functions for minimization, regression, Fourier-transformation and many others.*
- ✓ *Both NumPy and SciPy are usually not installed by default. NumPy has to be installed before installing SciPy. Numpy can be downloaded from the website:*

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

- *A powerful N-dimensional array object.*
- *sophisticated (broadcasting) functions.*
- *Tools for integrating C/C++ and Fortran code.*

- *useful linear algebra, Fourier transform, and random number capabilities*

Comparison between Core Python and Numpy :

When we say "Core Python", we mean Python without any special modules, i.e. especially without NumPy.

The advantages of Core Python:

- *high-level number objects: integers, floating point.*
- *containers: lists with cheap insertion and append methods, dictionaries with fast lookup*

Advantages of using Numpy with Python:

- *array oriented computing.*
- *efficiently implemented multi-dimensional arrays*
- *designed for scientific computation*

Before we can use NumPy we will have to import it. It has to be imported like any other module:

```
import numpy
```

But you will hardly ever see this. Numpy is usually renamed to np:

```
import numpy as np
```

We have a list with values, e.g. temperatures in Celsius:

```
cvalues = [20.1, 20.8, 21.9, 22.5, 22.7, 22.3, 21.8, 21.2, 20.9, 20.1]
```

We will turn our list "cvalues" into a one-dimensional numpy array:

```
C = np.array(cvalues)
```

```
print(C)
```

```
[ 20.1  20.8  21.9  22.5  22.7  22.3  21.8  21.2  20.9  20.1]
```

Let's assume, we want to turn the values into degrees Fahrenheit. This is very easy to accomplish with a numpy array. The solution to our problem can be achieved by simple scalar multiplication:

```
print(C * 9 / 5 + 32)
```

[68.18 69.44 71.42 72.5 72.86 72.14 71.24 70.16 69.62 68.18]

The array C has not been changed by this expression:

```
print(C)
```

[20.1 20.8 21.9 22.5 22.7 22.3 21.8 21.2 20.9 20.1]

Compared to this, the solution for our Python list looks awkward:

```
fvalues = [ x*9/5 + 32 for x in cvalues]
```

```
print(fvalues)
```

[68.18, 69.44, 71.42, 72.5, 72.86, 72.14, 71.24000000000001, 70.16, 69.62, 68.18]

So far, we referred to C as an array. The internal type is "ndarray" or to be even more precise "C is an instance of the class numpy.ndarray":

```
type(C)
```

The code above returned the following:

