Language Fundamentals

Introduction

- Python is a general purpose high level programming language.
- Python was developed by Guido Van Rossam in 1989 while working at National Research Institute at Netherlands.
- But officially Python was made available to public in 1991. The official Date of Birth for Python is : Feb 20th 1991.
- Python is recommended as first programming language for beginners.

Eg1: To print Helloworld:

Java:

```
1)  public class HelloWorld
2)  {
3)    p s v main(String[] args)
4)    {
5)       SOP("Hello world");
6)    }
7)  }
```

C:

```
1)  #include<stdio.h>
2)  void main()
3)  {
4)     print("Hello world");
5)  }
```

Python:

```
print("Hello World")
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Eg2: To print the sum of 2 numbers

Java:

```
1)  public class Add
2)  {
3)     public static void main(String[] args)
4)     {
5)        int a,b;
6)        a =10;
7)        b=20;
8)        System.out.println("The Sum:"+(a+b));
9)     }
10) }
```

C:

```
1)  #include <stdio.h>
2)
3)  void main()
4)  {
5)     int a,b;
6)     a =10;
7)     b=20;
8)     printf("The Sum:%d",(a+b));
9)  }
```

Python:

```
1)  a=10
2)  b=20
3)  print("The Sum:",(a+b))
```

The name Python was selected from the TV Show
"The Complete
Monty
Python's
Circus", which was broadcasted in BBC from 1969 to 1974.

Guido developed Python language by taking almost all programming features from different languages

1. Functional Programming Features from C
2. Object Oriented Programming Features from C++
3. Scripting Language Features from Perl and Shell Script

4. Modular Programming Features from Modula-3

Most of syntax in Python Derived from C and ABC languages.

Where we can use Python:

We can use everywhere. The most common important application areas are

1. For developing Desktop Applications
2. For developing web Applications
3. For developing database Applications
4. For Network Programming
5. For developing games
6. For Data Analysis Applications
7. For Machine Learning
8. For developing Artificial Intelligence Applications
9. For IOT
...

Note:

Internally Google and Youtube use Python coding
NASA and Nework Stock Exchange Applications developed by Python.
Top Software companies like Google, Microsoft, IBM, Yahoo using Python.

Features of Python:

1. Simple and easy to learn:

Python is a simple programming language. When we read Python program,we can feel like reading english statements.
The syntaxes are very simple and only 30+ kerywords are available.
When compared with other languages, we can write programs with very less number of lines. Hence more readability and simplicity.
We can reduce development and cost of the project.

2. Freeware and Open Source:

We can use Python software without any licence and it is freeware.

Its source code is open,so that we can we can customize based on our requirement.

Eg: Jython is customized version of Python to work with Java Applications.

3. High Level Programming language:

Python is high level programming language and hence it is programmer friendly language. Being a programmer we are not required to concentrate low level activities like memory management and security etc..

4. Platform Independent:

Once we write a Python program,it can run on any platform without rewriting once again. Internally PVM is responsible to convert into machine understandable form.

5. Portability:

Python programs are portable. ie we can migrate from one platform to another platform very easily. Python programs will provide same results on any paltform.

6. Dynamically Typed:

In Python we are not required to declare type for variables. Whenever we are assigning the value, based on value, type will be allocated automatically.Hence Python is considered as dynamically typed language.

But Java, C etc are Statically Typed Languages b'z we have to provide type at the beginning only.

This dynamic typing nature will provide more flexibility to the programmer.

7. Both Procedure Oriented and Object Oriented:

Python language supports both Procedure oriented (like C, pascal etc) and object oriented (like C++,Java) features. Hence we can get benefits of both like security and reusability etc

8. Interpreted:

We are not required to compile Python programs explcitly. Internally Python interpreter will take care that compilation.

If compilation fails interpreter raised syntax errors. Once compilation success then PVM (Python Virtual Machine) is responsible to execute.

9. Extensible:

We can use other language programs in Python.
The main advantages of this approach are:

1. We can use already existing legacy non-Python code
2. We can improve performance of the application

10. Embedded:

We can use Python programs in any other language programs.
i.e we can embedd Python programs anywhere.

11. Extensive Library:

Python has a rich inbuilt library.
Being a programmer we can use this library directly and we are not responsible to implement the functionality.

etc...

Limitations of Python:

1. Performance wise not up to the mark b'z it is interpreted language.
2. Not using for mobile Applications

Flavors of Python:

1.CPython:
It is the standard flavor of Python. It can be used to work with C lanugage Applications

2. Jython or JPython:
It is for Java Applications. It can run on JVM

3. IronPython:
It is for C#.Net platform

4.PyPy:
The main advantage of PyPy is performance will be improved because JIT compiler is available inside PVM.

5.RubyPython
For Ruby Platforms

6. AnacondaPython
It is specially designed for handling large volume of data processing.
...

Python Versions:

Python 1.0V introduced in Jan 1994
Python 2.0V introduced in October 2000
Python 3.0V introduced in December 2008

Note: Python 3 won't provide backward compatibility to Python2
i.e there is no guarantee that Python2 programs will run in Python3.

Current versions

Python 3.6.1 **Python 2.7.13**

Identifiers

A name in Python program is called identifier.
It can be class name or function name or module name or variable name.

a = 10

Rules to define identifiers in Python:

1. The only allowed characters in Python are

- **alphabet symbols(either lower case or upper case)**
- **digits(0 to 9)**
- **underscore symbol(_)**

By mistake if we are using any other symbol like $ then we will get syntax error.

- **cash = 10 ✓**
- **ca$h =20 ✗**

2. Identifier should not starts with digit

- **123total ✗**
- **total123 ✓**

3. Identifiers are case sensitive. Of course Python language is case sensitive language.

- **total=10**
- **TOTAL=999**
- **print(total) #10**
- **print(TOTAL) #999**

Identifier:

1. Alphabet Symbols (Either Upper case OR Lower case)

2. If Identifier is start with Underscore (_) then it indicates it is private.

3. Identifier should not start with Digits.

4. Identifiers are case sensitive.

5. We cannot use reserved words as identifiers
   Eg: def=10 ✗

6. There is no length limit for Python identifiers. But not recommended to use too lengthy identifiers.

7. Dollor ($) Symbol is not allowed in Python.

Q. Which of the following are valid Python identifiers?

1) 123total ✗
2) total123 ✓
3) java2share ✓
4) ca$h ✗
5) _abc_abc_ ✓
6) def ✗
7) if ✗

Note:

1. If identifier starts with _ symbol then it indicates that it is private
2. If identifier starts with __(two under score symbols) indicating that strongly private identifier.
3.If the identifier starts and ends with two underscore symbols then the identifier is language defined special name,which is also known as magic methods.

Eg: __add__

Reserved Words

In Python some words are reserved to represent some meaning or functionality. Such type of words are called Reserved words.

There are 33 reserved words available in Python.

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

Note:

1. All Reserved words in Python contain only alphabet symbols.

2. Except the following 3 reserved words, all contain only lower case alphabet symbols.

- **True**
- **False**
- **None**

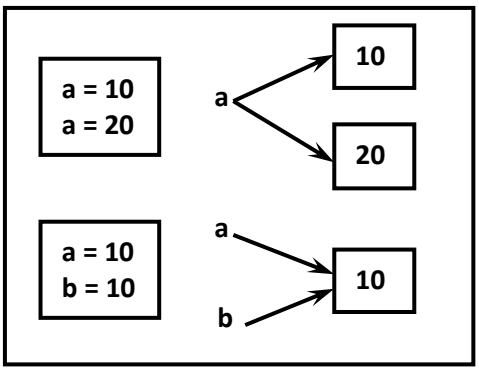
Eg: a= true ✗
a=True ✓

```
>>> import keyword
>>> 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']
```

Data Types

Data Type represent the type of data present inside a variable.
In Python we are not required to specify the type explicitly. Based on value provided,the type will be assigned automatically.Hence Python is Dynamically Typed Language.

Python contains the following inbuilt data types
1. int
2. float
3.complex
4.bool
5.str
6.bytes
7.bytearray
8.range
9.list
10.tuple
11.set
12.frozenset
13.dict
14.None



Note: **Python contains several inbuilt functions**

1.type()
to check the type of variable

2. id()
to get address of object

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

3. print()
to print the value

In Python everything is object

int data type:

We can use int data type to represent whole numbers (integral values)
Eg:

```
a=10
type(a) #int
```

Note:
In Python2 we have long data type to represent very large integral values.
But in Python3 there is no long type explicitly and we can represent long values also by using int type only.

We can represent int values in the following ways

1. Decimal form
2. Binary form
3. Octal form
4. Hexa decimal form

1. Decimal form(base-10):

It is the default number system in Python
The allowed digits are: 0 to 9
Eg: a =10

2. Binary form(Base-2):

The allowed digits are : 0 & 1
Literal value should be prefixed with 0b or 0B

Eg: a = 0B1111
a =0B123
a=b111

3. Octal Form(Base-8):

The allowed digits are : 0 to 7
Literal value should be prefixed with 0o or 0O.

Eg: a=0o123
a=0o786

4. Hexa Decimal Form(Base-16):

The allowed digits are : 0 to 9, a-f (both lower and upper cases are allowed)
Literal value should be prefixed with 0x or 0X

Eg:
a =0XFACE
a=0XBeef
a =0XBeer

Note: Being a programmer we can specify literal values in decimal, binary, octal and hexa decimal forms. But PVM will always provide values only in decimal form.

```
a=10
b=0o10
c=0X10
d=0B10
print(a)10
print(b)8
print(c)16
print(d)2
```

Base Conversions

Python provide the following in-built functions for base conversions

1.bin():

We can use bin() to convert from any base to binary

Eg:

```
>>> bin(15)
'0b1111'
>>> bin(0o11)
'0b1001'
>>> bin(0X10)
'0b10000'
```

2. oct():

We can use oct() to convert from any base to octal

Eg:

```
1)  >>> oct(10)
2)  '0o12'
3)  >>> oct(0B1111)
4)  '0o17'
5)  >>> oct(0X123)
6)  '0o443'
```

3. hex():

We can use hex() to convert from any base to hexa decimal

Eg:

```
1)  >>> hex(100)
2)  '0x64'
3)  >>> hex(0B111111)
4)  '0x3f'
5)  >>> hex(0o12345)
6)  '0x14e5'
```

float data type:

We can use float data type to represent floating point values (decimal values)

Eg: f=1.234
type(f) float

We can also represent floating point values by using exponential form (scientific notation)

Eg: f=1.2e3
print(f) 1200.0
instead of 'e' we can use 'E'

The main advantage of exponential form is we can represent big values in less memory.

***Note:**
We can represent int values in decimal, binary, octal and hexa decimal forms. But we can represent float values only by using decimal form.

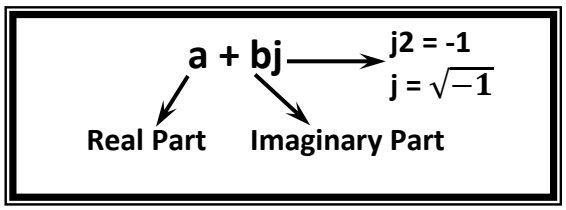
Eg:

```
1)   >>> f=0B11.01
2)  File "<stdin>", line 1
3)   f=0B11.01
```

```
4)          ^
5)  SyntaxError: invalid syntax
6)
7)  >>> f=0o123.456
8)   SyntaxError: invalid syntax
9)
10) >>> f=0X123.456
11)  SyntaxError: invalid syntax
```

Complex Data Type:

A complex number is of the form



a and b contain intergers or floating point values

Eg:

3+5j
10+5.5j
0.5+0.1j

In the real part if we use int value then we can specify that either by decimal,octal,binary or hexa decimal form.
But imaginary part should be specified only by using decimal form.

```
1)  >>> a=0B11+5j
2)  >>> a
3)  (3+5j)
4)  >>> a=3+0B11j
5)   SyntaxError: invalid syntax
```

Even we can perform operations on complex type values.

```
1)  >>> a=10+1.5j
2)  >>> b=20+2.5j
3)  >>> c=a+b
4)  >>> print(c)
5)  (30+4j)
6)  >>> type(c)
7)  <class 'complex'>
```

Note: Complex data type has some inbuilt attributes to retrieve the real part and imaginary part

```
c=10.5+3.6j

c.real==>10.5
c.imag==>3.6
```

We can use complex type generally in scientific Applications and electrical engineering Applications.

4.bool data type:

We can use this data type to represent boolean values.
The only allowed values for this data type are:
True and False

Internally Python represents True as 1 and False as 0

```
b=True
type(b)   =>bool
```

Eg:

```
a=10
b=20
c=a<b
print(c)==>True


True+True==>2
True-False==>1
```

str type:

str represents String data type.

A String is a sequence of characters enclosed within single quotes or double quotes.

```
s1='durga'
s1="durga"
```

By using single quotes or double quotes we cannot represent multi line string literals.

```
s1="durga
```

soft"

For this requirement we should go for triple single quotes(''') or triple double quotes(""")

```
s1='''durga
      soft'''

s1="""durga
       soft"""
```

We can also use triple quotes to use single quote or double quote in our String.

```
''' This is " character'''
' This i " Character '
```

We can embed one string in another string

```
'''This "Python class very helpful" for java students'''
```

Slicing of Strings:

slice means a piece
[ ] operator is called slice operator,which can be used to retrieve parts of String.
In Python Strings follows zero based index.
The index can be either +ve or -ve.
+ve index means forward direction from Left to Right
-ve index means backward direction from Right to Left

-5	-4	-3	-2	-1
d	u	r	g	a
0	1	2	3	4

```
1)  >>> s="durga"
2)  >>> s[0]
3)  'd'
4)  >>> s[1]
5)  'u'
6)  >>> s[-1]
7)  'a'
8)  >>> s[40]
```

IndexError: string index out of range

```
>>> s[1:40]
'urga'
>>> s[1:]
'urga'
>>> s[:4]
'durg'
>>> s[:]
'durga'
>>>


>>> s*3
'durgadurgadurga'

>>> len(s)
5
```

Note:

1. In Python the following data types are considered as Fundamental Data types

- **int**
- **float**
- **complex**
- **bool**
- **str**

2. In Python,we can represent char values also by using str type and explicitly char type is not available.

Eg:

```
>>> c='a'
>>> type(c)
<class 'str'>
```

3. long Data Type is available in Python2 but not in Python3. In Python3 long values also we can represent by using int type only.

4. In Python we can present char Value also by using str Type and explicitly char Type is not available.

Type Casting

We can convert one type value to another type. This conversion is called Typecasting or Type coersion.
The following are various inbuilt functions for type casting.

1. int()
2. float()
3. complex()
4. bool()
5. str()

1.int():

We can use this function to convert values from other types to int

Eg:

```
1)  >>> int(123.987)
2)  123
3)  >>> int(10+5j)
4)  TypeError: can't convert complex to int
5)  >>> int(True)
6)  1
7)  >>> int(False)
8)  0
9)  >>> int("10")
10) 10
11) >>> int("10.5")
12) ValueError: invalid literal for int() with base 10: '10.5'
13) >>> int("ten")
14) ValueError: invalid literal for int() with base 10: 'ten'
15) >>> int("0B1111")
16) ValueError: invalid literal for int() with base 10: '0B1111'
```

Note:

1. We can convert from any type to int except complex type.
2. If we want to convert str type to int type, compulsary str should contain only integral value and should be specified in base-10

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

2. float():

We can use float() function to convert other type values to float type.

```
1)  >>> float(10)
2)  10.0
3)  >>> float(10+5j)
4)  TypeError: can't convert complex to float
5)  >>> float(True)
6)  1.0
7)  >>> float(False)
8)  0.0
9)  >>> float("10")
10) 10.0
11) >>> float("10.5")
12) 10.5
13) >>> float("ten")
14) ValueError: could not convert string to float: 'ten'
15) >>> float("0B1111")
16) ValueError: could not convert string to float: '0B1111'
```

Note:

1. We can convert any type value to float type except complex type.

2. Whenever we are trying to convert str type to float type compulsary str should be either integral or floating point literal and should be specified only in base-10.

3.complex():

We can use complex() function to convert other types to complex type.

Form-1: complex(x)

We can use this function to convert x into complex number with real part x and imaginary part 0.

Eg:

```
1)  complex(10)==>10+0j
2)  complex(10.5)===>10.5+0j
3)  complex(True)==>1+0j
4)  complex(False)==>0j
5)  complex("10")==>10+0j
6)  complex("10.5")==>10.5+0j
7)  complex("ten")
8)      ValueError: complex() arg is a malformed string
```

Form-2: complex(x,y)

We can use this method to convert x and y into complex number such that x will be real part and y will be imaginary part.

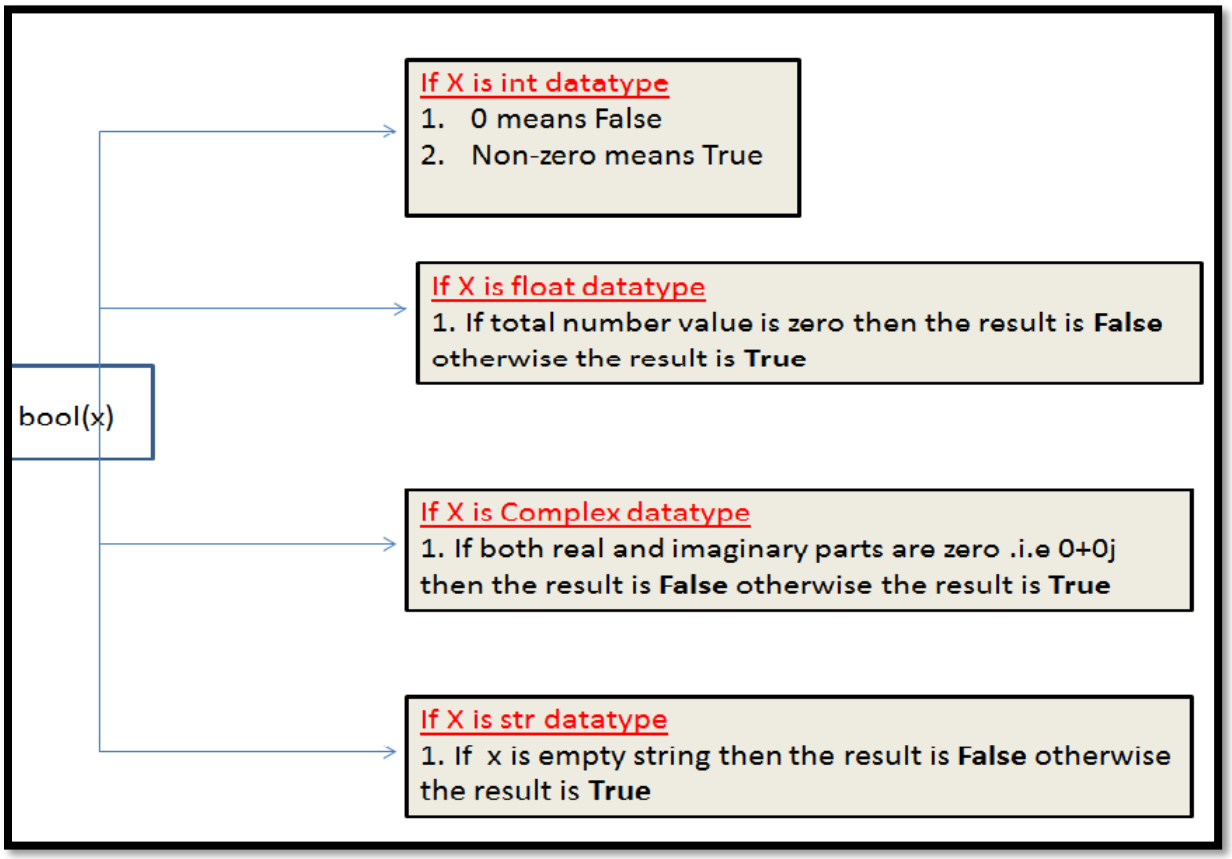
Eg: complex(10,-2)==>10-2j
complex(True,False)==>1+0j

4. bool():

We can use this function to convert other type values to bool type.

Eg:

```
1) bool(0)==>False
2) bool(1)==>True
3) bool(10)===>True
4) bool(10.5)===>True
5) bool(0.178)==>True
6) bool(0.0)==>False
7) bool(10-2j)==>True
8) bool(0+1.5j)==>True
9) bool(0+0j)==>False
10) bool("True")==>True
11) bool("False")==>True
12) bool("")==>False
```



5. str():

We can use this method to convert other type values to str type

Eg:

```
1) >>> str(10)
2) '10'
3) >>> str(10.5)
4) '10.5'
5) >>> str(10+5j)
6) '(10+5j)'
7) >>> str(True)
8) 'True'
```

Fundamental Data Types vs Immutability:

All Fundamental Data types are immutable. i.e once we creates an object,we cannot perform any changes in that object. If we are trying to change then with those changes a new object will be created. This non-chageable behaviour is called immutability.

In Python if a new object is required, then PVM wont create object immediately. First it will check is any object available with the required content or not. If available then existing object will be reused. If it is not available then only a new object will be created. The advantage of this approach is memory utilization and performance will be improved.

But the problem in this approach is,several references pointing to the same object,by using one reference if we are allowed to change the content in the existing object then the remaining references will be effected. To prevent this immutability concept is required. According to this once creates an object we are not allowed to change content. If we are trying to change with those changes a new object will be created.

Eg:

```
1)  >>> a=10
2)  >>> b=10
3)  >>> a is b
4)  True
5)  >>> id(a)
6)  1572353952
7)  >>> id(b)
8)  1572353952
9)  >>>
```

```
>>> a=10
>>> b=10
>>> id(a)
1572353952
>>> id(b)
1572353952
>>> a is b
True
```

```
>>> a=10+5j
>>> b=10+5j
>>> a is b
False
>>> id(a)
15980256
>>> id(b)
15979944
```

```
>>> a=True
>>> b=True
>>> a is b
True
>>> id(a)
1572172624
>>> id(b)
1572172624
```

```
>>> a='durga'
>>> b='durga'
>>> a is b
True
>>> id(a)
16378848
>>> id(b)
16378848
```

bytes Data Type:

bytes data type represens a group of byte numbers just like an array.

Eg:

```
1)  x = [10,20,30,40]
2)      b = bytes(x)
3)      type(b)==>bytes
4)      print(b[0])==> 10
5)      print(b[-1])==> 40
6)      >>> for i in b : print(i)
7)
8)        10
9)        20
10)       30
11)       40
```

Conclusion 1:

The only allowed values for byte data type are 0 to 256. By mistake if we are trying to provide any other values then we will get value error.

Conclusion 2:

Once we creates bytes data type value, we cannot change its values,otherwise we will get TypeError.

Eg:

```
1)  >>> x=[10,20,30,40]
2)  >>> b=bytes(x)
3)  >>> b[0]=100
4)   TypeError: 'bytes' object does not support item assignment
```

bytearray Data type:

bytearray is exactly same as bytes data type except that its elements can be modified.

Eg 1:

```
1)  x=[10,20,30,40]
2)  b = bytearray(x)
3)  for i in b : print(i)
4)  10
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

```
5)  20
6)  30
7)  40
8)   b[0]=100
9)   for i in b: print(i)
10) 100
11) 20
12) 30
13) 40
```

Eg 2:

```
1)  >>> x =[10,256]
2)  >>> b = bytearray(x)
3)  ValueError: byte must be in range(0, 256)
```

list data type:

If we want to represent a group of values as a single entity where insertion order required to preserve and duplicates are allowed then we should go for list data type.