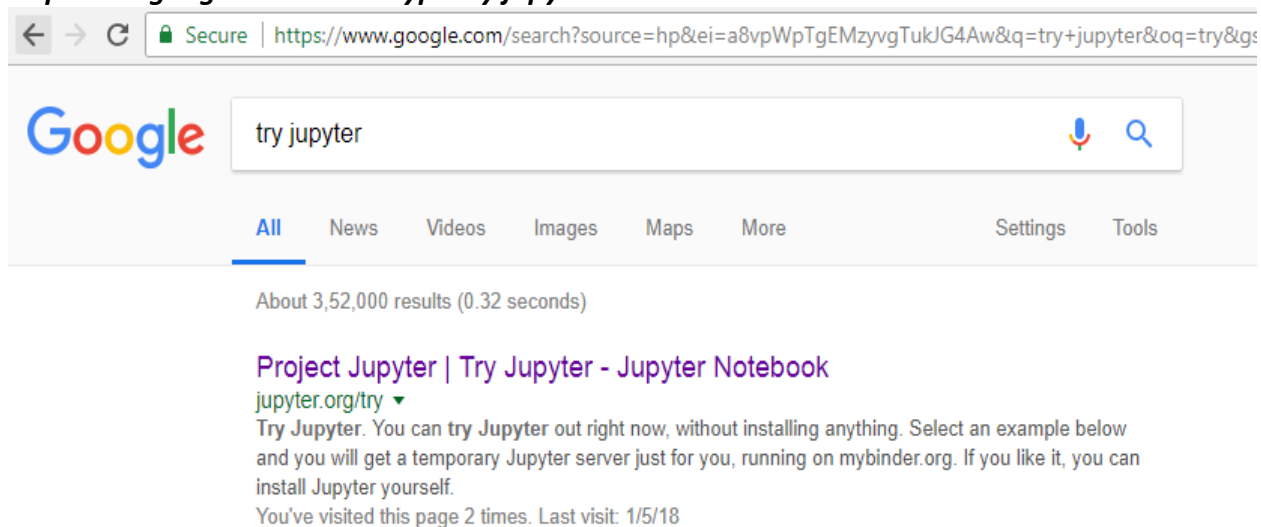
```
numpy.ndarray
```

In the following, we will use the terms "array" and "ndarray" in most cases synonymously.

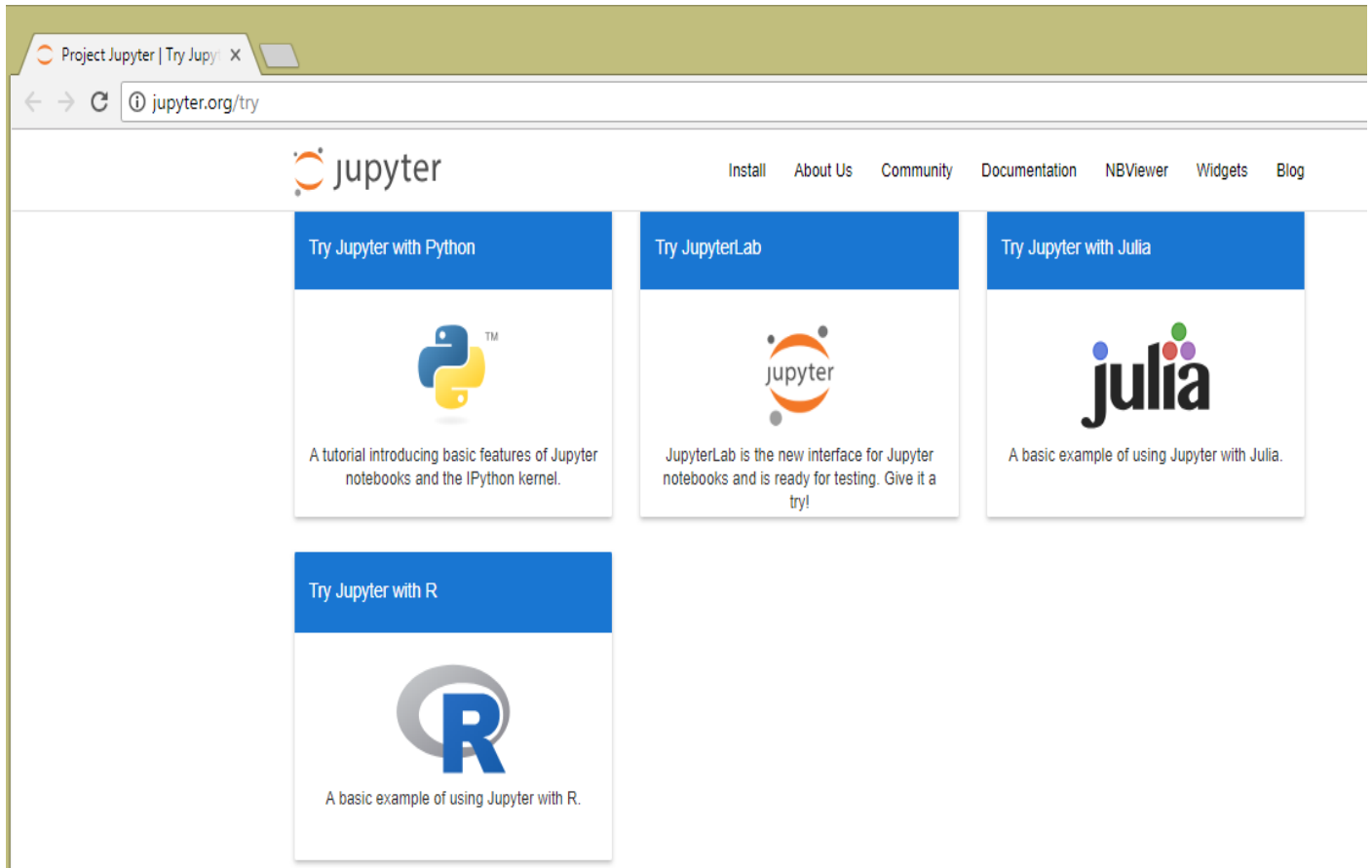
[https://hub.mybinder.org/user/ipython-ipython-in-depth-](https://hub.mybinder.org/user/ipython-ipython-in-depth-hvnr6mom/notebooks/examples/IPython%20Kernel/Rich%20Output.ipynb#)