1. insertion order is preserved
2. heterogeneous objects are allowed
3. duplicates are allowed
4. Growable in nature
5. values should be enclosed within square brackets.

Eg:

```
1)  list=[10,10.5,'durga',True,10]
2)      print(list) # [10,10.5,'durga',True,10]
```

Eg:

```
1)  list=[10,20,30,40]
2)  >>> list[0]
3)  10
4)  >>> list[-1]
5)  40
6)  >>> list[1:3]
7)  [20, 30]
8)  >>> list[0]=100
9)  >>> for i in list:print(i)
10) ...
11) 100
12) 20
13) 30
```

```
14) 40
```

list is growable in nature. i.e based on our requirement we can increase or decrease the size.

```
1)  >>> list=[10,20,30]
2)  >>> list.append("durga")
3)  >>> list
4)  [10, 20, 30, 'durga']
5)  >>> list.remove(20)
6)  >>> list
7)  [10, 30, 'durga']
8)  >>> list2=list*2
9)  >>> list2
10) [10, 30, 'durga', 10, 30, 'durga']
```

Note: An ordered, mutable, heterogenous collection of eleemnts is nothing but list, where duplicates also allowed.

tuple data type:

tuple data type is exactly same as list data type except that it is immutable.i.e we cannot chage values.

Tuple elements can be represented within parenthesis.

Eg:

```
1)  t=(10,20,30,40)
2)  type(t)
3)  <class 'tuple'>
4)  t[0]=100
5)  TypeError: 'tuple' object does not support item assignment
6)  >>> t.append("durga")
7)  AttributeError: 'tuple' object has no attribute 'append'
8)  >>> t.remove(10)
9)  AttributeError: 'tuple' object has no attribute 'remove'
```

Note: tuple is the read only version of list

range Data Type:

range Data Type represents a sequence of numbers.
The elements present in range Data type are not modifiable. i.e range Data type is immutable.

Form-1: range(10)
generate numbers from 0 to 9

Eg:
```
r=range(10)
for i in r : print(i)     0 to 9
```

Form-2: range(10,20)

generate numbers from 10 to 19

```
r = range(10,20)
for i in r : print(i)     10 to 19
```

Form-3: range(10,20,2)

2 means increment value

```
r = range(10,20,2)
for i in r : print(i)    10,12,14,16,18
```

We can access elements present in the range Data Type by using index.
```
r=range(10,20)
r[0]==>10
r[15]==>IndexError: range object index out of range
```

We cannot modify the values of range data type

Eg:
```
r[0]=100
TypeError: 'range' object does not support item assignment
```

We can create a list of values with range data type

Eg:

```
1)  >>> l = list(range(10))
2)  >>> l
3)  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

set Data Type:

If we want to represent a group of values without duplicates where order is not important then we should go for set Data Type.

1. insertion order is not preserved
2. duplicates are not allowed
3. heterogeneous objects are allowed
4. index concept is not applicable
5. It is mutable collection
6. Growable in nature

Eg:

```
1)  s={100,0,10,200,10,'durga'}
2)  s # {0, 100, 'durga', 200, 10}
3)  s[0] ==>TypeError: 'set' object does not support indexing
4)
5)  set is growable in nature, based on our requirement we can increase or decrease the size.
6)
7)  >>> s.add(60)
8)  >>> s
9)  {0, 100, 'durga', 200, 10, 60}
10) >>> s.remove(100)
11) >>> s
12) {0, 'durga', 200, 10, 60}
```

frozenset Data Type:

It is exactly same as set except that it is immutable.
Hence we cannot use add or remove functions.

```
1)  >>> s={10,20,30,40}
2)  >>> fs=frozenset(s)
3)  >>> type(fs)
4)  <class 'frozenset'>
5)  >>> fs
6)  frozenset({40, 10, 20, 30})
7)  >>> for i in fs:print(i)
8)  ...
9)  40
10) 10
11) 20
12) 30
13)
14) >>> fs.add(70)
15) AttributeError: 'frozenset' object has no attribute 'add'
16) >>> fs.remove(10)
17) AttributeError: 'frozenset' object has no attribute 'remove'
```

dict Data Type:

If we want to represent a group of values as key-value pairs then we should go for dict data type.

Eg:

```
d={101:'durga',102:'ravi',103:'shiva'}
```

Duplicate keys are not allowed but values can be duplicated. If we are trying to insert an entry with duplicate key then old value will be replaced with new value.

Eg:

```
>>> d={101:'durga',102:'ravi',103:'shiva'}
>>> d[101]='sunny'
>>> d
{101: 'sunny', 102: 'ravi', 103: 'shiva'}

We can create empty dictionary as follows
d={ }
We can add key-value pairs as follows
d['a']='apple'
d['b']='banana'
print(d)
```

Note: dict is mutable and the order wont be preserved.

Note:

1. In general we can use bytes and bytearray data types to represent binary information like images,video files etc
2. In Python2 long data type is available. But in Python3 it is not available and we can represent long values also by using int type only.
3. In Python there is no char data type. Hence we can represent char values also by using str type.

Summary of Datatypes in Python3

Datatype	Description	Is Immutable	Example

Int	We can use to represent the whole/integral numbers	Immutable	>>> a=10
			>>> type(a)
			<class 'int'>

Float	We can use to represent the decimal/floating point numbers	Immutable	>>> b=10.5
			>>> type(b)
			<class 'float'>

Complex	We can use to represent the complex numbers	Immutable	>>> c=10+5j
			>>> type(c)
			<class 'complex'>
			>>> c.real
			10.0
			>>> c.imag
			5.0

Bool	We can use to represent the logical values(Only allowed values are True and False)	Immutable	>>> flag=True
			>>> flag=False
			>>> type(flag)
			<class 'bool'>

Str	To represent sequence of Characters	Immutable	>>> s='durga'
			>>> type(s)
			<class 'str'>
			>>> s="durga"
			>>> s='''Durga Software Solutions
			... Ameerpet'''
			>>> type(s)
			<class 'str'>

bytes	To represent a sequence of byte values from 0-255	Immutable	>>> list=[1,2,3,4]
			>>> b=bytes(list)
			>>> type(b)
			<class 'bytes'>

bytearray	To represent a sequence of byte values from 0-255	Mutable	>>> list=[10,20,30]
			>>> ba=bytearray(list)
			>>> type(ba)
			<class 'bytearray'>

range	To represent a range of values	Immutable	>>> r=range(10)
			>>> r1=range(0,10)
			>>> r2=range(0,10,2)

list	To represent an ordered collection of objects	Mutable	>>> l=[10,11,12,13,14,15]
			>>> type(l)
			<class 'list'>

tuple	To represent an ordered collections of objects	Immutable	>>> t=(1,2,3,4,5)
			>>> type(t)
			<class 'tuple'>

set	To represent an unordered collection of unique objects	Mutable	>>> s={1,2,3,4,5,6} >>> type(s)

			<class 'set'>

frozenset	To represent an unordered collection of unique objects	Immutable	>>> s={11,2,3,'Durga',100,'Ramu'}
			>>> fs=frozenset(s)
			>>> type(fs)
			<class 'frozenset'>

dict	To represent a group of key value pairs	Mutable	>>> d={101:'durga',102:'ramu',103:'hari'} >>> type(d) <class 'dict'>

None Data Type:

None means Nothing or No value associated.
If the value is not available,then to handle such type of cases None introduced.
It is something like null value in Java.

Eg:

```
def  m1():
  a=10

print(m1())
  None
```

Escape Characters:

In String literals we can use esacpe characters to associate a special meaning.

```
1)  >>> s="durga\nsoftware"
2)  >>> print(s)
3)  durga
4)  software
5)  >>> s="durga\tsoftware"
6)  >>> print(s)
7)  durga   software
8)  >>> s="This is " symbol"
9)    File "<stdin>", line 1
10)    s="This is " symbol"
11)                 ^
12) SyntaxError: invalid syntax
13) >>> s="This is \" symbol"
14) >>> print(s)
15) This is " symbol
```

The following are various important escape characters in Python

1) **\n==>New Line**
2) **\t===>Horizontal tab**
3) **\r ==>Carriage Return**
4) **\b===>Back space**
5) **\f===>Form Feed**
6) **\v==>Vertical tab**
7) **\'===>Single quote**
8) **\"===>Double quote**
9) **\\===>back slash symbol**

....

Constants:

Constants concept is not applicable in Python.
But it is convention to use only uppercase characters if we don't want to change value.

MAX_VALUE=10

It is just convention but we can change the value.

Operators

Operator is a symbol that performs certain operations.
Python provides the following set of operators

1. **Arithmetic Operators**
2. **Relational Operators or Comparison Operators**
3. **Logical operators**
4. **Bitwise oeprators**
5. **Assignment operators**
6. **Special operators**

1. Arithmetic Operators:

+ ==>Addition
- ==>Subtraction
*** ==>Multiplication**
/ ==>Division operator
% ===>Modulo operator

// ==>Floor Division operator

**** ==>Exponent operator or power operator**

Eg: test.py:

```
1) a=10
2) b=2
3) print('a+b=',a+b)
4) print('a-b=',a-b)
5) print('a*b=',a*b)
6) print('a/b=',a/b)
7) print('a//b=',a//b)
8) print('a%b=',a%b)
9) print('a**b=',a**b)
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Output:

```
1) Python test.py   or py test.py
2) a+b= 12
3) a-b= 8
4) a*b= 20
5) a/b= 5.0
6) a//b= 5
7) a%b= 0
8) a**b= 100
```

Eg:

```
1) a = 10.5
2) b=2
3)
4) a+b= 12.5
5) a-b= 8.5
6) a*b= 21.0
7) a/b= 5.25
8) a//b= 5.0
9) a%b= 0.5
10) a**b= 110.25
```

Eg:

10/2==>5.0
10//2==>5
10.0/2===>5.0
10.0//2===>5.0

Note: / operator always performs floating point arithmetic. Hence it will always returns float value.

But Floor division (//) can perform both floating point and integral arithmetic. If arguments are int type then result is int type. If atleast one argument is float type then result is float type.

Note:

We can use +,* operators for str type also.
If we want to use + operator for str type then compulsory both arguments should be str type only otherwise we will get error.

```
1) >>> "durga"+10
2) TypeError: must be str, not int
3) >>> "durga"+"10"
4) 'durga10'
```

If we use * operator for str type then compulsory one argument should be int and other argument should be str type.

2*"durga"
"durga"*2
2.5*"durga" ==>TypeError: can't multiply sequence by non-int of type 'float'
"durga"*"durga"==>TypeError: can't multiply sequence by non-int of type 'str'

+====>String concatenation operator
*** ===>String multiplication operator**

Note: For any number x,

x/0 and x%0 always raises "ZeroDivisionError"

10/0
10.0/0
.....

Relational Operators:

>,>=,<,<=

Eg 1:

```
1)  a=10
2)  b=20
3)  print("a > b is ",a>b)
4)  print("a >= b is ",a>=b)
5)  print("a < b is ",a<b)
6)  print("a <= b is ",a<=b)
7)
8)  a > b is  False
9)  a >= b is  False
10) a < b is  True
11) a <= b is  True
```

We can apply relational operators for str types also

Eg 2:

```
1)  a="durga"
2)  b="durga"
3)  print("a > b is ",a>b)
4)  print("a >= b is ",a>=b)
5)  print("a < b is ",a<b)
```

```
6)  print("a <= b is ",a<=b)
7)
8)  a > b is  False
9)  a >= b is  True
10) a < b is  False
11) a <= b is  True
```

Eg:

```
1)  print(True>True)   False
2)  print(True>=True)   True
3)  print(10 >True) True
4)  print(False > True) False
5)
6)  print(10>'durga')
7)     TypeError: '>' not supported between instances of 'int' and 'str'
```

Eg:

```
1)  a=10
2)  b=20
3)  if(a>b):
4)     print("a is greater than b")
5)  else:
6)     print("a is not greater than b")
```

Outputa is not greater than b

Note: **Chaining of relational operators is possible. In the chaining, if all comparisons returns True then only result is True. If atleast one comparison returns False then the result is False**

Eg:

```
1)  10<20 ==>True
2)  10<20<30  ==>True
3)  10<20<30<40  ==>True
4)  10<20<30<40>50 ==>False
```

equality operators:

== , !=

We can apply these operators for any type even for incompatible types also

```
1)  >>> 10==20
2)  False
3)  >>> 10!= 20
```

```
4)  True
5)  >>> 10==True
6)  False
7)  >>> False==False
8)  True
9)  >>> "durga"=="durga"
10) True
11) >>> 10=="durga"
12) False
```

Note: Chaining concept is applicable for equality operators. If atleast one comparison returns False then the result is False. otherwise the result is True.

Eg:

```
1)  >>> 10==20==30==40
2)  False
3)  >>> 10==10==10==10
4)  True
```

Logical Operators:

and, or ,not

We can apply for all types.

For boolean types behaviour:

and ==>If both arguments are True then only result is True
or ====>If atleast one arugemnt is True then result is True
not ==>complement

True and False ==>False
True or False ===>True
not False ==>True

For non-boolean types behaviour:

0 means False
non-zero means True
empty string is always treated as False

x and y:

==>if x is evaluates to false return x otherwise return y

Eg:
10 and 20
0 and 20

If first argument is zero then result is zero otherwise result is y

x or y:

If x evaluates to True then result is x otherwise result is y

10 or 20 ==> 10
0 or 20 ==> 20

not x:

If x is evalutates to False then result is True otherwise False

not 10 ==>False

not 0 ==>True

Eg:

```
1) "durga" and "durgasoft" ==>durgasoft
2) "" and "durga" ==>""
3) "durga" and "" ==>""
4) "" or "durga" ==>"durga"
5) "durga" or ""==>"durga"
6) not ""==>True
7) not "durga" ==>False
```

Bitwise Operators:

We can apply these operators bitwise.
These operators are applicable only for int and boolean types.
By mistake if we are trying to apply for any other type then we will get Error.

&,|,^,~,<<,>>

print(4&5) ==>valid
print(10.5 & 5.6) ==>
TypeError: unsupported operand type(s) for &: 'float' and 'float'

print(True & True) ==>valid

& ==> If both bits are 1 then only result is 1 otherwise result is 0
| ==> If atleast one bit is 1 then result is 1 otherwise result is 0
^ ==>If bits are different then only result is 1 otherwise result is 0
~ ==>bitwise complement operator
1==>0 & 0==>1
<< ==>Bitwise Left shift
>> ==>Bitwise Right Shift

print(4&5) ==>4
print(4|5) ==>5
print(4^5) ==>1

Operator	Description
&	If both bits are 1 then only result is 1 otherwise result is 0
|	If atleast one bit is 1 then result is 1 otherwise result is 0
^	If bits are different then only result is 1 otherwise result is 0
~	bitwise complement operator i.e 1 means 0 and 0 means 1
>>	Bitwise Left shift Operator
<<	Bitwise Right shift Operator

bitwise complement operator(~):

We have to apply complement for total bits.

Eg: print(~5) ==>-6

Note:

The most significant bit acts as sign bit. 0 value represents +ve number where as 1 represents -ve value.
positive numbers will be repesented directly in the memory where as -ve numbers will be represented indirectly in 2's complement form.

Shift Operators:

<< Left shift operator

After shifting the empty cells we have to fill with zero

print(10<<2)==>40

~~0~~	~~0~~	0	0	1	0	1	0

0	0	1	0	1	0	0	0

>> Right Shift operator

After shifting the empty cells we have to fill with sign bit.(0 for +ve and 1 for -ve)

print(10>>2) ==>2

0	0	0	0	1	0	~~1~~	~~0~~

0	0	0	0	0	0	1	0

We can apply bitwise operators for boolean types also

```
print(True & False)   ==>False
print(True | False)   ===>True
print(True ^ False)  ==>True
print(~True)            ==>-2
print(True<<2)          ==>4
print(True>>2)          ==>0
```

Assignment Operators:

We can use assignment operator to assign value to the variable.

Eg:
x=10

We can combine asignment operator with some other operator to form compound assignment operator.

Eg: x+=10 ====> x = x+10

The following is the list of all possible compound assignment operators in Python

```
+=
-=
*=
/=
%=
//=
**=
&=
|=
^=
```

>>=
<<=

Eg:

```
1) x=10
2) x+=20
3) print(x) ==>30
```

Eg:

```
1) x=10
2) x&=5
3) print(x) ==>0
```

Special operators:

Python defines the following 2 special operators

1. Identity Operators
2. Membership operators

1. Identity Operators

We can use identity operators for address comparison .
2 identity operators are available
1. is
2. is not

r1 is r2 returns True if both r1 and r2 are pointing to the same object
r1 is not r2 returns True if both r1 and r2 are not pointing to the same object

Eg:

```
1)  a=10
2) b=10
3) print(a is b)      True
4) x=True
5) y=True
6) print( x is y)     True
```

Eg:

```
1) a="durga"
2) b="durga"
3) print(id(a))
4) print(id(b))
5) print(a is b)
```

Eg:

```
1) list1=["one","two","three"]
2) list2=["one","two","three"]
3) print(id(list1))
4) print(id(list2))
5) print(list1 is list2)   False
6) print(list1 is not list2)   True
7) print(list1 == list2)    True
```

Note:

We can use is operator for address comparison where as == operator for content comparison.

2. Membership operators:

We can use Membership operators to check whether the given object present in the given collection.(It may be String,List,Set,Tuple or Dict)

in ==>Returns True if the given object present in the specified Collection
not in ==>Retruns True if the given object not present in the specified Collection

Eg:

```
1) x="hello learning Python is very easy!!!"
2) print('h' in x)    True
3) print('d' in x)     False
4) print('d' not in x)     True
5) print('Python' in x)    True
```

Eg:

```
1) list1=["sunny","bunny","chinny","pinny"]
2) print("sunny" in list1)  True
3) print("tunny" in list1)  False
4) print("tunny" not in list1)   True
```

Operator Precedence:

If multiple operators present then which operator will be evaluated first is decided by operator precedence.

Eg:

```
print(3+10*2) ==>23
print((3+10)*2) ==>26
```

The following list describes operator precedence in Python

() ==>Parenthesis
**** ==>exponential operator**
~,- ==>Bitwise complement operator,unary minus operator
***,/,%,// ==>multiplication,division,modulo,floor division**
+,- ==>addition,subtraction
<<,>> ==>Left and Right Shift
&==>bitwise And
^===>Bitwise X-OR
| ==>Bitwise OR
>,>=,<,<=, ==, != ==>Relational or Comparison operators
=,+=,-=,*=... ==>Assignment operators
is , is not ==>Identity Operators
in , not in ==>Membership operators
not ==>Logical not
and ==>Logical and
or ==>Logical or

Eg:

```
1)  a=30
2)  b=20
3)  c=10
4)  d=5
5)  print((a+b)*c/d)      100.0
6)  print((a+b)*(c/d))    100.0
7)  print(a+(b*c)/d)      70.0
8)
9)
10) 3/2*4+3+(10/5)**3-2
11) 3/2*4+3+2.0**3-2
12) 3/2*4+3+8.0-2
13) 1.5*4+3+8.0-2
14) 6.0+3+8.0-2
15) 15.0
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Mathematical Functions (math Module)

A Module is collection of functions ,variables and classes etc.

math is a module that contains several functions to perform mathematical operations

If we want to use any module in Python, first we have to import that module.

```
import math
```

Once we import a module then we can call any function of that module.

```
import math
print(math.sqrt(16))
print(math.pi)
```

```
4.0
3.141592653589793
```

We can create alias name by using as keyword.

```
import math as m
```

Once we create alias name , by using that we can access functions and variables of that module

```
import math   as m
print(m.sqrt(16))
print(m.pi)
```

We can import a particular member of a module explicitly as follows

```
from math import sqrt
from math import sqrt,pi
```

If we import a member explicitly then it is not required to use module name while accessing.

```
from math import sqrt,pi
print(sqrt(16))
print(pi)
print(math.pi)    ==>NameError: name 'math' is not defined
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

important functions of math module:

ceil(x)
floor(x)
pow(x,y)
factorial(x)
trunc(x)
gcd(x,y)
sin(x)
cos(x)
tan(x)
....

important variables of math module:

pi3.14
e===>2.71
inf ==>infinity
nan ==>not a number

Q. Write a Python program to find area of circle

pi*r2**

```
from math import pi
r=16
print("Area of Circle is :",pi*r**2)
```

OutputArea of Circle is : 804.247719318987

Input And Output Statements

Reading dynamic input from the keyboard:

In Python 2 the following 2 functions are available to read dynamic input from the keyboard.

1. raw_input()
2. input()

1. raw_input():

This function always reads the data from the keyboard in the form of String Format. We have to convert that string type to our required type by using the corresponding type casting methods.

Eg:

```
x=raw_input("Enter First Number:")
print(type(x))    It will always print  str type only for any input type
```

2. input():

input() function can be used to read data directly in our required format.We are not required to perform type casting.

```
x=input("Enter Value)
type(x)
```

```
10 ===> int
"durga"===>str
10.5===>float
True==>bool
```

***Note: But in Python 3 we have only input() method and raw_input() method is not available.

Python3 input() function behaviour exactly same as raw_input() method of Python2. i.e every input value is treated as str type only.

raw_input() function of Python 2 is renamed as input() function in Python3

Eg:

```
>>> type(input("Enter value:"))
Enter value:10
<class 'str'>

Enter value:10.5
<class 'str'>

Enter value:True
<class 'str'>
```

Q. Write a program to read 2 numbers from the keyboard and print sum.

```
x=input("Enter First Number:")
y=input("Enter Second Number:")
i = int(x)
j = int(y)
print("The Sum:",i+j)

Enter First Number:100
Enter Second Number:200
The Sum: 300
```

```
x=int(input("Enter First Number:"))
y=int(input("Enter Second Number:"))
print("The Sum:",x+y)
```

```
print("The Sum:",int(input("Enter First Number:"))+int(input("Enter Second Number:")))
```

Q. Write a program to read Employee data from the keyboard and print that data.

```
eno=int(input("Enter Employee No:"))
ename=input("Enter Employee Name:")
esal=float(input("Enter Employee Salary:"))
eaddr=input("Enter Employee Address:")
married=bool(input("Employee Married ?[True|False]:"))
print("Please Confirm Information")
print("Employee No :",eno)
print("Employee Name :",ename)
print("Employee Salary :",esal)
print("Employee Address :",eaddr)
print("Employee Married ? :",married)

```

```
D:\Python_classes>py test.py
Enter Employee No:100
Enter Employee Name:Sunny
Enter Employee Salary:1000
Enter Employee Address:Mumbai
Employee Married ?[True|False]:True
Please Confirm Information
Employee No : 100
Employee Name : Sunny
Employee Salary : 1000.0
Employee Address : Mumbai
Employee Married ? : True
```

How to read multiple values from the keyboard in a single line:

```
a,b= [int(x) for x in input("Enter 2 numbers :").split()]
print("Product is :", a*b)

D:\Python_classes>py test.py
Enter 2 numbers :10 20
Product is : 200
```

Note: split() function can take space as seperator by default .But we can pass anything as seperator.

Q. Write a program to read 3 float numbers from the keyboard with , seperator and print their sum.

```
a,b,c= [float(x) for x in input("Enter 3 float numbers :").split(',')]
print("The Sum is :", a+b+c)

D:\Python_classes>py test.py
Enter 3 float numbers :10.5,20.6,20.1
The Sum is : 51.2
```

eval():

eval Function take a String and evaluate the Result.

Eg: x = eval("10+20+30")
print(x)

Output: 60

Eg: x = eval(input("Enter Expression"))
Enter Expression: 10+2*3/4
Output11.5

eval() can evaluate the Input to list, tuple, set, etc based the provided Input.

Eg: Write a Program to accept list from the keynboard on the display

```
1) l = eval(input("Enter List"))
2) print (type(l))
3) print(l)
```

Command Line Arguments

- argv is not Array it is a List. It is available sys Module.
- The Argument which are passing at the time of execution are called Command Line Arguments.

Eg: D:\Python_classes py test.py 10 20 30

Command Line Arguments

Within the Python Program this Command Line Arguments are available in argv. Which is present in SYS Module.

test.py	10	20	30

Note: argv[0] represents Name of Program. But not first Command Line Argument. argv[1] represent First Command Line Argument.

Program: To check type of argv from sys

```
import argv
print(type(argv))
```

D:\Python_classes\py test.py

Write a Program to display Command Line Arguments

```
1) from sys import argv
2) print("The Number of Command Line Arguments:", len(argv))
3) print("The List of Command Line Arguments:", argv)
4) print("Command Line Arguments one by one:")
5) for x in argv:
6)     print(x)
7)
8) D:\Python_classes>py test.py 10 20 30
9) The Number of Command Line Arguments: 4
```

```
10) The List of Command Line Arguments: ['test.py', '10','20','30']
11) Command Line Arguments one by one:
12) test.py
13) 10
14) 20
15) 30
```

```
1)  from sys import argv
2)  sum=0
3)  args=argv[1:]
4)  for x in args   :
5)    n=int(x)
6)    sum=sum+n
7)  print("The Sum:",sum)
8)
9)  D:\Python_classes>py test.py 10 20 30 40
10) The Sum: 100
```

Note1: usually space is seperator between command line arguments. If our command line argument itself contains space then we should enclose within double quotes(but not single quotes)

Eg:

```
1)  from sys import argv
2)  print(argv[1])
3)
4)  D:\Python_classes>py test.py Sunny Leone
5)  Sunny
6)
7)  D:\Python_classes>py test.py 'Sunny Leone'
8)  'Sunny
9)
10) D:\Python_classes>py test.py "Sunny Leone"
11) Sunny Leone
```

Note2: Within the Python program command line arguments are available in the String form. Based on our requirement,we can convert into corresponding type by using type casting methods.

Eg:

```
1)  from sys import argv
2)  print(argv[1]+argv[2])
3)  print(int(argv[1])+int(argv[2]))
```

```
4)
5)  D:\Python_classes>py test.py 10 20
6)  1020
7)  30
```

Note3: If we are trying to access command line arguments with out of range index then we will get Error.

Eg:

```
1)  from sys import argv
2)  print(argv[100])
3)
4)  D:\Python_classes>py test.py 10 20
5)  IndexError: list index out of range
```

Note:

In Python there is argparse module to parse command line arguments and display some help messages whenever end user enters wrong input.

input()
raw_input()

command line arguments

output statements:

We can use print() function to display output.

Form-1: print() without any argument
Just it prints new line character

Form-2:

```
1)  print(String):
2)  print("Hello World")
3)  We can use escape characters also
4)  print("Hello \n World")
5)  print("Hello\tWorld")
6)  We can use repetetion operator (*) in the string
7)  print(10*"Hello")
8)  print("Hello"*10)
9)  We can use + operator also
10) print("Hello"+"World")
```

Note:

If both arguments are String type then + operator acts as concatenation operator.

If one argument is string type and second is any other type like int then we will get Error

If both arguments are number type then + operator acts as arithmetic addition operator.

Note:

```
1) print("Hello"+"World")
2) print("Hello","World")
3)
4) HelloWorld
5) Hello World
```

Form-3: print() with variable number of arguments:

```
1. a,b,c=10,20,30
2. print("The Values are :",a,b,c)
3.
4. OutputThe Values are : 10 20 30
```

By default output values are seperated by space.If we want we can specify seperator by using "sep" attribute

```
1. a,b,c=10,20,30
2. print(a,b,c,sep=',')
3. print(a,b,c,sep=':')
4.
5. D:\Python_classes>py test.py
6. 10,20,30
7. 10:20:30
```

Form-4:print() with end attribute:

```
1. print("Hello")
2. print("Durga")
3. print("Soft")
```

Output:

```
1. Hello
2. Durga
3. Soft
```

If we want output in the same line with space

```
1. print("Hello",end=' ')
2. print("Durga",end=' ')
3. print("Soft")
```

Output: Hello Durga Soft

Note: **The default value for end attribute is \n,which is nothing but new line character.**

Form-5: print(object) statement:

We can pass any object (like list,tuple,set etc)as argument to the print() statement.

Eg:

```
1. l=[10,20,30,40]
2. t=(10,20,30,40)
3. print(l)
4. print(t)
```

Form-6: print(String,variable list):

We can use print() statement with String and any number of arguments.

Eg:

```
1. s="Durga"
2. a=48
3. s1="java"
4. s2="Python"
5. print("Hello",s,"Your Age is",a)
6. print("You are teaching",s1,"and",s2)
```

Output:

```
1) Hello Durga Your Age is 48
2) You are teaching java and Python
```

Form-7: print(formatted string):

%i====>int
%d====>int
%f=====>float
%s======>String type

Syntax:

print("formatted string" %(variable list))

Eg 1:

```
1) a=10
2) b=20
3) c=30
4) print("a value is %i" %a)
5) print("b  value is %d and c value is %d" %(b,c))
6)
7) Output
8) a value is 10
9) b  value is 20 and c value is 30
```

Eg 2:

```
1) s="Durga"
2) list=[10,20,30,40]
3) print("Hello %s ...The List of Items are %s"  %(s,list))
4)
5) Output Hello Durga ...The List of Items are [10, 20, 30, 40]
```

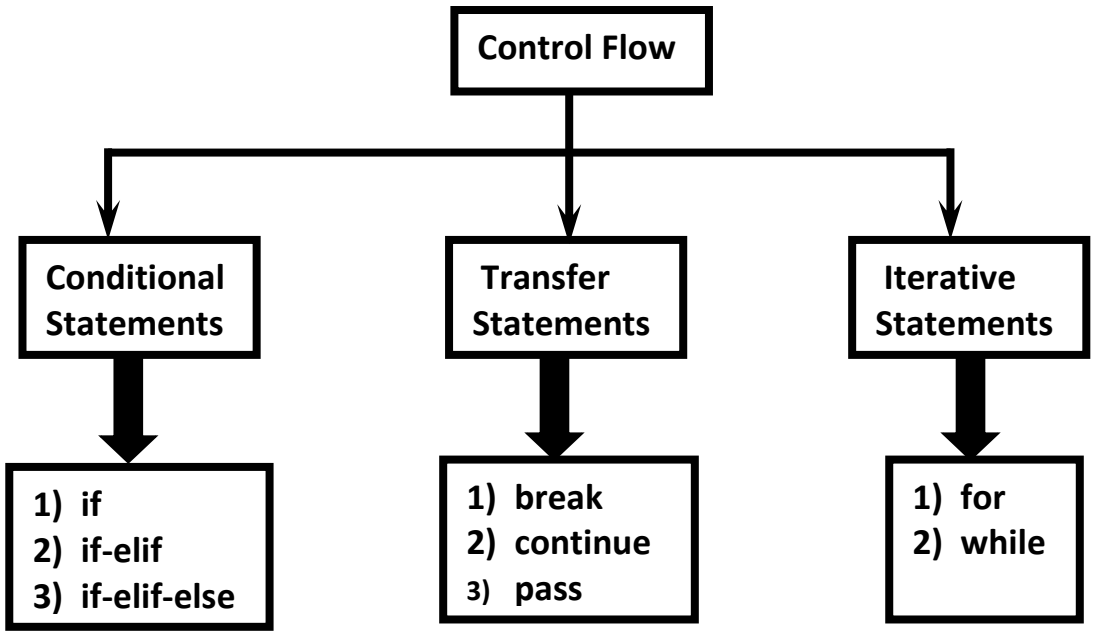
Form-8: print() with replacement operator {}

Eg:

```
1) name="Durga"
2) salary=10000
3) gf="Sunny"
4) print("Hello {0} your salary is {1} and Your Friend {2} is waiting".format(name,salary,gf))
5) print("Hello {x} your salary is {y} and Your Friend {z} is waiting".format(x=name,y=salary,z=
   gf))
6)
7) Output
8) Hello Durga your salary is 10000 and Your Friend Sunny is waiting
9) Hello Durga your salary is 10000 and Your Friend Sunny is waiting
```

Flow Control

Flow control describes the order in which statements will be executed at runtime.



I. Conditional Statements

1) if

if condition : statement

or

if condition :
statement-1
statement-2
statement-3
If condition is true then statements will be executed.

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Eg:

```
1)  name=input("Enter Name:")
2)  if name=="durga" :
3)      print("Hello Durga Good Morning")
4)  print("How are you!!!")
5)
6)  D:\Python_classes>py test.py
7)  Enter Name:durga
8)  Hello Durga Good Morning
9)  How are you!!!
10)
11) D:\Python_classes>py test.py
12) Enter Name:Ravi
13) How are you!!!
```

2) if-else:

if condition :
Action-1
else :
Action-2

if condition is true then Action-1 will be executed otherwise Action-2 will be executed.

Eg:

```
1)  name=input("Enter Name:")
2)  if name=="durga" :
3)      print("Hello Durga Good Morning")
4)  else:
5)     print("Hello Guest Good Moring")
6)  print("How are you!!!")
7)
8)  D:\Python_classes>py test.py
9)  Enter Name:durga
10) Hello Durga Good Morning
11) How are you!!!
12)
13) D:\Python_classes>py test.py
14) Enter Name:Ravi
15) Hello Guest Good Moring
16) How are you!!!
```

3) if-elif-else:

Syntax:

```
if condition1:
   Action-1
elif condition2:
   Action-2
elif condition3:
   Action-3
 elif condition4:
   Action-4
   ...
else:
   Default Action
```

Based condition the corresponding action will be executed.

Eg:

```
1)  brand=input("Enter Your Favourite Brand:")
2)  if brand=="RC" :
3)      print("It is childrens brand")
4)  elif brand=="KF":
5)     print("It is not that much kick")
6)  elif brand=="FO":
7)     print("Buy one get Free One")
8)  else :
9)     print("Other Brands are not recommended")
10)
11)
12) D:\Python_classes>py test.py
13) Enter Your Favourite Brand:RC
14) It is childrens brand
15)
16) D:\Python_classes>py test.py
17) Enter Your Favourite Brand:KF
18) It is not that much kick
19)
20) D:\Python_classes>py test.py
21) Enter Your Favourite Brand:KALYANI
22) Other Brands are not recommended
```

Note:

1. else part is always optional
   Hence the following are various possible syntaxes.
   1. if
   2. if - else
   3. if-elif-else
   4.if-elif

2. There is no switch statement in Python

Q. Write a program to find biggest of given 2 numbers from the commad prompt?

```
1)  n1=int(input("Enter First Number:"))
2)  n2=int(input("Enter Second Number:"))
3)  if n1>n2:
4)      print("Biggest Number is:",n1)
5)  else :
6)      print("Biggest Number is:",n2)
7)
8)   D:\Python_classes>py test.py
9)  Enter First Number:10
10) Enter Second Number:20
11) Biggest Number is: 20
```

Q. Write a program to find biggest of given 3 numbers from the commad prompt?

```
1)  n1=int(input("Enter First Number:"))
2)  n2=int(input("Enter Second Number:"))
3)  n3=int(input("Enter Third Number:"))
4)  if n1>n2 and n1>n3:
5)      print("Biggest Number is:",n1)
6)  elif n2>n3:
7)      print("Biggest Number is:",n2)
8)  else :
9)      print("Biggest Number is:",n3)
10)
11) D:\Python_classes>py test.py
12) Enter First Number:10
13) Enter Second Number:20
14) Enter Third Number:30
15) Biggest Number is: 30
16)
17) D:\Python_classes>py test.py
18) Enter First Number:10
```

```
Enter Second Number:30
Enter Third Number:20
Biggest Number is: 30
```

Q. Write a program to find smallest of given 2 numbers?
Q. Write a program to find smallest of given 3 numbers?
Q. Write a program to check whether the given number is even or odd?
Q. Write a program to check whether the given number is in between 1 and 100?

```
n=int(input("Enter Number:"))
if n>=1 and n<=10 :
    print("The number",n,"is in between 1 to 10")
else:
    print("The number",n,"is not in between 1 to 10")
```

Q. Write a program to take a single digit number from the key board and print is value in English word?

```
0==>ZERO
1 ==>ONE

n=int(input("Enter a digit from o to 9:"))
if n==0 :
    print("ZERO")
elif n==1:
    print("ONE")
elif n==2:
    print("TWO")
elif n==3:
    print("THREE")
elif n==4:
    print("FOUR")
elif n==5:
    print("FIVE")
elif n==6:
    print("SIX")
elif n==7:
    print("SEVEN")
elif n==8:
    print("EIGHT")
elif n==9:
    print("NINE")
else:
    print("PLEASE ENTER A DIGIT FROM 0 TO 9")
```

II. Iterative Statements

If we want to execute a group of statements multiple times then we should go for Iterative statements.

Python supports 2 types of iterative statements.

1. for loop
2. while loop

1) for loop:

If we want to execute some action for every element present in some sequence(it may be string or collection)then we should go for for loop.

Syntax:

```
for x in sequence :
   body
```

where sequence can be string or any collection.
Body will be executed for every element present in the sequence.

Eg 1: To print characters present in the given string

```
1)  s="Sunny Leone"
2)  for x in s :
3)      print(x)
4)
5)  Output
6)  S
7)  u
8)  n
9)  n
10) y
11)
12) L
13) e
14) o
15) n
16) e
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Eg 2: To print characters present in string index wise:

```
1)  s=input("Enter some String: ")
2)  i=0
3)  for x in s :
4)      print("The character present at ",i,"index is :",x)
5)      i=i+1
6)
7)
8)   D:\Python_classes>py test.py
9)  Enter some String: Sunny Leone
10) The character present at  0 index is : S
11) The character present at  1 index is : u
12) The character present at  2 index is : n
13) The character present at  3 index is : n
14) The character present at  4 index is : y
15) The character present at  5 index is :
16) The character present at  6 index is : L
17) The character present at  7 index is : e
18) The character present at  8 index is : o
19) The character present at  9 index is : n
20) The character present at  10 index is : e
```

Eg 3: To print Hello 10 times

```
1)  for x in range(10) :
2)      print("Hello")
```

Eg 4: To display numbers from 0 to 10

```
1)      for x in range(11) :
2)      print(x)
```

Eg 5: To display odd numbers from 0 to 20

```
1)      for x in range(21) :
2)      if (x%2!=0):
3)          print(x)
```

Eg 6: To display numbers from 10 to 1 in descending order

```
1)          for x in range(10,0,-1) :
2)            print(x)
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Eg 7: To print sum of numbers presenst inside list

```
1)          list=eval(input("Enter List:"))
2)      sum=0;
3)      for x in list:
4)          sum=sum+x;
5)      print("The Sum=",sum)
6)
7)  D:\Python_classes>py test.py
8)  Enter List:[10,20,30,40]
9)  The Sum= 100
10)
11) D:\Python_classes>py test.py
12) Enter List:[45,67]
13) The Sum= 112
```

2) while loop:

If we want to execute a group of statements iteratively until some condition false,then we should go for while loop.

Syntax:

```
while condition :
   body
```

Eg: To print numbers from 1 to 10 by using while loop

```
1)      x=1
2)      while x <=10:
3)      print(x)
4)      x=x+1
```

Eg: To display the sum of first n numbers

```
1)  n=int(input("Enter number:"))
2)  sum=0
3)  i=1
4)  while i<=n:
5)      sum=sum+i
6)      i=i+1
7)  print("The sum of first",n,"numbers is :",sum)
```

Eg: write a program to prompt user to enter some name until entering Durga

```
1) name=""
2) while name!="durga":
3)     name=input("Enter Name:")
4) print("Thanks for confirmation")
```

Infinite Loops:

```
1) i=0;
2) while True :
3)     i=i+1;
4)     print("Hello",i)
```

Nested Loops:

Sometimes we can take a loop inside another loop,which are also known as nested loops.