[hvnr6mom/notebooks/examples/IPython%20Kernel/Rich%20Output.ipynb#](https://hub.mybinder.org/user/ipython-ipython-in-depth-hvnr6mom/notebooks/examples/IPython%20Kernel/Rich%20Output.ipynb#)

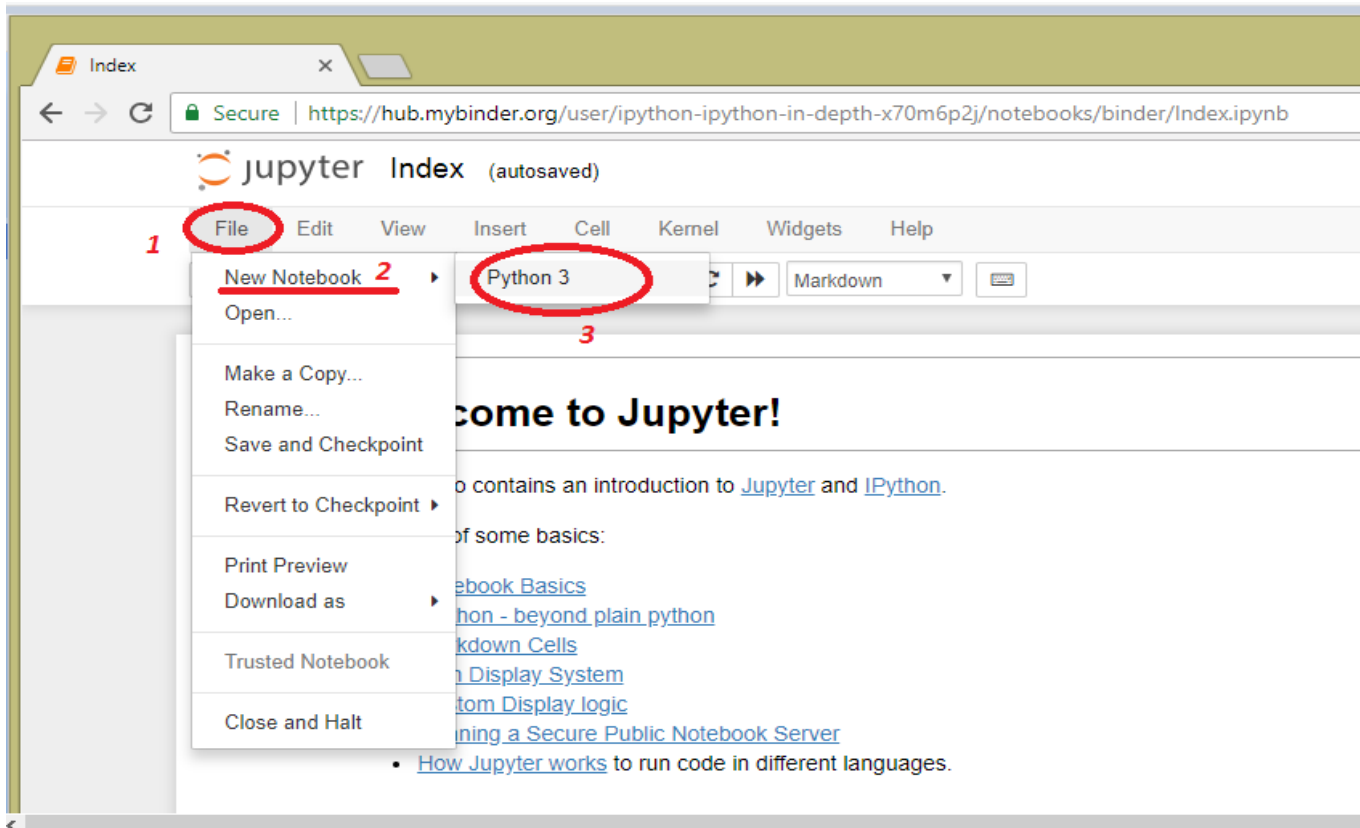
Step 1 : in google search bar type try jupyter.