Eg:

```
1) for i in range(4):
2)     for j in range(4):
3)         print("i=",i,"   j=",j)
4)
5) Output
6) D:\Python_classes>py test.py
7) i= 0   j= 0
8) i= 0   j= 1
9) i= 0   j= 2
10) i= 0   j= 3
11) i= 1   j= 0
12) i= 1   j= 1
13) i= 1   j= 2
14) i= 1   j= 3
15) i= 2   j= 0
16) i= 2   j= 1
17) i= 2   j= 2
18) i= 2   j= 3
19) i= 3   j= 0
20) i= 3   j= 1
21) i= 3   j= 2
22) i= 3   j= 3
```

Q. Write a program to dispaly *'s in Right angled triangled form

```
1)  *
2)  * *
3)  * * *
4)  * * * *
5)  * * * * *
6)  * * * * * *
7)  * * * * * * *
8)
9)  n = int(input("Enter number of rows:"))
10) for i in range(1,n+1):
11)     for j in range(1,i+1):
12)         print("*",end=" ")
13)     print()
```

Alternative way:

```
1)  n = int(input("Enter number of rows:"))
2)  for i in range(1,n+1):
3)      print("* " * i)
```

Q. Write a program to display *'s in pyramid style(also known as equivalent triangle)

```
1)       *
2)      * *
3)     * * *
4)    * * * *
5)   * * * * *
6)  * * * * * *
7)  * * * * * * *
8)
9)  n = int(input("Enter number of rows:"))
10) for i in range(1,n+1):
11)     print(" " * (n-i),end="")
12)     print("* "*i)
```

III. Transfer Statements

1) break:

We can use break statement inside loops to break loop execution based on some condition.

Eg:

```
1)  for i in range(10):
2)      if i==7:
3)          print("processing is enough..plz break")
4)          break
5)      print(i)
6)
7)  D:\Python_classes>py test.py
8)  0
9)  1
10) 2
11) 3
12) 4
13) 5
14) 6
15) processing is enough..plz break
```

Eg:

```
1)  cart=[10,20,600,60,70]
2)  for item in cart:
3)      if item>500:
4)          print("To place this order insurence must be required")
5)          break
6)      print(item)
7)
8)  D:\Python_classes>py test.py
9)  10
10) 20
11) To place this order insurence must be required
```

2) continue:

We can use continue statement to skip current iteration and continue next iteration.

Eg 1: To print odd numbers in the range 0 to 9

```
1)  for i in range(10):
2)      if i%2==0:
3)          continue
4)      print(i)
5)
6)  D:\Python_classes>py test.py
7)  1
8)  3
9)  5
10) 7
11) 9
```

Eg 2:

```
1)  cart=[10,20,500,700,50,60]
2)  for item in cart:
3)      if item>=500:
4)          print("We cannot process this item :",item)
5)          continue
6)      print(item)
7)
8)  Output
9)  D:\Python_classes>py test.py
10) 10
11) 20
12) We cannot process this item : 500
13) We cannot process this item : 700
14) 50
15) 60
```

Eg 3:

```
1)  numbers=[10,20,0,5,0,30]
2)  for n in numbers:
3)      if n==0:
4)          print("Hey how we can divide with zero..just skipping")
5)          continue
6)      print("100/{} = {}".format(n,100/n))
7)
```

```
Output

100/10 = 10.0
100/20 = 5.0
Hey how we can divide with zero..just skipping
100/5 = 20.0
Hey how we can divide with zero..just skipping
100/30 = 3.3333333333333335
```

loops with else block:

Inside loop execution,if break statement not executed ,then only else part will be executed.

else means loop without break

Eg:

```
cart=[10,20,30,40,50]
for item in cart:
    if item>=500:
        print("We cannot process this order")
        break
    print(item)
else:
    print("Congrats ...all items processed successfully")

Output
10
20
30
40
50
Congrats ...all items processed successfully
```

Eg:

```
cart=[10,20,600,30,40,50]
for item in cart:
    if item>=500:
        print("We cannot process this order")
        break
    print(item)
else:
    print("Congrats ...all items processed successfully")
```

```
9)
10) Output
11) D:\Python_classes>py test.py
12) 10
13) 20
14) We cannot process this order
```

Q. What is the difference between for loop and while loop in Python?

We can use loops to repeat code execution
Repeat code for every item in sequence ==>for loop
Repeat code as long as condition is true ==>while loop

Q. How to exit from the loop?
by using break statement

Q. How to skip some iterations inside loop?
by using continue statement.

Q. When else part will be executed wrt loops?
If loop executed without break

3) pass statement:

pass is a keyword in Python.

In our programming syntactically if block is required which won't do anything then we can define that empty block with pass keyword.

pass
|- It is an empty statement
|- It is null statement
|- It won't do anything

Eg:

```
if True:
SyntaxError: unexpected EOF while parsing

if True: pass
==>valid

def m1():
SyntaxError: unexpected EOF while parsing
```

def m1(): pass

use case of pass:

Sometimes in the parent class we have to declare a function with empty body and child class responsible to provide proper implementation. Such type of empty body we can define by using pass keyword. (It is something like abstract method in java)

Eg:

```
def m1(): pass
```

Eg:

```
1)  for i in range(100):
2)      if i%9==0:
3)          print(i)
4)      else:pass
5)
6)  D:\Python_classes>py test.py
7)  0
8)  9
9)  18
10) 27
11) 36
12) 45
13) 54
14) 63
15) 72
16) 81
17) 90
18) 99
```

del statement:

del is a keyword in Python.

After using a variable, it is highly recommended to delete that variable if it is no longer required,so that the corresponding object is eligible for Garbage Collection.
We can delete variable by using del keyword.

Eg:

```
1)  x=10
2)  print(x)
3)  del x
```

After deleting a variable we cannot access that variable otherwise we will get NameError.

Eg:

```
1) x=10
2) del x
3) print(x)
```

NameError: name 'x' is not defined.

Note:

We can delete variables which are pointing to immutable objects.But we cannot delete the elements present inside immutable object.

Eg:

```
1) s="durga"
2) print(s)
3) del  s==>valid
4) del s[0] ==>TypeError: 'str' object doesn't support item deletion
```

Difference between del and None:

In the case del, the variable will be removed and we cannot access that variable(unbind operation)

```
1) s="durga"
2)  del s
3)  print(s)      ==>NameError: name 's' is not defined.
```

But in the case of None assignment the variable won't be removed but the corresponding object is eligible for Garbage Collection(re bind operation). Hence after assigning with None value,we can access that variable.

```
1) s="durga"
2) s=None
3) print(s)    # None
```

List Data Structure

If we want to represent a group of individual objects as a single entity where insertion order preserved and duplicates are allowed, then we should go for List.

insertion order preserved.
duplicate objects are allowed
heterogeneous objects are allowed.
List is dynamic because based on our requirement we can increase the size and decrease the size.
In List the elements will be placed within square brackets and with comma seperator.

We can differentiate duplicate elements by using index and we can preserve insertion order by using index. Hence index will play very important role.
Python supports both positive and negative indexes. +ve index means from left to right where as negative index means right to left

[10,"A","B",20, 30, 10]

-6	-5	-4	-3	-2	-1
10	A	B	20	30	10
0	1	2	3	4	5

List objects are mutable.i.e we can change the content.

Creation of List Objects:

1. We can create empty list object as follows...

```
1)  list=[]
2)  print(list)
3)  print(type(list))
4)
5)  []
6)  <class 'list'>
```

2. If we know elements already then we can create list as follows
list=[10,20,30,40]

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

3. With dynamic input:

```
1) list=eval(input("Enter List:"))
2) print(list)
3) print(type(list))
4)
5)  D:\Python_classes>py test.py
6) Enter List:[10,20,30,40]
7) [10, 20, 30, 40]
8) <class 'list'>
```

4. With list() function:

```
1) l=list(range(0,10,2))
2) print(l)
3) print(type(l))
4)
5) D:\Python_classes>py test.py
6) [0, 2, 4, 6, 8]
7) <class 'list'>
```

Eg:

```
1) s="durga"
2) l=list(s)
3) print(l)
4)
5) D:\Python_classes>py test.py
6) ['d', 'u', 'r', 'g', 'a']
```

5. with split() function:

```
1) s="Learning Python is very very easy !!!"
2) l=s.split()
3) print(l)
4) print(type(l))
5)
6) D:\Python_classes>py test.py
7) ['Learning', 'Python', 'is', 'very', 'very', 'easy', '!!!']
8) <class 'list'>
```

Note:

Sometimes we can take list inside another list,such type of lists are called nested lists.
[10,20,[30,40]]

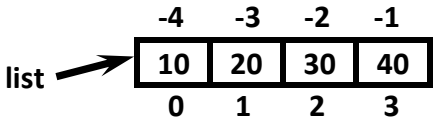
Accessing elements of List:

We can access elements of the list either by using index or by using slice operator(:)

1. By using index:

List follows zero based index. ie index of first element is zero.
List supports both +ve and -ve indexes.
+ve index meant for Left to Right
-ve index meant for Right to Left

list=[10,20,30,40]



print(list[0]) ==>10
print(list[-1]) ==>40
print(list[10]) ==>IndexError: list index out of range

2. By using slice operator:

Syntax:

list2= list1[start:stop:step]

start ==>it indicates the index where slice has to start
default value is 0

stop ===>It indicates the index where slice has to end
default value is max allowed index of list ie length of the list

step ==>increment value
default value is 1

Eg:

```
1) n=[1,2,3,4,5,6,7,8,9,10]
2) print(n[2:7:2])
3) print(n[4::2])
4) print(n[3:7])
5) print(n[8:2:-2])
6) print(n[4:100])
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

```
7)
8)  Output
9)  D:\Python_classes>py test.py
10) [3, 5, 7]
11) [5, 7, 9]
12) [4, 5, 6, 7]
13) [9, 7, 5]
14) [5, 6, 7, 8, 9, 10]
```

List vs mutability:

Once we creates a List object,we can modify its content. Hence List objects are mutable.

Eg:

```
1)  n=[10,20,30,40]
2)  print(n)
3)  n[1]=777
4)  print(n)
5)
6)  D:\Python_classes>py test.py
7)  [10, 20, 30, 40]
8)  [10, 777, 30, 40]
```

Traversing the elements of List:

The sequential access of each element in the list is called traversal.

1. By using while loop:

```
1)  n=[0,1,2,3,4,5,6,7,8,9,10]
2)  i=0
3)  while i<len(n):
4)      print(n[i])
5)      i=i+1
6)
7)  D:\Python_classes>py test.py
8)  0
9)  1
10) 2
11) 3
12) 4
13) 5
14) 6
15) 7
16) 8
17) 9
```

```
18) 10
```

2. By using for loop:

```
1)  n=[0,1,2,3,4,5,6,7,8,9,10]
2)  for n1 in n:
3)      print(n1)
4)
5)  D:\Python_classes>py test.py
6)  0
7)  1
8)  2
9)  3
10) 4
11) 5
12) 6
13) 7
14) 8
15) 9
16) 10
```

3. To display only even numbers:

```
1)  n=[0,1,2,3,4,5,6,7,8,9,10]
2)  for n1 in n:
3)      if n1%2==0:
4)          print(n1)
5)
6)  D:\Python_classes>py test.py
7)  0
8)  2
9)  4
10) 6
11) 8
12) 10
```

4. To display elements by index wise:

```
1)  l=["A","B","C"]
2)  x=len(l)
3)  for i in range(x):
4)      print(l[i],"is available at positive index: ",i,"and at negative index: ",i-x)
5)
6)  Output
7)  D:\Python_classes>py test.py
8)  A is available at positive index:  0 and at negative index:  -3
9)  B is available at positive index:  1 and at negative index:  -2
10) C is available at positive index:  2 and at negative index:  -1
```

Important functions of List:

I. To get information about list:

1. len():

returns the number of elements present in the list

Eg: n=[10,20,30,40]
print(len(n))==>4

2. count():

It returns the number of occurrences of specified item in the list

```
1)  n=[1,2,2,2,2,3,3]
2)  print(n.count(1))
3)  print(n.count(2))
4)  print(n.count(3))
5)  print(n.count(4))
6)
7)   Output
8)  D:\Python_classes>py test.py
9)  1
10) 4
11) 2
12) 0
```

3. index() function:

returns the index of first occurrence of the specified item.

Eg:

```
1)  n=[1,2,2,2,2,3,3]
2)  print(n.index(1))    ==>0
3)  print(n.index(2))    ==>1
4)  print(n.index(3))    ==>5
5)  print(n.index(4))    ==>ValueError: 4 is not in list
```

Note: If the specified element not present in the list then we will get ValueError.Hence before index() method we have to check whether item present in the list or not by using in operator.

print(4 in n)==>False

II. Manipulating elements of List:

1. append() function:

We can use append() function to add item at the end of the list.

Eg:

```
1)  list=[]
2)  list.append("A")
3)  list.append("B")
4)  list.append("C")
5)  print(list)
6)
7)  D:\Python_classes>py test.py
8)  ['A', 'B', 'C']
```

Eg: To add all elements to list upto 100 which are divisible by 10

```
1)  list=[]
2)  for i in range(101):
3)      if i%10==0:
4)          list.append(i)
5)  print(list)
6)
7)
8)  D:\Python_classes>py test.py
9)  [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

2. insert() function:

To insert item at specified index position

```
1)  n=[1,2,3,4,5]
2)  n.insert(1,888)
3)  print(n)
4)
5)  D:\Python_classes>py test.py
6)  [1, 888, 2, 3, 4, 5]
```

Eg:

```
1)  n=[1,2,3,4,5]
2)  n.insert(10,777)
3)  n.insert(-10,999)
4)  print(n)
5)
```

```
6)  D:\Python_classes>py test.py
7)  [999, 1, 2, 3, 4, 5, 777]
```

Note: If the specified index is greater than max index then element will be inserted at last position. If the specified index is smaller than min index then element will be inserted at first position.

Differences between append() and insert()

append()	insert()
In List when we add any element it will come in last i.e. it will be last element.	In List we can insert any element in particular index number

3. extend() function:

To add all items of one list to another list

l1.extend(l2)
all items present in l2 will be added to l1

Eg:

```
1)  order1=["Chicken","Mutton","Fish"]
2)  order2=["RC","KF","FO"]
3)  order1.extend(order2)
4)  print(order1)
5)
6)  D:\Python_classes>py test.py
7)  ['Chicken', 'Mutton', 'Fish', 'RC', 'KF', 'FO']
```

Eg:

```
1)  order=["Chicken","Mutton","Fish"]
2)  order.extend("Mushroom")
3)  print(order)
4)
5)  D:\Python_classes>py test.py
6)  ['Chicken', 'Mutton', 'Fish', 'M', 'u', 's', 'h', 'r', 'o', 'o', 'm']
```

4. remove() function:

We can use this function to remove specified item from the list.If the item present multiple times then only first occurrence will be removed.

```
1) n=[10,20,10,30]
2) n.remove(10)
3) print(n)
4)
5) D:\Python_classes>py test.py
6) [20, 10, 30]
```

If the specified item not present in list then we will get ValueError

```
1) n=[10,20,10,30]
2) n.remove(40)
3) print(n)
4)
5) ValueError: list.remove(x): x not in list
```

Note: Hence before using remove() method first we have to check specified element present in the list or not by using in operator.

5. pop() function:

It removes and returns the last element of the list.
This is only function which manipulates list and returns some element.

Eg:

```
1) n=[10,20,30,40]
2) print(n.pop())
3) print(n.pop())
4) print(n)
5)
6) D:\Python_classes>py test.py
7) 40
8) 30
9) [10, 20]
```

If the list is empty then pop() function raises IndexError

Eg:

```
1) n=[]
2) print(n.pop())  ==>   IndexError: pop from empty list
```

Note:

1. pop() is the only function which manipulates the list and returns some value
2. In general we can use append() and pop() functions to implement stack datastructure by using list,which follows LIFO(Last In First Out) order.

In general we can use pop() function to remove last element of the list. But we can use to remove elements based on index.

n.pop(index)===>To remove and return element present at specified index.
n.pop()==>To remove and return last element of the list

```
1)  n=[10,20,30,40,50,60]
2)  print(n.pop()) #60
3)  print(n.pop(1))    #20
4)  print(n.pop(10))  ==>IndexError: pop index out of range
```

Differences between remove() and pop()

remove()	pop()
1) We can use to remove special element from the List.	1) We can use to remove last element from the List.
2) It can't return any value.	2) It returned removed element.
3) If special element not available then we get VALUE ERROR.	3) If List is empty then we get Error.

Note:

List objects are dynamic. i.e based on our requirement we can increase and decrease the size.

append(),insert() ,extend() ===>for increasing the size/growable nature
remove(), pop() ======>for decreasing the size /shrinking nature

III. Ordering elements of List:

1. reverse():

We can use to reverse() order of elements of list.

```
1)  n=[10,20,30,40]
2)  n.reverse()
3)  print(n)
4)
5)  D:\Python_classes>py test.py
6)  [40, 30, 20, 10]
```

2. sort() function:

In list by default insertion order is preserved. If want to sort the elements of list according to default natural sorting order then we should go for sort() method.

For numbers ==>default natural sorting order is Ascending Order
For Strings ==> default natural sorting order is Alphabetical Order

```
1)  n=[20,5,15,10,0]
2)  n.sort()
3)  print(n)      #[0,5,10,15,20]
4)
5)  s=["Dog","Banana","Cat","Apple"]
6)  s.sort()
7)  print(s)  #['Apple','Banana','Cat','Dog']
```

Note: To use sort() function, compulsory list should contain only homogeneous elements. otherwise we will get TypeError

Eg:

```
1)  n=[20,10,"A","B"]
2)  n.sort()
3)  print(n)
4)
5)   TypeError: '<' not supported between instances of 'str' and 'int'
```

Note: In Python 2 if List contains both numbers and Strings then sort() function first sort numbers followed by strings

```
1)  n=[20,"B",10,"A"]
2)  n.sort()
```

```
3) print(n)# [10,20,'A','B']
```

But in Python 3 it is invalid.

To sort in reverse of default natural sorting order:

We can sort according to reverse of default natural sorting order by using reverse=True argument.

Eg:

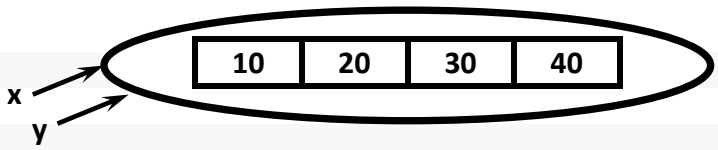
```
1. n=[40,10,30,20]
2. n.sort()
3. print(n) ==>[10,20,30,40]
4. n.sort(reverse=True)
5. print(n) ===>[40,30,20,10]
6. n.sort(reverse=False)
7. print(n) ==>[10,20,30,40]
```

Aliasing and Cloning of List objects:

The process of giving another reference variable to the existing list is called aliasing.

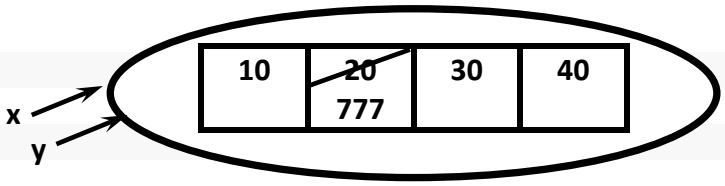
Eg:

```
1) x=[10,20,30,40]
2) y=x
3) print(id(x))
4) print(id(y))
```



The problem in this approach is by using one reference variable if we are changing content,then those changes will be reflected to the other reference variable.

```
1) x=[10,20,30,40]
2) y=x
3) y[1]=777
4) print(x) ==>[10,777,30,40]
```

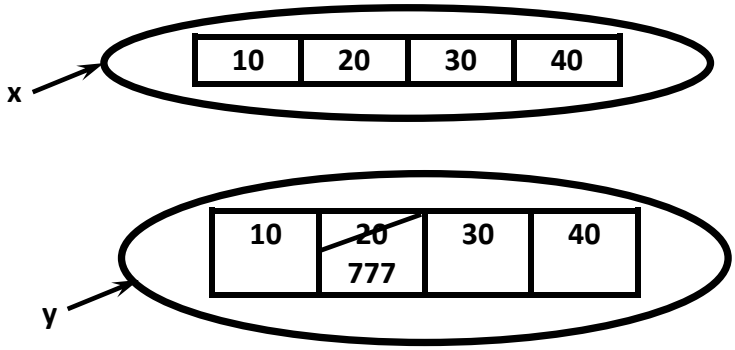


To overcome this problem we should go for cloning.
The process of creating exactly duplicate independent object is called cloning.

We can implement cloning by using slice operator or by using copy() function

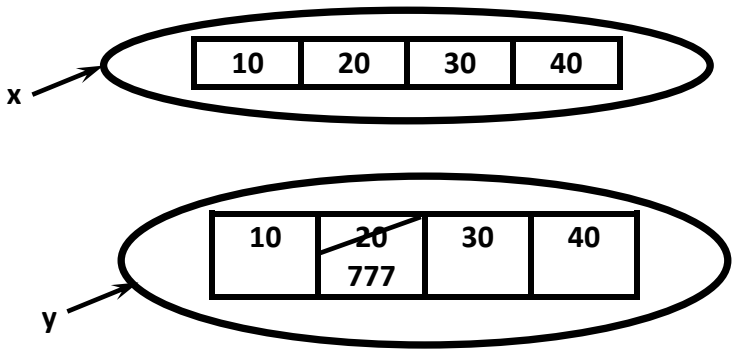
1. By using slice operator:

```
1) x=[10,20,30,40]
2) y=x[:]
3) y[1]=777
4) print(x) ==>[10,20,30,40]
5) print(y) ==>[10,777,30,40]
```



2. By using copy() function:

```
1) x=[10,20,30,40]
2) y=x.copy()
3) y[1]=777
4) print(x) ==>[10,20,30,40]
5) print(y) ==>[10,777,30,40]
```



Q. Difference between = operator and copy() function

= operator meant for aliasing
copy() function meant for cloning

Using Mathematical operators for List Objects:

We can use + and * operators for List objects.

1. Concatenation operator(+):

We can use + to concatenate 2 lists into a single list

```
1) a=[10,20,30]
2) b=[40,50,60]
3) c=a+b
4) print(c) ==>[10,20,30,40,50,60]
```

Note: To use + operator compulsory both arguments should be list objects,otherwise we will get TypeError.

Eg:

```
c=a+40      ==>TypeError: can only concatenate list (not "int") to list
c=a+[40]    ==>valid
```

2. Repetition Operator(*):

We can use repetition operator * to repeat elements of list specified number of times

Eg:

```
1) x=[10,20,30]
2) y=x*3
3) print(y)==>[10,20,30,10,20,30,10,20,30]
```

Comparing List objects

We can use comparison operators for List objects.

Eg:

```
1. x=["Dog","Cat","Rat"]
2. y=["Dog","Cat","Rat"]
3. z=["DOG","CAT","RAT"]
4. print(x==y)   True
5. print(x==z)   False
6. print(x != z)     True
```

Note:

Whenever we are using comparison operators(==,!=) for List objects then the following should be considered

1. **The number of elements**
2. **The order of elements**
3. **The content of elements (case sensitive)**

Note: When ever we are using relatational operators(<,<=,>,>=) between List objects,only first element comparison will be performed.

Eg:

```
1. x=[50,20,30]
2. y=[40,50,60,100,200]
3. print(x>y)  True
4. print(x>=y)  True
5. print(x<y)    False
6. print(x<=y)   False
```

Eg:

```
1. x=["Dog","Cat","Rat"]
2. y=["Rat","Cat","Dog"]
3. print(x>y)  False
4. print(x>=y) False
5. print(x<y)   True
6. print(x<=y)  True
```

Membership operators:

We can check whether element is a member of the list or not by using memebership operators.

in operator
not in operator

Eg:

```
1. n=[10,20,30,40]
2. print (10 in n)
3. print (10 not in n)
4. print (50 in n)
5. print (50 not in n)
6.
7. Output
```

```
8. True
9. False
10. False
11. True
```

clear() function:

We can use clear() function to remove all elements of List.

Eg:

```
1. n=[10,20,30,40]
2. print(n)
3. n.clear()
4. print(n)
5.
6. Output
7. D:\Python_classes>py test.py
8. [10, 20, 30, 40]
9. []
```

Nested Lists:

Sometimes we can take one list inside another list. Such type of lists are called nested lists.

Eg:

```
1. n=[10,20,[30,40]]
2. print(n)
3. print(n[0])
4. print(n[2])
5. print(n[2][0])
6. print(n[2][1])
7.
8. Output
9. D:\Python_classes>py test.py
10. [10, 20, [30, 40]]
11. 10
12. [30, 40]
13. 30
14. 40
```

Note: We can access nested list elements by using index just like accessing multi dimensional array elements.

Nested List as Matrix:

In Python we can represent matrix by using nested lists.

```
1)  n=[[10,20,30],[40,50,60],[70,80,90]]
2)  print(n)
3)  print("Elements by Row wise:")
4)  for r in n:
5)      print(r)
6)  print("Elements by Matrix style:")
7)  for i in range(len(n)):
8)      for j in range(len(n[i])):
9)          print(n[i][j],end=' ')
10)     print()
11)
12) Output
13) D:\Python_classes>py test.py
14) [[10, 20, 30], [40, 50, 60], [70, 80, 90]]
15) Elements by Row wise:
16) [10, 20, 30]
17) [40, 50, 60]
18) [70, 80, 90]
19) Elements by Matrix style:
20) 10 20 30
21) 40 50 60
22) 70 80 90
```

List Comprehensions:

It is very easy and compact way of creating list objects from any iterable objects(like list,tuple,dictionary,range etc) based on some condition.

Syntax:

list=[expression for item in list if condition]

Eg:

```
1)  s=[ x*x for x in range(1,11)]
2)  print(s)
3)  v=[2**x  for x in range(1,6)]
4)  print(v)
5)  m=[x for x in s if x%2==0]
6)  print(m)
7)
8)  Output
9)  D:\Python_classes>py test.py
10) [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
11) [2, 4, 8, 16, 32]
12) [4, 16, 36, 64, 100]
```

Eg:

```
1)  words=["Balaiah","Nag","Venkatesh","Chiranjeevi"]
2)  l=[w[0] for  w in words]
3)  print(l)
4)
5)  Output['B', 'N', 'V', 'C']
```

Eg:

```
1)  num1=[10,20,30,40]
2)  num2=[30,40,50,60]
3)  num3=[ i for i in num1 if i not in num2]
4)  print(num3)   [10,20]
5)
6)  common elements present in num1 and num2
7)  num4=[i for i in num1 if i in num2]
8)  print(num4)   [30, 40]
```

Eg:

```
1)  words="the quick brown fox jumps over the lazy dog".split()
2)  print(words)
3)  l=[[w.upper(),len(w)]     for w in words]
4)  print(l)
5)
6)  Output
7)  ['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
8)  [['THE', 3], ['QUICK', 5], ['BROWN', 5], ['FOX', 3], ['JUMPS', 5], ['OVER', 4],
9)  ['THE', 3], ['LAZY', 4], ['DOG', 3]]
```

Q. Write a program to display unique vowels present in the given word?

```
1)  vowels=['a','e','i','o','u']
2)  word=input("Enter the word to search for vowels: ")
3)  found=[]
4)  for letter in word:
5)      if letter in vowels:
6)          if letter not in found:
7)              found.append(letter)
8)  print(found)
9)  print("The number of different vowels present in",word,"is",len(found))
10)
11)
12) D:\Python_classes>py test.py
```

```
13) Enter the word to search for vowels: durgasoftwaresolutions
14) ['u', 'a', 'o', 'e', 'i']
15) The number of different vowels present in durgasoftwaresolutions is 5
```

list out all functions of list and write a program to use these functions

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Tuple Data Structure

1. Tuple is exactly same as List except that it is immutable. i.e once we creates Tuple object,we cannot perform any changes in that object.
Hence Tuple is Read Only version of List.
2. If our data is fixed and never changes then we should go for Tuple.
3. Insertion Order is preserved
4. Duplicates are allowed
5. Heterogeneous objects are allowed.
6. We can preserve insertion order and we can differentiate duplicate objects by using index. Hence index will play very important role in Tuple also.
Tuple support both +ve and -ve index. +ve index means forward direction(from left to right) and -ve index means backward direction(from right to left)
7. We can represent Tuple elements within Parenthesis and with comma seperator.
Parenethesis are optional but recommended to use.

Eg:

```
1.  t=10,20,30,40
2.  print(t)
3.  print(type(t))
4.
5.  Output
6.  (10, 20, 30, 40)
7.  <class 'tuple'>
8.
9.    t=()
10.  print(type(t)) # tuple
```

Note: We have to take special care about single valued tuple.compulsary the value should ends with comma,otherwise it is not treated as tuple.

Eg:

```
1.  t=(10)
2.  print(t)
3.  print(type(t))
4.
5.  Output
6.  10
7.  <class 'int'>
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Eg:

```
1. t=(10,)
2. print(t)
3. print(type(t))
4.
5. Output
6. (10,)
7. <class 'tuple'>
```

Q. Which of the following are valid tuples?

1. t=()
2. t=10,20,30,40
3. t=10
4. t=10,
5. t=(10)
6. t=(10,)
7. t=(10,20,30,40)

Tuple creation:

1. t=()

creation of empty tuple

2. t=(10,)

t=10,

creation of single valued tuple ,parenthesis are optional,should ends with comma

3. t=10,20,30

t=(10,20,30)

creation of multi values tuples & parenthesis are optional

4. By using tuple() function:

```
1. list=[10,20,30]
2.       t=tuple(list)
3.       print(t)
4.
5.       t=tuple(range(10,20,2))
6.       print(t)
```

Accessing elements of tuple:

We can access either by index or by slice operator

1. By using index:

```
1. t=(10,20,30,40,50,60)
2. print(t[0]) #10
3. print(t[-1])  #60
4. print(t[100])    IndexError: tuple index out of range
```

2. By using slice operator:

```
1. t=(10,20,30,40,50,60)
2. print(t[2:5])
3. print(t[2:100])
4. print(t[::2])
5.
6. Output
7. (30, 40, 50)
8. (30, 40, 50, 60)
9. (10, 30, 50)
```

Tuple vs immutability:

Once we creates tuple,we cannot change its content.
Hence tuple objects are immutable.

Eg:
t=(10,20,30,40)
t[1]=70 TypeError: 'tuple' object does not support item assignment

Mathematical operators for tuple:

We can apply + and * operators for tuple

1. Concatenation Operator(+):

```
1. t1=(10,20,30)
2. t2=(40,50,60)
3. t3=t1+t2
4. print(t3)  # (10,20,30,40,50,60)
```

2. Multiplication operator or repetition operator(*)

```
1. t1=(10,20,30)
2. t2=t1*3
3. print(t2) #(10,20,30,10,20,30,10,20,30)
```

Important functions of Tuple:

1. len()

To return number of elements present in the tuple

Eg:

```
t=(10,20,30,40)
print(len(t))  #4
```

2. count()

To return number of occurrences of given element in the tuple

Eg:

```
t=(10,20,10,10,20)
print(t.count(10)) #3
```

3. index()

returns index of first occurrence of the given element.
If the specified element is not available then we will get ValueError.

Eg:

```
t=(10,20,10,10,20)
print(t.index(10)) #0
print(t.index(30)) ValueError: tuple.index(x): x not in tuple
```

4. sorted()

To sort elements based on default natural sorting order

```
1. t=(40,10,30,20)
2. t1=sorted(t)
3. print(t1)
4. print(t)
5.
6.  Output
7.  [10, 20, 30, 40]
8. (40, 10, 30, 20)
```

We can sort according to reverse of default natural sorting order as follows

t1=sorted(t,reverse=True)
print(t1) [40, 30, 20, 10]

5. min() and max() functions:

These functions return min and max values according to default natural sorting order.

Eg:

```
1. t=(40,10,30,20)
2. print(min(t))  #10
3. print(max(t))  #40
```

6. cmp():

It compares the elements of both tuples.
If both tuples are equal then returns 0
If the first tuple is less than second tuple then it returns -1
If the first tuple is greater than second tuple then it returns +1

Eg:

```
1. t1=(10,20,30)
2. t2=(40,50,60)
3. t3=(10,20,30)
4. print(cmp(t1,t2))  # -1
5. print(cmp(t1,t3))  # 0
6. print(cmp(t2,t3))  # +1
```

Note: cmp() function is available only in Python2 but not in Python 3

Tuple Packing and Unpacking:

We can create a tuple by packing a group of variables.

Eg:

```
a=10
b=20
c=30
d=40
t=a,b,c,d
print(t) #(10, 20, 30, 40)
```

Here a,b,c,d are packed into a tuple t. This is nothing but tuple packing.

Tuple unpacking is the reverse process of tuple packing
We can unpack a tuple and assign its values to different variables

Eg:

```
t=(10,20,30,40)
a,b,c,d=t
print("a=",a,"b=",b,"c=",c,"d=",d)

Output
a= 10 b= 20 c= 30 d= 40
```

Note: At the time of tuple unpacking the number of variables and number of values should be same. ,otherwise we will get ValueError.

Eg:

```
t=(10,20,30,40)
a,b,c=t    #ValueError: too many values to unpack (expected 3)
```

Tuple Comprehension:

Tuple Comprehension is not supported by Python.

```
t= ( x**2  for x in range(1,6))
```

Here we are not getting tuple object and we are getting generator object.

```
t= ( x**2  for x in range(1,6))
print(type(t))
for x in t:
    print(x)

Output
D:\Python_classes>py test.py
<class 'generator'>
1
4
9
16
25
```

Q. Write a program to take a tuple of numbers from the keyboard and print its sum and average?

```
1.  t=eval(input("Enter Tuple of Numbers:"))
2.  l=len(t)
3.  sum=0
4.  for x in t:
5.      sum=sum+x
6.  print("The Sum=",sum)
7.  print("The Average=",sum/l)
8.
9.  D:\Python_classes>py test.py
10. Enter Tuple of Numbers:(10,20,30,40)
11. The Sum= 100
12. The Average= 25.0
13.
14. D:\Python_classes>py test.py
15. Enter Tuple of Numbers:(100,200,300)
16. The Sum= 600
17. The Average= 200.0
```

Differences between List and Tuple:

List and Tuple are exactly same except small difference: List objects are mutable where as Tuple objects are immutable.

In both cases insertion order is preserved, duplicate objects are allowed, heterogenous objects are allowed, index and slicing are supported.

List	Tuple
1) List is a Group of Comma separeated Values within Square Brackets and Square Brackets are mandatory. Eg: i = [10, 20, 30, 40]	1) Tuple is a Group of Comma separeated Values within Parenthesis and Parenthesis are optional. Eg: t = (10, 20, 30, 40) t = 10, 20, 30, 40
2) List Objects are Mutable i.e. once we creates List Object we can perform any changes in that Object. Eg: i[1] = 70	2) Tuple Objeccts are Immutable i.e. once we creates Tuple Object we cannot change its content. t[1] = 70 ➔ ValueError: tuple object does not support item assignment.
3) If the Content is not fixed and keep on changing then we should go for List.	3) If the content is fixed and never changes then we should go for Tuple.
4) List Objects can not used as Keys for Dictionries because Keys should be Hashable and Immutable.	4) Tuple Objects can be used as Keys for Dictionries because Keys should be Hashable and Immutable.

Set Data Structure

- If we want to represent a group of unique values as a single entity then we should go for set.
- Duplicates are not allowed.
- Insertion order is not preserved.But we can sort the elements.
- Indexing and slicing not allowed for the set.
- Heterogeneous elements are allowed.
- Set objects are mutable i.e once we creates set object we can perform any changes in that object based on our requirement.
- We can represent set elements within curly braces and with comma seperation
- We can apply mathematical operations like union,intersection,difference etc on set objects.

Creation of Set objects:

Eg:

```
1. s={10,20,30,40}
2. print(s)
3. print(type(s))
4.
5. Output
6. {40, 10, 20, 30}
7. <class 'set'>
```

We can create set objects by using set() function

s=set(any sequence)

Eg 1:

```
1. l = [10,20,30,40,10,20,10]
2. s=set(l)
3. print(s) # {40, 10, 20, 30}
```

Eg 2:

```
1. s=set(range(5))
2. print(s) #{0, 1, 2, 3, 4}
```

Note: While creating empty set we have to take special care.
Compulsory we should use set() function.

s={} ==>It is treated as dictionary but not empty set.

Eg:

```
1. s={}
2. print(s)
3. print(type(s))
4.
5. Output
6. {}
7. <class 'dict'>
```

Eg:

```
1. s=set()
2. print(s)
3. print(type(s))
4.
5. Output
6. set()
7. <class 'set'>
```

Important functions of set:

1. add(x):

Adds item x to the set

Eg:

```
1. s={10,20,30}
2. s.add(40);
3. print(s)  #{40, 10, 20, 30}
```

2. update(x,y,z):

To add multiple items to the set.
Arguments are not individual elements and these are Iterable objects like List,range etc.
All elements present in the given Iterable objects will be added to the set.

Eg:

```
1. s={10,20,30}
2. l=[40,50,60,10]
3. s.update(l,range(5))
4. print(s)
```

```
5.
6.  Output
7.  {0, 1, 2, 3, 4, 40, 10, 50, 20, 60, 30}
```

Q. What is the difference between add() and update() functions in set?

We can use add() to add individual item to the Set,where as we can use update() function to add multiple items to Set.
add() function can take only one argument where as update() function can take any number of arguments but all arguments should be iterable objects.

Q. Which of the following are valid for set s?

1. s.add(10)
2. s.add(10,20,30) TypeError: add() takes exactly one argument (3 given)
3. s.update(10) TypeError: 'int' object is not iterable
4. s.update(range(1,10,2),range(0,10,2))

3. copy():

Returns copy of the set.
It is cloned object.

```
s={10,20,30}
s1=s.copy()
print(s1)
```

4. pop():

It removes and returns some random element from the set.

Eg:

```
1.  s={40,10,30,20}
2.  print(s)
3.  print(s.pop())
4.  print(s)
5.
6.  Output
7.  {40, 10, 20, 30}
8.  40
9.  {10, 20, 30}
```

5. remove(x):

It removes specified element from the set.
If the specified element not present in the Set then we will get KeyError

```
s={40,10,30,20}
s.remove(30)
print(s)          # {40, 10, 20}
s.remove(50)   ==>KeyError: 50
```

6. discard(x):

It removes the specified element from the set.
If the specified element not present in the set then we won't get any error.

```
s={10,20,30}
s.discard(10)
print(s)          ===>{20, 30}
s.discard(50)
print(s)          ==>{20, 30}
```

Q. What is the difference between remove() and discard() functions in Set?
Q. Explain differences between pop(),remove() and discard() functionsin Set?

7.clear():

To remove all elements from the Set.

```
s={10,20,30}
print(s)
s.clear()
print(s)

Output
{10, 20, 30}
set()
```

Mathematical operations on the Set:

1.union():

x.union(y) ==>We can use this function to return all elements present in both sets

x.union(y) or x|y

Eg:

```
x={10,20,30,40}
y={30,40,50,60}
print(x.union(y))    #{10, 20, 30, 40, 50, 60}
print(x|y)       #{10, 20, 30, 40, 50, 60}
```

2. intersection():

x.intersection(y) or x&y

Returns common elements present in both x and y

Eg:

```
x={10,20,30,40}
y={30,40,50,60}
print(x.intersection(y))         #{40, 30}
print(x&y)      #{40, 30}
```

3. difference():

x.difference(y) or x-y
returns the elements present in x but not in y

Eg:

```
 x={10,20,30,40}
y={30,40,50,60}
print(x.difference(y))  #{10, 20}
print(x-y)       #{10, 20}
print(y-x)       #{50, 60}
```

4.symmetric_difference():

x.symmetric_difference(y) or x^y

Returns elements present in either x or y but not in both

Eg:

```
x={10,20,30,40}
y={30,40,50,60}
print(x.symmetric_difference(y))      #{10, 50, 20, 60}
print(x^y)       #{10, 50, 20, 60}
```

Membership operators: (in , not in)

Eg:

```
1. s=set("durga")
2. print(s)
3. print('d' in s)
4. print('z' in s)
5.
6. Output
7. {'u', 'g', 'r', 'd', 'a'}
8. True
9. False
```

Set Comprehension:

Set comprehension is possible.

```
s={x*x  for x   in range(5)}
print(s) #{0, 1, 4, 9, 16}


 s={2**x  for x in range(2,10,2)}
print(s)            #{16, 256, 64, 4}
```

set objects won't support indexing and slicing:

Eg:

```
s={10,20,30,40}
print(s[0])       ==>TypeError: 'set' object does not support indexing
print(s[1:3])  ==>TypeError: 'set' object is not subscriptable
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Q.Write a program to eliminate duplicates present in the list?

Approach-1:

```
l=eval(input("Enter List of values: "))
s=set(l)
print(s)

Output
D:\Python_classes>py test.py
Enter List of values: [10,20,30,10,20,40]
{40, 10, 20, 30}
```

Approach-2:

```
l=eval(input("Enter List of values: "))
l1=[]
for x in l:
    if x not in l1:
        l1.append(x)
print(l1)

Output
D:\Python_classes>py test.py
Enter List of values: [10,20,30,10,20,40]
[10, 20, 30, 40]
```

Q. Write a program to print different vowels present in the given word?

```
w=input("Enter word to search for vowels: ")
s=set(w)
v={'a','e','i','o','u'}
d=s.intersection(v)
print("The different vowel present in",w,"are",d)

Output
D:\Python_classes>py test.py
Enter word to search for vowels: durga
The different vowel present in durga are {'u', 'a'}
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Dictionary Data Structure

We can use List,Tuple and Set to represent a group of individual objects as a single entity.

If we want to represent a group of objects as key-value pairs then we should go for Dictionary.

Eg:

rollno----name
phone number--address
ipaddress---domain name

Duplicate keys are not allowed but values can be duplicated.
Hetrogeneous objects are allowed for both key and values.
insertion order is not preserved
Dictionaries are mutable
Dictionaries are dynamic
indexing and slicing concepts are not applicable

Note: In C++ and Java Dictionaries are known as "Map" where as in Perl and Ruby it is known as "Hash"

How to create Dictionary?

```
d={}    or   d=dict()
```

we are creating empty dictionary. We can add entries as follows

```
d[100]="durga"
d[200]="ravi"
d[300]="shiva"
print(d)   #{100: 'durga', 200: 'ravi', 300: 'shiva'}
```

If we know data in advance then we can create dictionary as follows

```
d={100:'durga' ,200:'ravi', 300:'shiva'}
```

```
d={key:value, key:value}
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

How to access data from the dictionary?