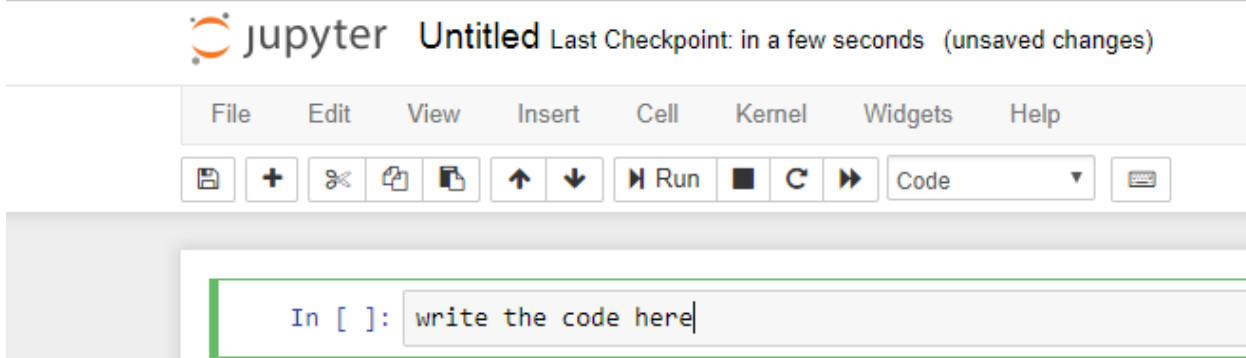
Step 2: select the jupyter window (any one). Here I am doing practical by using jupyter with python



Step 3: select the python 3 note book



Step 4: write the code in notebook.



Jupyter Untitled1 Last Checkpoint: 10 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help



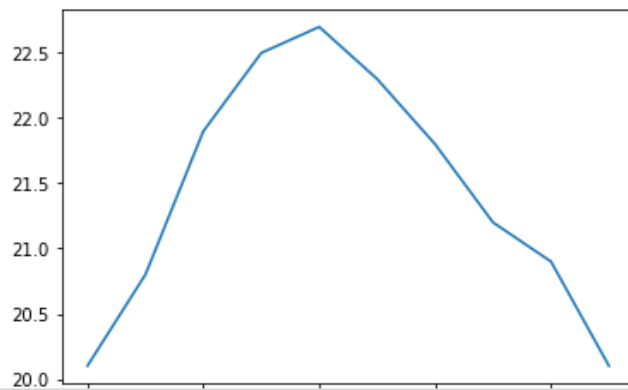
```
In [4]: import numpy
import numpy as np

cvalues = [20.1, 20.8, 21.9, 22.5, 22.7, 22.3, 21.8, 21.2, 20.9, 20.1]

C = np.array(cvalues)
print(C)

import matplotlib.pyplot as plt
plt.plot(C)
plt.show()
```

```
[20.1 20.8 21.9 22.5 22.7 22.3 21.8 21.2 20.9 20.1]
```



Thank You

***The pain you feel today
Will be the
STRENGTH
You feel tomorrow***