We can access data by using keys.

```
d={100:'durga' ,200:'ravi', 300:'shiva'}
print(d[100])    #durga
print(d[300])    #shiva
```

If the specified key is not available then we will get KeyError

```
print(d[400])  # KeyError: 400
```

We can prevent this by checking whether key is already available or not by using has_key() function or by using in operator.

d.has_key(400) ==> returns 1 if key is available otherwise returns 0

But has_key() function is available only in Python 2 but not in Python 3. Hence compulsory we have to use in operator.

```
if 400 in d:
        print(d[400])
```

Q. Write a program to enter name and percentage marks in a dictionary and display information on the screen

```
1)  rec={}
2)  n=int(input("Enter number of students: "))
3)  i=1
4)  while i <=n:
5)      name=input("Enter Student Name: ")
6)      marks=input("Enter % of Marks of Student: ")
7)      rec[name]=marks
8)      i=i+1
9)  print("Name of Student","\t","% of marks")
10) for x in rec:
11)     print("\t",x,"\t\t",rec[x])
12)
13) Output
14) D:\Python_classes>py test.py
15) Enter number of students: 3
16) Enter Student Name: durga
17) Enter % of Marks of Student: 60%
18) Enter Student Name: ravi
19) Enter % of Marks of Student: 70%
20) Enter Student Name: shiva
```

```
Enter % of Marks of Student: 80%
Name of Student        % of marks
        durga          60%
        ravi           70 %
        shiva          80%
```

How to update dictionaries?

d[key]=value

If the key is not available then a new entry will be added to the dictionary with the specified key-value pair

If the key is already available then old value will be replaced with new value.

Eg:

```
d={100:"durga",200:"ravi",300:"shiva"}
print(d)
d[400]="pavan"
print(d)
d[100]="sunny"
print(d)

Output
{100: 'durga', 200: 'ravi', 300: 'shiva'}
{100: 'durga', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
{100: 'sunny', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
```

How to delete elements from dictionary?

del d[key]

It deletes entry associated with the specified key.
If the key is not available then we will get KeyError

Eg:

```
d={100:"durga",200:"ravi",300:"shiva"}
print(d)
del d[100]
print(d)
del d[400]

Output
{100: 'durga', 200: 'ravi', 300: 'shiva'}
```

```
{200: 'ravi', 300: 'shiva'}
KeyError: 400
```

d.clear()

To remove all entries from the dictionary

Eg:

```
d={100:"durga",200:"ravi",300:"shiva"}
print(d)
d.clear()
print(d)

Output
{100: 'durga', 200: 'ravi', 300: 'shiva'}
{}
```

del d

To delete total dictionary.Now we cannot access d

Eg:

```
d={100:"durga",200:"ravi",300:"shiva"}
print(d)
del d
print(d)

Output
{100: 'durga', 200: 'ravi', 300: 'shiva'}
NameError: name 'd' is not defined
```

Important functions of dictionary:

1. dict():

To create a dictionary

d=dict() ===>It creates empty dictionary
d=dict({100:"durga",200:"ravi"}) ==>It creates dictionary with specified elements
d=dict([(100,"durga"),(200,"shiva"),(300,"ravi")])==>It creates dictionary with the given list of tuple elements

2. len()

Returns the number of items in the dictionary

3. clear():

To remove all elements from the dictionary

4. get():

To get the value associated with the key

d.get(key)
If the key is available then returns the corresponding value otherwise returns None.It wont raise any error.

d.get(key,defaultvalue)
If the key is available then returns the corresponding value otherwise returns default value.

Eg:

```
d={100:"durga",200:"ravi",300:"shiva"}
print(d[100]) ==>durga
print(d[400]) ==>KeyError:400
print(d.get(100)) ==durga
print(d.get(400)) ==>None
print(d.get(100,"Guest")) ==durga
print(d.get(400,"Guest")) ==>Guest
```

3. pop():

d.pop(key)
It removes the entry associated with the specified key and returns the corresponding value
If the specified key is not available then we will get KeyError

Eg:

```
1)  d={100:"durga",200:"ravi",300:"shiva"}
2)  print(d.pop(100))
3)  print(d)
4)  print(d.pop(400))
5)
6)  Output
```

```
durga
{200: 'ravi', 300: 'shiva'}
KeyError: 400
```

4. popitem():

It removes an arbitrary item(key-value) from the dictionaty and returns it.

Eg:

```
d={100:"durga",200:"ravi",300:"shiva"}
print(d)
print(d.popitem())
print(d)

Output
{100: 'durga', 200: 'ravi', 300: 'shiva'}
(300, 'shiva')
{100: 'durga', 200: 'ravi'}
```

If the dictionary is empty then we will get KeyError
d={}
print(d.popitem()) ==>KeyError: 'popitem(): dictionary is empty'

5. keys():

It returns all keys associated eith dictionary

Eg:

```
d={100:"durga",200:"ravi",300:"shiva"}
print(d.keys())
for k in d.keys():
    print(k)

Output
dict_keys([100, 200, 300])
100
200
300
```

6. values():

It returns all values associated with the dictionary

Eg:

```
d={100:"durga",200:"ravi",300:"shiva"}
print(d.values())
for v in d.values():
    print(v)

Output
dict_values(['durga', 'ravi', 'shiva'])
durga
ravi
shiva
```

7. items():

It returns list of tuples representing key-value pairs.

[(k,v),(k,v),(k,v)]

Eg:

```
d={100:"durga",200:"ravi",300:"shiva"}
for k,v in d.items():
    print(k,"--",v)

Output
100 -- durga
200 -- ravi
300 -- shiva
```

8. copy():

To create exactly duplicate dictionary(cloned copy)

d1=d.copy();

9. setdefault():

d.setdefault(k,v)

If the key is already available then this function returns the corresponding value.

If the key is not available then the specified key-value will be added as new item to the dictionary.

Eg:

```
d={100:"durga",200:"ravi",300:"shiva"}
print(d.setdefault(400,"pavan"))
print(d)
print(d.setdefault(100,"sachin"))
print(d)

Output
pavan
{100: 'durga', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
durga
{100: 'durga', 200: 'ravi', 300: 'shiva', 400: 'pavan'}
```

10. update():

d.update(x)

All items present in the dictionary x will be added to dictionary d

Q. Write a program to take dictionary from the keyboard and print the sum of values?

```
d=eval(input("Enter dictionary:"))
s=sum(d.values())
print("Sum= ",s)

Output
D:\Python_classes>py test.py
Enter dictionary:{'A':100,'B':200,'C':300}
Sum=  600
```

Q. Write a program to find number of occurrences of each letter present in the given string?

```
word=input("Enter any word: ")
d={}
for x in word:
    d[x]=d.get(x,0)+1
for k,v in d.items():
    print(k,"occurred ",v," times")

Output
D:\Python_classes>py test.py
Enter any word: mississippi
m occurred  1  times
i occurred  4  times
s occurred  4  times
```

```
14. p occurred  2  times
```

Q. Write a program to find number of occurrences of each vowel present in the given string?

```
1.  word=input("Enter any word: ")
2.  vowels={'a','e','i','o','u'}
3.  d={}
4.  for x in word:
5.      if x in vowels:
6.          d[x]=d.get(x,0)+1
7.  for k,v in sorted(d.items()):
8.      print(k,"occurred ",v," times")
9.
10. Output
11. D:\Python_classes>py test.py
12. Enter any word: doganimaldoganimal
13. a occurred  4  times
14. i occurred  2  times
15. o occurred  2  times
```

Q. Write a program to accept student name and marks from the keyboard and creates a dictionary. Also display student marks by taking student name as input?

```
1)  n=int(input("Enter the number of students: "))
2)  d={}
3)  for i in range(n):
4)      name=input("Enter Student Name: ")
5)      marks=input("Enter Student Marks: ")
6)      d[name]=marks
7)  while True:
8)      name=input("Enter Student Name to get Marks: ")
9)      marks=d.get(name,-1)
10)     if marks== -1:
11)         print("Student Not Found")
12)     else:
13)         print("The Marks of",name,"are",marks)
14)     option=input("Do you want to find another student marks[Yes|No]")
15)     if option=="No":
16)         break
17) print("Thanks for using our application")
18)
19) Output
20) D:\Python_classes>py test.py
21) Enter the number of students: 5
22) Enter Student Name: sunny
23) Enter Student Marks: 90
```

```
Enter Student Name: banny
Enter Student Marks: 80
Enter Student Name: chinny
Enter Student Marks: 70
Enter Student Name: pinny
Enter Student Marks: 60
Enter Student Name: vinny
Enter Student Marks: 50
Enter Student Name to get Marks: sunny
The Marks of sunny are 90
Do you want to find another student marks[Yes|No]Yes
Enter Student Name to get Marks: durga
Student Not Found
Do you want to find another student marks[Yes|No]No
Thanks for using our application
```

Dictionary Comprehension:

Comprehension concept applicable for dictionaries also.

```
squares={x:x*x for x in range(1,6)}
print(squares)
doubles={x:2*x for x in range(1,6)}
print(doubles)

Output
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
{1: 2, 2: 4, 3: 6, 4: 8, 5: 10}
```

FUNCTIONS

If a group of statements is repeatedly required then it is not recommended to write these statements everytime seperately.We have to define these statements as a single unit and we can call that unit any number of times based on our requirement without rewriting. This unit is nothing but function.

The main advantage of functions is code Reusability.
Note: In other languages functions are known as methods,procedures,subroutines etc

Python supports 2 types of functions

1. Built in Functions
2. User Defined Functions

1. Built in Functions:

The functions which are coming along with Python software automatically,are called built in functions or pre defined functions

Eg:
id()
type()
input()
eval()
etc..

2. User Defined Functions:

The functions which are developed by programmer explicitly according to business requirements ,are called user defined functions.

Syntax to create user defined functions:

```
def   function_name(parameters) :
  """ doc string"""
  ----
  -----
  return value
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Note: While creating functions we can use 2 keywords
1. def (mandatory)
2. return (optional)

Eg 1: Write a function to print Hello

test.py:

```
1) def wish():
2)     print("Hello Good Morning")
3) wish()
4) wish()
5) wish()
```

Parameters

Parameters are inputs to the function. If a function contains parameters,then at the time of calling,compulsory we should provide values otherwise,otherwise we will get error.

Eg: Write a function to take name of the student as input and print wish message by name.

```
1. def wish(name):
2.       print("Hello",name," Good Morning")
3. wish("Durga")
4. wish("Ravi")
5.
6.
7. D:\Python_classes>py test.py
8. Hello Durga  Good Morning
9. Hello Ravi  Good Morning
```

Eg: Write a function to take number as input and print its square value

```
1. def squareIt(number):
2.     print("The Square of",number,"is", number*number)
3. squareIt(4)
4. squareIt(5)
5.
6. D:\Python_classes>py test.py
7. The Square of 4 is 16
8. The Square of 5 is 25
```

Return Statement:

Function can take input values as parameters and executes business logic, and returns output to the caller with return statement.

Q. Write a function to accept 2 numbers as input and return sum.

```
def add(x,y):
    return x+y
result=add(10,20)
print("The sum is",result)
print("The sum is",add(100,200))


D:\Python_classes>py test.py
The sum is 30
The sum is 300
```

If we are not writing return statement then default return value is None

Eg:

```
def  f1():
    print("Hello")
f1()
print(f1())

Output
Hello
Hello
None
```

Q. Write a function to check whether the given number is even or odd?

```
def even_odd(num):
    if num%2==0:
        print(num,"is Even Number")
    else:
        print(num,"is Odd Number")
even_odd(10)
even_odd(15)

Output
D:\Python_classes>py test.py
10 is Even Number
15 is Odd Number
```

Q. Write a function to find factorial of given number?

```
1)  def fact(num):
2)      result=1
3)      while num>=1:
4)          result=result*num
5)          num=num-1
6)      return result
7)  for i in range(1,5):
8)      print("The Factorial of",i,"is :",fact(i))
9)
10) Output
11) D:\Python_classes>py test.py
12) The Factorial of 1 is : 1
13) The Factorial of 2 is : 2
14) The Factorial of 3 is : 6
15) The Factorial of 4 is : 24
```

Returning multiple values from a function:

In other languages like C,C++ and Java, function can return atmost one value. But in Python, a function can return any number of values.

Eg 1:

```
1)  def sum_sub(a,b):
2)      sum=a+b
3)      sub=a-b
4)      return sum,sub
5)  x,y=sum_sub(100,50)
6)  print("The Sum is :",x)
7)  print("The Subtraction is :",y)
8)
9)  Output
10) The Sum is : 150
11) The Subtraction is : 50
```

Eg 2:

```
1)   def calc(a,b):
2)      sum=a+b
3)      sub=a-b
4)      mul=a*b
5)      div=a/b
6)      return sum,sub,mul,div
7)  t=calc(100,50)
8)  print("The Results are")
```

```
for i in t:
    print(i)

Output
The Results are
150
50
5000
2.0
```

Types of arguments

```
def  f1(a,b):
   ------
   ------
   ------
 f1(10,20)
```

a,b are formal arguments where as 10,20 are actual arguments

There are 4 types are actual arguments are allowed in Python.

1. positional arguments
2. keyword arguments
3. default arguments
4. Variable length arguments

1. positional arguments:

These are the arguments passed to function in correct positional order.

```
def sub(a,b):
  print(a-b)

sub(100,200)
sub(200,100)
```

The number of arguments and position of arguments must be matched. If we change the order then result may be changed.

If we change the number of arguments then we will get error.

2. keyword arguments:

We can pass argument values by keyword i.e by parameter name.

Eg:

```
1. def wish(name,msg):
2.     print("Hello",name,msg)
3. wish(name="Durga",msg="Good Morning")
4. wish(msg="Good Morning",name="Durga")
5.
6. Output
7. Hello Durga Good Morning
8. Hello Durga Good Morning
```

Here the order of arguments is not important but number of arguments must be matched.

Note:

We can use both positional and keyword arguments simultaneously. But first we have to take positional arguments and then keyword arguments,otherwise we will get syntaxerror.

```
def wish(name,msg):
        print("Hello",name,msg)
wish("Durga","GoodMorning")          ==>valid
wish("Durga",msg="GoodMorning")    ==>valid
wish(name="Durga","GoodMorning")  ==>invalid
SyntaxError: positional argument follows keyword argument
```

3. Default Arguments:

Sometimes we can provide default values for our positional arguments.

Eg:

```
1) def wish(name="Guest"):
2)     print("Hello",name,"Good Morning")
3)
4) wish("Durga")
5) wish()
6)
7) Output
8) Hello Durga Good Morning
9) Hello Guest Good Morning
```

If we are not passing any name then only default value will be considered.

***Note:

After default arguments we should not take non default arguments

def wish(name="Guest",msg="Good Morning"): ===>Valid
def wish(name,msg="Good Morning"): ===>Valid
def wish(name="Guest",msg): ==>Invalid

SyntaxError: non-default argument follows default argument

4. Variable length arguments:

Sometimes we can pass variable number of arguments to our function,such type of arguments are called variable length arguments.

We can declare a variable length argument with * symbol as follows

def f1(*n):

We can call this function by passing any number of arguments including zero number. Internally all these values represented in the form of tuple.

Eg:

```
1.  def sum(*n):
2.      total=0
3.      for n1 in n:
4.          total=total+n1
5.      print("The Sum=",total)
6.
7.  sum()
8.  sum(10)
9.  sum(10,20)
10. sum(10,20,30,40)
11.
12. Output
13. The Sum= 0
14. The Sum= 10
15. The Sum= 30
16. The Sum= 100
```

Note:

We can mix variable length arguments with positional arguments.

Eg:

```
1.  def  f1(n1,*s):
2.     print(n1)
3.     for s1 in s:
4.        print(s1)
5.
6.  f1(10)
7.  f1(10,20,30,40)
8.  f1(10,"A",30,"B")
9.
10. Output
11. 10
12. 10
13. 20
14. 30
15. 40
16. 10
17. A
18. 30
19. B
```

Note: After variable length argument,if we are taking any other arguments then we should provide values as keyword arguments.

Eg:

```
1.  def  f1(*s,n1):
2.     for s1 in s:
3.        print(s1)
4.     print(n1)
5.
6.  f1("A","B",n1=10)
7.  Output
8.  A
9.  B
10. 10
```

f1("A","B",10) ==>Invalid
TypeError: f1() missing 1 required keyword-only argument: 'n1'

Note: We can declare key word variable length arguments also.
For this we have to use **.

```
def  f1(**n):
```

We can call this function by passing any number of keyword arguments. Internally these keyword arguments will be stored inside a dictionary.

Eg:

```
1)  def  display(**kwargs):
2)     for k,v in kwargs.items():
3)        print(k,"=",v)
4)  display(n1=10,n2=20,n3=30)
5)  display(rno=100,name="Durga",marks=70,subject="Java")
6)
7)  Output
8)  n1 = 10
9)  n2 = 20
10) n3 = 30
11) rno = 100
12) name = Durga
13) marks = 70
14) subject = Java
```

Case Study:

```
def f(arg1,arg2,arg3=4,arg4=8):
        print(arg1,arg2,arg3,arg4)
```

1. f(3,2) ==> 3 2 4 8

2. f(10,20,30,40) ===>10 20 30 40

3. f(25,50,arg4=100) ==>25 50 4 100

4. f(arg4=2,arg1=3,arg2=4)===>3 4 4 2
5. f()===>Invalid
   TypeError: f() missing 2 required positional arguments: 'arg1' and 'arg2'

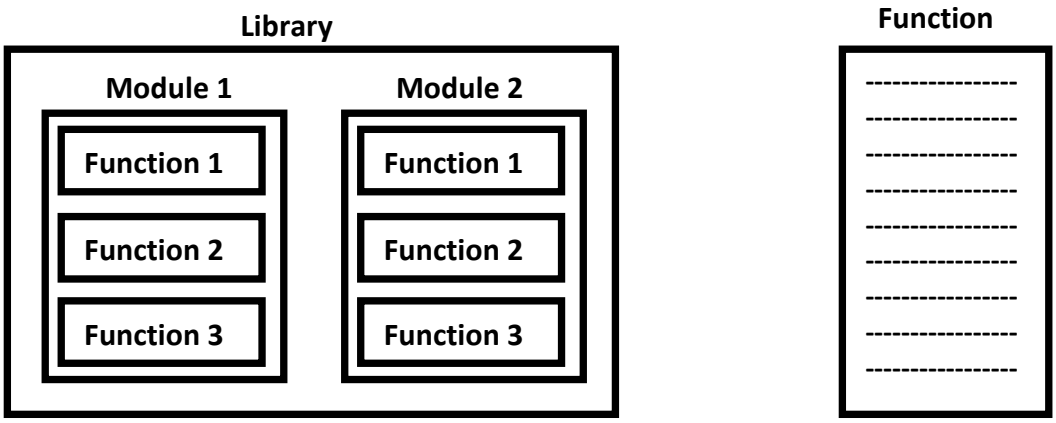
6. f(arg3=10,arg4=20,30,40) ==>Invalid
   SyntaxError: positional argument follows keyword argument
   [After keyword arguments we should not take positional arguments]

7. f(4,5,arg2=6)==>Invalid
   TypeError: f() got multiple values for argument 'arg2'

8. f(4,5,arg3=5,arg5=6)==>Invalid
   TypeError: f() got an unexpected keyword argument 'arg5'

Note: **Function vs Module vs Library:**

1. A group of lines with some name is called a function
2. A group of functions saved to a file , is called Module
3. A group of Modules is nothing but Library



Types of Variables

Python supports 2 types of variables.

1. Global Variables
2. Local Variables

1. Global Variables

The variables which are declared outside of function are called global variables.
These variables can be accessed in all functions of that module.

Eg:

```
1)  a=10  # global variable
2)  def f1():
3)      print(a)
4)
5)  def f2():
6)      print(a)
7)
8)  f1()
9)  f2()
10)
11) Output
12) 10
13) 10
```

2. Local Variables:

The variables which are declared inside a function are called local variables.
Local variables are available only for the function in which we declared it.i.e from outside of function we cannot access.

Eg:

```
1)  def f1():
2)      a=10
3)      print(a)    # valid
4)
5)  def f2():
6)      print(a)     #invalid
7)
8)  f1()
9)  f2()
10)
11) NameError: name 'a' is not defined
```

global keyword:

We can use global keyword for the following 2 purposes:

1. To declare global variable inside function
2. To make global variable available to the function so that we can perform required modifications

Eg 1:

```
1)  a=10
2)  def f1():
3)      a=777
4)      print(a)
5)
6)  def f2():
7)      print(a)
8)
9)  f1()
10) f2()
11)
12) Output
13) 777
14) 10
```

Eg 2:

```
1)  a=10
2)  def f1():
3)      global a
4)      a=777
5)      print(a)
6)
7)  def f2():
8)      print(a)
9)
10) f1()
11) f2()
12)
13) Output
14) 777
15) 777
```

Eg 3:

```
1)  def f1():
2)      a=10
3)      print(a)
4)
5)  def f2():
6)      print(a)
7)
8)  f1()
9)  f2()
10)
11) NameError: name 'a' is not defined
```

Eg 4:

```
1)  def f1():
2)      global a
3)      a=10
4)      print(a)
5)
6)  def f2():
7)      print(a)
8)
9)  f1()
10) f2()
11)
12) Output
13) 10
14) 10
```

Note: If global variable and local variable having the same name then we can access global variable inside a function as follows

```
1)  a=10    #global variable
2)  def f1():
3)     a=777  #local variable
4)     print(a)
5)     print(globals()['a'])
6)  f1()
7)
8)
9)  Output
10) 777
11) 10
```

Recursive Functions

A function that calls itself is known as Recursive Function.

Eg:

```
factorial(3)=3*factorial(2)
            =3*2*factorial(1)
            =3*2*1*factorial(0)
            =3*2*1*1
                  =6
factorial(n)= n*factorial(n-1)
```

The main advantages of recursive functions are:

1. We can reduce length of the code and improves readability
2. We can solve complex problems very easily.

Q. Write a Python Function to find factorial of given number with recursion.

Eg:

```
1)   def  factorial(n):
2)     if n==0:
3)        result=1
4)     else:
5)        result=n*factorial(n-1)
6)     return result
7)  print("Factorial of 4 is :",factorial(4))
8)  print("Factorial of 5 is :",factorial(5))
9)
10) Output
```

```
11) Factorial of 4 is : 24
12) Factorial of 5 is : 120
```

Anonymous Functions:

Sometimes we can declare a function without any name,such type of nameless functions are called anonymous functions or lambda functions.

The main purpose of anonymous function is just for instant use(i.e for one time usage)

Normal Function:

We can define by using def keyword.

```
def  squareIt(n):
    return n*n
```

lambda Function:

We can define by using lambda keyword

```
lambda n:n*n
```

Syntax of lambda Function:

```
lambda   argument_list : expression
```

Note: By using Lambda Functions we can write very concise code so that readability of the program will be improved.

Q. Write a program to create a lambda function to find square of given number?

```
1)  s=lambda n:n*n
2)  print("The Square of 4 is :",s(4))
3)  print("The Square of 5 is :",s(5))
4)
5)  Output
6)  The Square of 4 is : 16
7)  The Square of 5 is : 25
```

Q. Lambda function to find sum of 2 given numbers

```
1)  s=lambda a,b:a+b
2)  print("The Sum of 10,20 is:",s(10,20))
```

```
3)  print("The Sum of 100,200 is:",s(100,200))
4)
5)  Output
6)  The Sum of 10,20 is: 30
7)  The Sum of 100,200 is: 300
```

Q. Lambda Function to find biggest of given values.

```
1)  s=lambda a,b:a if a>b else b
2)  print("The Biggest of 10,20 is:",s(10,20))
3)  print("The Biggest of 100,200 is:",s(100,200))
4)
5)  Output
6)  The Biggest of 10,20 is: 20
7)  The Biggest of 100,200 is: 200
```

Note:

Lambda Function internally returns expression value and we are not required to write return statement explicitly.

Note: Sometimes we can pass function as argument to another function. In such cases lambda functions are best choice.

We can use lambda functions very commonly with filter(),map() and reduce() functions,b'z these functions expect function as argument.

filter() function:

We can use filter() function to filter values from the given sequence based on some condition.

filter(function,sequence)

where function argument is responsible to perform conditional check
sequence can be list or tuple or string.

Q. Program to filter only even numbers from the list by using filter() function?

without lambda Function:

```
1)  def isEven(x):
2)      if x%2==0:
3)          return True
4)      else:
```

```
5)        return False
6)  l=[0,5,10,15,20,25,30]
7)  l1=list(filter(isEven,l))
8)  print(l1)  #[0,10,20,30]
```

with lambda Function:

```
1)  l=[0,5,10,15,20,25,30]
2)  l1=list(filter(lambda x:x%2==0,l))
3)  print(l1)   #[0,10,20,30]
4)  l2=list(filter(lambda x:x%2!=0,l))
5)  print(l2) #[5,15,25]
```

map() function:

For every element present in the given sequence,apply some functionality and generate new element with the required modification. For this requirement we should go for map() function.

Eg: For every element present in the list perform double and generate new list of doubles.

Syntax:

map(function,sequence)

The function can be applied on each element of sequence and generates new sequence.

Eg: Without lambda

```
1)   l=[1,2,3,4,5]
2)  def doubleIt(x):
3)     return 2*x
4)  l1=list(map(doubleIt,l))
5)  print(l1) #[2, 4, 6, 8, 10]
```

with lambda

```
1)  l=[1,2,3,4,5]
2)  l1=list(map(lambda x:2*x,l))
3)  print(l1)    #[2, 4, 6, 8, 10]
```

Eg 2: To find square of given numbers

```
1. l=[1,2,3,4,5]
2. l1=list(map(lambda x:x*x,l))
3. print(l1)      #[1, 4, 9, 16, 25]
```

We can apply map() function on multiple lists also.But make sure all list should have same length.

Syntax: map(lambda x,y:x*y,l1,l2))
x is from l1 and y is from l2

Eg:

```
1. l1=[1,2,3,4]
2. l2=[2,3,4,5]
3. l3=list(map(lambda x,y:x*y,l1,l2))
4. print(l3)      #[2, 6, 12, 20]
```

reduce() function:

reduce() function reduces sequence of elements into a single element by applying the specified function.

reduce(function,sequence)

reduce() function present in functools module and hence we should write import statement.

Eg:

```
1) from functools import *
2) l=[10,20,30,40,50]
3) result=reduce(lambda x,y:x+y,l)
4) print(result) # 150
```

Eg:

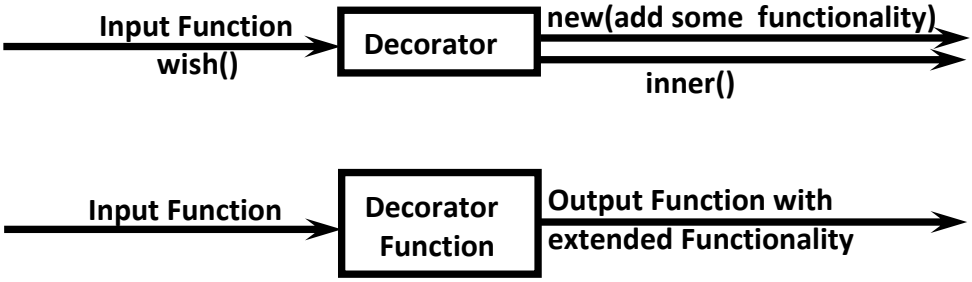
```
1) result=reduce(lambda x,y:x*y,l)
2) print(result)  #12000000
```

Eg:

```
1) from functools import *
2) result=reduce(lambda x,y:x+y,range(1,101))
3) print(result)  #5050
```

Function Decorators:

Decorator is a function which can take a function as argument and extend its functionality and returns modified function with extended functionality.



The main objective of decorator functions is we can extend the functionality of existing functions without modifies that function.

```
1)  def wish(name):
2)    print("Hello",name,"Good Morning")
```

This function can always print same output for any name

Hello Durga Good Morning
Hello Ravi Good Morning
Hello Sunny Good Morning

But we want to modify this function to provide different message if name is Sunny.
We can do this without touching wish() function by using decorator.

Eg:

```
1)  def decor(func):
2)    def inner(name):
3)      if name=="Sunny":
4)        print("Hello Sunny Bad Morning")
5)      else:
6)        func(name)
7)    return inner
8)
9)  @decor
10) def wish(name):
11)   print("Hello",name,"Good Morning")
12)
13) wish("Durga")
14) wish("Ravi")
15) wish("Sunny")
16)
```

```
17) Output
18) Hello Durga Good Morning
19) Hello Ravi Good Morning
20) Hello Sunny Bad Morning
```

In the above program whenever we call wish() function automatically decor function will be executed.

How to call same function with decorator and without decorator:

We should not use @decor

```
1)  def decor(func):
2)      def inner(name):
3)          if name=="Sunny":
4)              print("Hello Sunny Bad Morning")
5)          else:
6)              func(name)
7)      return inner
8)
9)
10) def wish(name):
11)     print("Hello",name,"Good Morning")
12)
13) decorfunction=decor(wish)
14)
15) wish("Durga") #decorator wont be executed
16) wish("Sunny") #decorator wont be executed
17)
18) decorfunction("Durga")#decorator will be executed
19) decorfunction("Sunny")#decorator will be executed
20)
21) Output
22) Hello Durga Good Morning
23) Hello Sunny Good Morning
24) Hello Durga Good Morning
25) Hello Sunny Bad Morning
```

Eg 2:

```
1)  def  smart_division(func):
2)      def inner(a,b):
3)          print("We are dividing",a,"with",b)
4)          if b==0:
5)              print("OOPS...cannot divide")
6)              return
7)          else:
8)              return func(a,b)
```

```
9)      return inner
10)
11) @smart_division
12) def  division(a,b):
13)     return a/b
14)
15) print(division(20,2))
16) print(division(20,0))
17)
18) without decorator we will get Error.In this case output is:
19)
20) 10.0
21) Traceback (most recent call last):
22)   File "test.py", line 16, in <module>
23)     print(division(20,0))
24)   File "test.py", line 13, in division
25)     return a/b
26) ZeroDivisionError: division by zero
```

with decorator we won't get any error. In this case output is:

We are dividing 20 with 2
10.0
We are dividing 20 with 0
OOPS...cannot divide
None

Decorator Chaining

We can define multiple decorators for the same function and all these decorators will form Decorator Chaining.

Eg:
@decor1
@decor
def num():

For num() function we are applying 2 decorator functions. First inner decorator will work and then outer decorator.

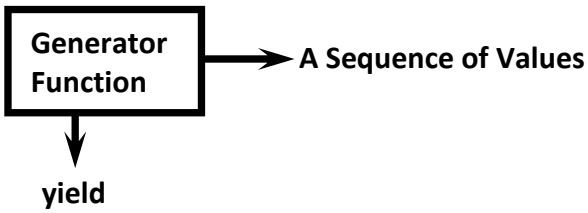
Eg:

```
1)  def decor1(func):
2)      def inner():
3)          x=func()
4)          return x*x
```

```
5)     return inner
6)
7)  def decor(func):
8)     def inner():
9)        x=func()
10)       return 2*x
11)    return inner
12)
13) @decor1
14) @decor
15) def num():
16)    return 10
17)
18) print(num())
```

Generators

Generator is a function which is responsible to generate a sequence of values.
We can write generator functions just like ordinary functions, but it uses yield keyword to return values.



Eg 1:

```
1)  def mygen():
2)     yield 'A'
3)     yield 'B'
4)     yield 'C'
5)
6)  g=mygen()
7)  print(type(g))
8)
9)  print(next(g))
10) print(next(g))
11) print(next(g))
12) print(next(g))
13)
14) Output
15) <class 'generator'>
16) A
17) B
18) C
```

```
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print(next(g))
StopIteration
```

Eg 2:

```
def countdown(num):
    print("Start Countdown")
    while(num>0):
        yield num
        num=num-1

values=countdown(5)
for x in values:
    print(x)

Output
Start Countdown
5
4
3
2
1
```

Eg 3: To generate first n numbers:

```
def firstn(num):
    n=1
    while n<=num:
        yield n
        n=n+1

values=firstn(5)
for x in values:
    print(x)

Output
1
2
3
4
5
```

We can convert generator into list as follows:

```
values=firstn(10)
l1=list(values)
print(l1)    #[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Eg 4: To generate Fibonacci Numbers...

The next is the sum of previous 2 numbers

Eg: 0,1,1,2,3,5,8,13,21,...

```
1)  def fib():
2)      a,b=0,1
3)      while True:
4)          yield a
5)          a,b=b,a+b
6)  for f in fib():
7)      if f>100:
8)          break
9)      print(f)
10)
11) Output
12) 0
13) 1
14) 1
15) 2
16) 3
17) 5
18) 8
19) 13
20) 21
21) 34
22) 55
23) 89
```

Advantages of generator functions:

1. when compared with class level iterators, generators are very easy to use
2. Improves memory utilization and performance.
3. Generators are best suitable for reading data from large number of large files
4. Generators work great for web scraping and crawling

Modules

A group of functions ,variables and classes saved to a file,which is nothing but module.

Every Python file(.py) acts as a module.

Eg: durgamath.py

```
1)    x=888
2)
3)  def add(a,b):
4)    print("The Sum:",a+b)
5)
6)  def product(a,b):
7)    print("The Product:",a*b)
```

durgamath module contains one variable and 2 functions.

If we want to use members of module in our program then we should import that module.

import modulename

We can access members by using module name.
 modulename.variable
 modulename.function()

test.py:

```
1)  import durgamath
2)  print(durgamath.x)
3)  durgamath.add(10,20)
4)  durgamath.product(10,20)
5)
6)   Output
7)   888
8)  The Sum: 30
9)  The Product: 200
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Note:

whenever we are using a module in our program, for that module compiled file will be generated and stored in the hard disk permanently.

Renaming a module at the time of import (module aliasing):

Eg:

```
import durgamath as m
```

here durgamath is original module name and m is alias name.
We can access members by using alias name m

test.py:

```
1) import durgamath as m
2) print(m.x)
3) m.add(10,20)
4) m.product(10,20)
```

from ... import:

We can import particular members of module by using from ... import .
The main advantage of this is we can access members directly without using module name.

Eg:

```
from durgamath import x,add
print(x)
add(10,20)
product(10,20)==> NameError: name 'product' is not defined
```

We can import all members of a module as follows

```
from durgamath import *
```

test.py:

```
1) from durgamath import *
2) print(x)
3) add(10,20)
4) product(10,20)
```

Various possibilties of import:

```
import modulename
import module1,module2,module3
import module1 as m
import module1 as m1,module2 as m2,module3
from module import member
from module import member1,member2,memebr3
from module import memeber1 as x
from   module import *
```

member aliasing:

```
from durgamath import x as y,add as sum
print(y)
sum(10,20)
```

Once we defined as alias name,we should use alias name only and we should not use original name

Eg:

```
from durgamath import x as y
print(x)==>NameError: name 'x' is not defined
```

Reloading a Module:

By default module will be loaded only once eventhough we are importing multiple multiple times.

module1.py:

```
print("This is  from module1")
```

test.py

```
import module1
import module1
import module1
import module1
print("This is test module")

Output
This is  from module1
This is test module
```

In the above program test module will be loaded only once eventhough we are importing multiple times.

The problem in this approach is after loading a module if it is updated outside then updated version of module1 is not available to our program.

We can solve this problem by reloading module explicitly based on our requirement.
We can reload by using reload() function of imp module.

```
import imp
imp.reload(module1)
```

test.py:

```
1) import module1
2) import module1
3) from imp import reload
4) reload(module1)
5) reload(module1)
6) reload(module1)
7) print("This is test module")
```

In the above program module1 will be loaded 4 times in that 1 time by default and 3 times explicitly. In this case output is

```
1) This is  from module1
2) This is  from module1
3) This is  from module1
4) This is  from module1
5) This is test module
```

The main advantage of explicit module reloading is we can ensure that updated version is always available to our program.

Finding members of module by using dir() function:

Python provides inbuilt function dir() to list out all members of current module or a specified module.

dir() ===>To list out all members of current module
dir(moduleName)==>To list out all members of specified module

Eg 1: test.py

```
1) x=10
2) y=20
```

```
3)  def f1():
4)      print("Hello")
5)  print(dir())    # To print all members of current module
6)
7)  Output
8)  ['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__nam
    e__', '__package__', '__spec__', 'f1', 'x', 'y']
```

Eg 2: To display members of particular module:

durgamath.py:

```
1.  x=888
2.
3.  def add(a,b):
4.    print("The Sum:",a+b)
5.
6.  def product(a,b):
7.    print("The Product:",a*b)
```

test.py:

```
1.  import durgamath
2.  print(dir(durgamath))
3.
4.  Output
5.  ['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
6.  '__package__', '__spec__', 'add', 'product', 'x']
```

Note: For every module at the time of execution Python interpreter will add some special properties automatically for internal use.

Eg: __builtins__,__cached__,'__doc__,__file__, __loader__, __name__,__package__, __spec__

Based on our requirement we can access these properties also in our program.

Eg: test.py:

```
1.  print(__builtins__  )
2.  print(__cached__   )
3.  print(__doc__)
4.  print(__file__)
5.  print(__loader__)
6.  print(__name__)
7.  print(__package__)
8.  print(__spec__)
9.
```

```
Output
<module 'builtins' (built-in)>
None
None
```

test.py

```
<_frozen_importlib_external.SourceFileLoader object at 0x00572170>
__main__
None
None
```

The Special variable __name__:

For every Python program , a special variable __name__ will be added internally. This variable stores information regarding whether the program is executed as an individual program or as a module.

If the program executed as an individual program then the value of this variable is __main__
If the program executed as a module from some other program then the value of this variable is the name of module where it is defined.

Hence by using this __name__ variable we can identify whether the program executed directly or as a module.

Demo program:

module1.py:

```
def f1():
    if __name__=='__main__':
        print("The code executed as a program")
    else:
        print("The code executed as a module from some other program")
f1()
```

test.py:

```
import module1
module1.f1()

D:\Python_classes>py module1.py
The code executed as a program

D:\Python_classes>py test.py
```

```
8) The code executed as a module from some other program
9) The code executed as a module from some other program
```

Working with math module:

Python provides inbuilt module math.
This module defines several functions which can be used for mathematical operations.
The main important functions are

1. sqrt(x)
2. ceil(x)
3. floor(x)
4. fabs(x)
5.log(x)
6. sin(x)
7. tan(x)
....

Eg:

```
1) from math import *
2) print(sqrt(4))
3) print(ceil(10.1))
4) print(floor(10.1))
5) print(fabs(-10.6))
6) print(fabs(10.6))
7)
8) Output
9) 2.0
10) 11
11) 10
12) 10.6
13) 10.6
```

Note:

We can find help for any module by using help() function

Eg:

```
import math
help(math)
```

Working with random module:

This module defines several functions to generate random numbers.
We can use these functions while developing games,in cryptography and to generate random numbers on fly for authentication.

1. random() function:

This function always generate some float value between 0 and 1 (not inclusive)

0<x<1

Eg:

```
1)  from random import *
2)  for i in range(10):
3)      print(random())
4)
5)   Output
6)   0.4572685609302056
7)  0.6584325233197768
8)  0.15444034016553587
9)  0.18351427005232201
10) 0.1330257265904884
11) 0.9291139798071045
12) 0.6586741197891783
13) 0.8901649834019002
14) 0.25540891083913053
15) 0.7290504335962871
```

2. randint() function:

To generate random integer beween two given numbers(inclusive)

Eg:

```
1)  from random import *
2)  for i in range(10):
3)      print(randint(1,100)) # generate random int value between 1 and 100(inclusive)
4)
5)  Output
6)  51
7)  44
8)  39
9)  70
10) 49
11) 74
12) 52
```

```
13) 10
14) 40
15) 8
```

3. uniform():

It returns random float values between 2 given numbers(not inclusive)

Eg:

```
1)  from random import *
2)  for i in range(10):
3)      print(uniform(1,10))
4)
5)  Output
6)  9.787695398230332
7)  6.81102218793548
8)  8.068672144377329
9)  8.567976357239834
10) 6.363511674803802
11) 2.176137584071641
12) 4.822867939432386
13) 6.0801725149678445
14) 7.508457735544763
15) 1.9982221862917555
```

random() ===>in between 0 and 1 (not inclusive)
randint(x,y) ==>in between x and y (inclusive)
uniform(x,y) ==> in between x and y (not inclusive)

4. randrange([start],stop,[step])

returns a random number from range
start<= x < stop
start argument is optional and default value is 0
step argument is optional and default value is 1

randrange(10)-->generates a number from 0 to 9
randrange(1,11)-->generates a number from 1 to 10
randrange(1,11,2)-->generates a number from 1,3,5,7,9

Eg 1:

```
1)  from random import *
2)  for i in range(10):
3)      print(randrange(10))
4)
```

```
Output
9
4
0
2
9
4
8
9
5
9
```

Eg 2:

```
from random import *
for i in range(10):
    print(randrange(1,11))

Output
2
2
8
10
3
5
9
1
6
3
```

Eg 3:

```
from random import *
for i in range(10):
    print(randrange(1,11,2))

Output
1
3
9
5
7
1
1
1
7
3
```

5. choice() function:

It wont return random number.
It will return a random object from the given list or tuple.

Eg:

```
1)  from random import *
2)  list=["Sunny","Bunny","Chinny","Vinny","pinny"]
3)  for i in range(10):
4)      print(choice(list))
5)
6)  Output
7)  Bunny
8)  pinny
9)  Bunny
10) Sunny
11) Bunny
12) pinny
13) pinny
14) Vinny
15) Bunny
16) Sunny
```

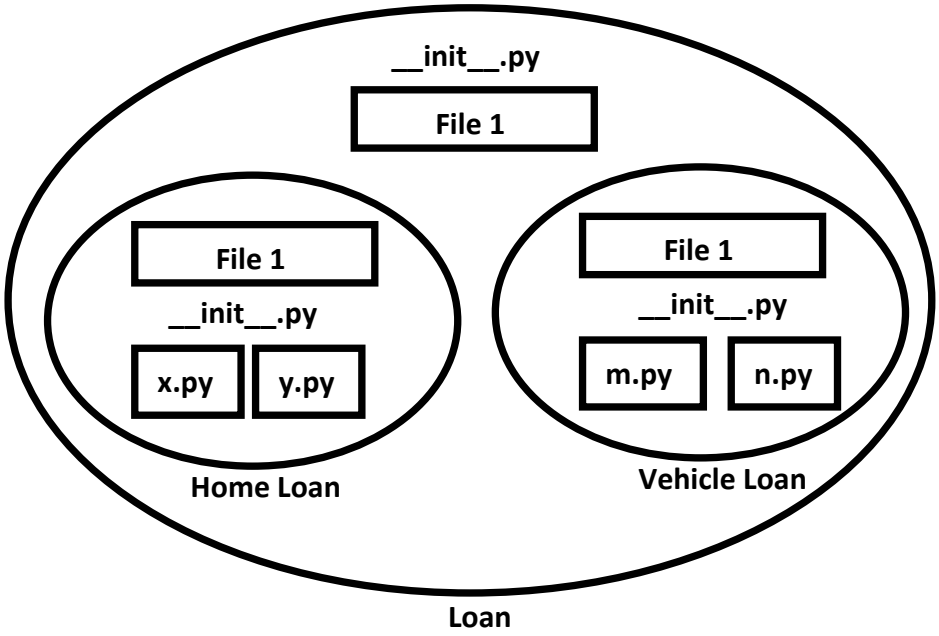

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Packages

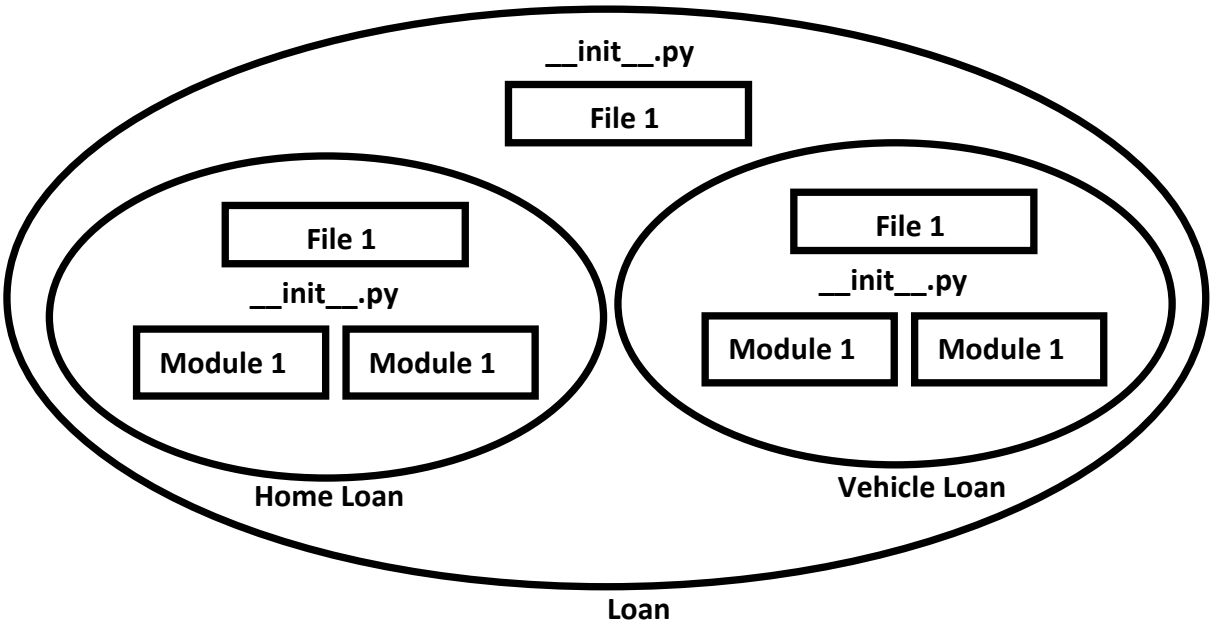
It is an encapsulation mechanism to group related modules into a single unit.
package is nothing but folder or directory which represents collection of Python modules.

Any folder or directory contains __init__.py file,is considered as a Python package.This file can be empty.

A package can contains sub packages also.



DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com



The main advantages of package statement are

1. We can resolve naming conflicts
2. We can identify our components uniquely
3. It improves modularity of the application

Eg 1:

```
D:\Python_classes>
    |-test.py
    |-pack1
        |-module1.py
        |-__init__.py
```

__init__.py:

empty file

module1.py:

```
def f1():
        print("Hello this is from module1 present in pack1")
```

test.py (version-1):

```
import pack1.module1
pack1.module1.f1()
```

test.py (version-2):

```
from pack1.module1 import f1
f1()
```

Eg 2:

```
D:\Python_classes>
    |-test.py
    |-com
        |-module1.py
        |-__init__.py
            |-durgasoft
                |-module2.py
                |-__init__.py
```

__init__.py:

empty file

module1.py:

```
def f1():
        print("Hello this is from module1 present in com")
```

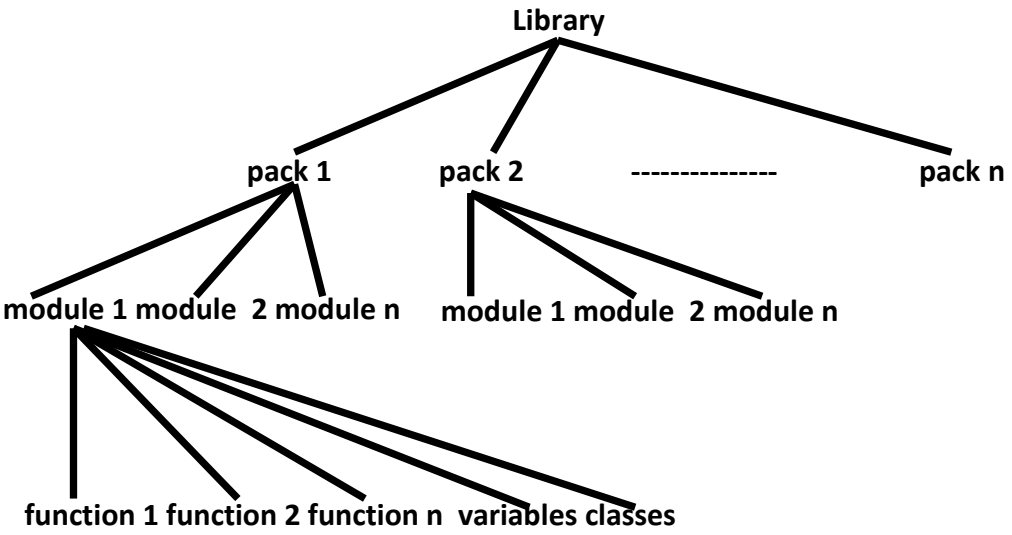
module2.py:

```
def f2():
        print("Hello this is from module2 present in com.durgasoft")
```

test.py:

```
1. from com.module1 import f1
2. from com.durgasoft.module2 import f2
3. f1()
4. f2()
5.
6. Output
7. D:\Python_classes>py test.py
8. Hello this is from module1 present in com
9. Hello this is from module2 present in com.durgasoft
```

Note: Summary diagram of library,packages,modules which contains functions,classes and variables.


Library
pack 1
pack 2
----
pack n
module 1 module 2 module n
module 1 module 2 module n
function 1 function 2 function n variables classes

Exception Handling

In any programming language there are 2 types of errors are possible.

1. Syntax Errors
2. Runtime Errors

1. Syntax Errors:

The errors which occurs because of invalid syntax are called syntax errors.

Eg 1:

```
x=10
if x==10
  print("Hello")
```

SyntaxError: invalid syntax

Eg 2:

```
print "Hello"
```

SyntaxError: Missing parentheses in call to 'print'

Note:

Programmer is responsible to correct these syntax errors. Once all syntax errors are corrected then only program execution will be started.

2. Runtime Errors:

Also known as exceptions.
While executing the program if something goes wrong because of end user input or programming logic or memory problems etc then we will get Runtime Errors.

Eg: print(10/0) ==>ZeroDivisionError: division by zero

print(10/"ten") ==>TypeError: unsupported operand type(s) for /: 'int' and 'str'

```
x=int(input("Enter Number:"))
print(x)
```

```
D:\Python_classes>py test.py
```

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

Note: Exception Handling concept applicable for Runtime Errors but not for syntax errors

What is Exception:

An unwanted and unexpected event that disturbs normal flow of program is called exception.

Eg:

ZeroDivisionError
TypeError
ValueError
FileNotFoundError
EOFError
SleepingError
TyrePuncturedError

It is highly recommended to handle exceptions. The main objective of exception handling is Graceful Termination of the program(i.e we should not block our resources and we should not miss anything)

Exception handling does not mean repairing exception. We have to define alternative way to continue rest of the program normally.

Eg:

For example our programming requirement is reading data from remote file locating at London. At runtime if london file is not available then the program should not be terminated abnormally. We have to provide local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

try:

```
   read data from remote file locating at london
except FileNotFoundError:
   use local file and continue rest of the program  normally
```

Q. What is an Exception?
Q. What is the purpose of Exception Handling?
Q. What is the meaning of Exception Handling?

Default Exception Handing in Python:

Every exception in Python is an object. For every exception type the corresponding classes are available.

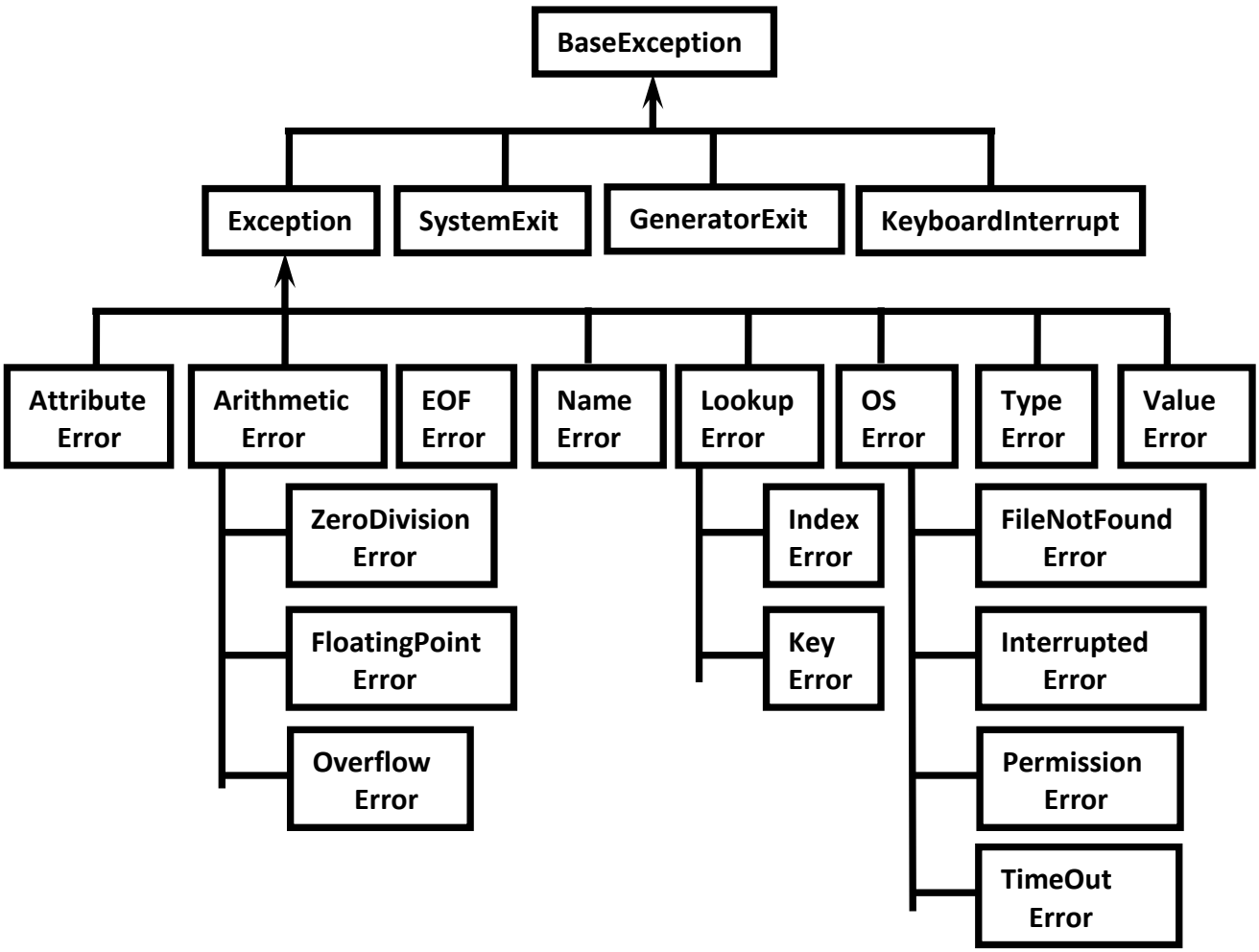
Whevever an exception occurs PVM will create the corresponding exception object and will check for handling code. If handling code is not available then Python interpreter terminates the program abnormally and prints corresponding exception information to the console.
The rest of the program won't be executed.

Eg:

```
1)  print("Hello")
2)  print(10/0)
3)  print("Hi")
4)
5)  D:\Python_classes>py test.py
6)  Hello
7)  Traceback (most recent call last):
8)    File "test.py", line 2, in <module>
9)      print(10/0)
10) ZeroDivisionError: division by zero
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Python's Exception Hierarchy



Every Exception in Python is a class.
All exception classes are child classes of BaseException.i.e every exception class extends BaseException either directly or indirectly. Hence BaseException acts as root for Python Exception Hierarchy.

Most of the times being a programmer we have to concentrate Exception and its child classes.

Customized Exception Handling by using try-except:

It is highly recommended to handle exceptions.
The code which may raise exception is called risky code and we have to take risky code inside try block. The corresponding handling code we have to take inside except block.

try:
  Risky Code
except XXX:
  Handling code/Alternative Code

without try-except:

```
1. print("stmt-1")
2. print(10/0)
3. print("stmt-3")
4.
5. Output
6. stmt-1
7. ZeroDivisionError: division by zero
```

Abnormal termination/Non-Graceful Termination

with try-except:

```
1. print("stmt-1")
2. try:
3.     print(10/0)
4. except ZeroDivisionError:
5.     print(10/2)
6. print("stmt-3")
7.
8. Output
9. stmt-1
10. 5.0
11. stmt-3
```

Normal termination/Graceful Termination

Control Flow in try-except:

```
try:
        stmt-1
        stmt-2
        stmt-3
except XXX:
      stmt-4
stmt-5
```

case-1: If there is no exception
  1,2,3,5 and Normal Termination

case-2: If an exception raised at stmt-2 and corresponding except block matched
1,4,5 Normal Termination

case-3: If an exception raised at stmt-2 and corresponding except block not matched
1, Abnormal Termination

case-4: If an exception raised at stmt-4 or at stmt-5 then it is always abnormal termination.

Conclusions:

1. within the try block if anywhere exception raised then rest of the try block wont be executed eventhough we handled that exception. Hence we have to take only risky code inside try block and length of the try block should be as less as possible.

2. In addition to try block,there may be a chance of raising exceptions inside except and finally blocks also.

3. If any statement which is not part of try block raises an exception then it is always abnormal termination.

How to print exception information:

try:

```
1. print(10/0)
2. except ZeroDivisionError as msg:
3.     print("exception raised and its description is:",msg)
4.
5. Output exception raised and its description is: division by zero
```

try with multiple except blocks:

The way of handling exception is varied from exception to exception.Hence for every exception type a seperate except block we have to provide. i.e try with multiple except blocks is possible and recommended to use.

Eg:

```
try:
    -------
    -------
    -------
except ZeroDivisionError:
    perform alternative
    arithmetic operations
```

except FileNotFoundError:
    use local file instead of remote file

If try with multiple except blocks available then based on raised exception the corresponding except block will be executed.

Eg:

```
1)  try:
2)     x=int(input("Enter First Number: "))
3)     y=int(input("Enter Second Number: "))
4)     print(x/y)
5)  except ZeroDivisionError :
6)     print("Can't Divide with Zero")
7)  except ValueError:
8)     print("please provide int value only")
9)
10) D:\Python_classes>py test.py
11) Enter First Number: 10
12) Enter Second Number: 2
13) 5.0
14)
15) D:\Python_classes>py test.py
16) Enter First Number: 10
17) Enter Second Number: 0
18) Can't Divide with Zero
19)
20) D:\Python_classes>py test.py
21) Enter First Number: 10
22) Enter Second Number: ten
23) please provide int value only
```

If try with multiple except blocks available then the order of these except blocks is important .Python interpreter will always consider from top to bottom until matched except block identified.

Eg:

```
1)  try:
2)     x=int(input("Enter First Number: "))
3)     y=int(input("Enter Second Number: "))
4)     print(x/y)
5)  except ArithmeticError :
6)     print("ArithmeticError")
7)  except ZeroDivisionError:
8)     print("ZeroDivisionError")
9)
10) D:\Python_classes>py test.py
```

```
11) Enter First Number: 10
12) Enter Second Number: 0
13) ArithmeticError
```

Single except block that can handle multiple exceptions:

We can write a single except block that can handle multiple different types of exceptions.

except (Exception1,Exception2,exception3,..): or
except (Exception1,Exception2,exception3,..) as msg :

Parenthesis are mandatory and this group of exceptions internally considered as tuple.

Eg:

```
1)  try:
2)      x=int(input("Enter First Number: "))
3)      y=int(input("Enter Second Number: "))
4)      print(x/y)
5)  except (ZeroDivisionError,ValueError) as msg:
6)      print("Plz Provide valid numbers only and problem is: ",msg)
7)
8)  D:\Python_classes>py test.py
9)  Enter First Number: 10
10) Enter Second Number: 0
11) Plz Provide valid numbers only and problem is:  division by zero
12)
13) D:\Python_classes>py test.py
14) Enter First Number: 10
15) Enter Second Number: ten
16) Plz Provide valid numbers only and problem is:  invalid literal for int() with b
17) ase 10: 'ten'
```

Default except block:

We can use default except block to handle any type of exceptions.
In default except block generally we can print normal error messages.

Syntax:

```
except:
    statements
```

Eg:

```
1)  try:
2)      x=int(input("Enter First Number: "))
3)      y=int(input("Enter Second Number: "))
4)      print(x/y)
```

```
except ZeroDivisionError:
    print("ZeroDivisionError:Can't divide with zero")
except:
    print("Default Except:Plz provide valid input only")

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: 0
ZeroDivisionError:Can't divide with zero

D:\Python_classes>py test.py
Enter First Number: 10
Enter Second Number: ten
Default Except:Plz provide valid input only
```

***Note: If try with multiple except blocks available then default except block should be last,otherwise we will get SyntaxError.

Eg:

```
try:
    print(10/0)
except:
    print("Default Except")
except ZeroDivisionError:
    print("ZeroDivisionError")

SyntaxError: default 'except:' must be last
```

Note:

The following are various possible combinations of except blocks

1. except ZeroDivisionError:
1. except ZeroDivisionError as msg:
3. except (ZeroDivisionError,ValueError) :
4. except (ZeroDivisionError,ValueError) as msg:
5. except :

finally block:

1. It is not recommended to maintain clean up code(Resource Deallocating Code or Resource Releasing code) inside try block because there is no guarentee for the execution of every statement inside try block always.

2. It is not recommended to maintain clean up code inside except block, because if there is no exception then except block won't be executed.

Hence we required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. Such type of best place is nothing but finally block.

Hence the main purpose of finally block is to maintain clean up code.

```
try:
    Risky Code
except:
    Handling Code
finally:
    Cleanup code
```

The speciality of finally block is it will be executed always whether exception raised or not raised and whether exception handled or not handled.

Case-1: If there is no exception

```
1)  try:
2)      print("try")
3)  except:
4)      print("except")
5)  finally:
6)      print("finally")
7)
8)  Output
9)  try
10) finally
```

Case-2: If there is an exception raised but handled:

```
1)  try:
2)      print("try")
3)      print(10/0)
4)  except ZeroDivisionError:
5)      print("except")
6)  finally:
7)      print("finally")
8)
9)  Output
10) try
11) except
12) finally
```

Case-3: If there is an exception raised but not handled:

```
1)  try:
2)      print("try")
3)      print(10/0)
4)  except NameError:
5)      print("except")
6)  finally:
7)      print("finally")
8)
9)  Output
10) try
11) finally
12) ZeroDivisionError: division by zero(Abnormal Termination)
```

*** **Note:** There is only one situation where finally block won't be executed ie whenever we are using os._exit(0) function.

Whenever we are using os._exit(0) function then Python Virtual Machine itself will be shutdown.In this particular case finally won't be executed.

```
1)  imports
2)  try:
3)      print("try")
4)      os._exit(0)
5)  except NameError:
6)      print("except")
7)  finally:
8)      print("finally")
9)
10) Output
11) try
```

Note:

os._exit(0)

where 0 represents status code and it indicates normal termination

There are multiple status codes are possible.

Control flow in try-except-finally:

```
try:
   stmt-1
   stmt-2
   stmt-3
except:
  stmt-4
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

finally:
  stmt-5
stmt6

Case-1: If there is no exception
1,2,3,5,6 Normal Termination

Case-2: If an exception raised at stmt2 and the corresponding except block matched
1,4,5,6 Normal Termination

Case-3: If an exception raised at stmt2 but the corresponding except block not matched
1,5 Abnormal Termination

Case-4:If an exception raised at stmt4 then it is always abnormal termination but before that finally block will be executed.

Case-5: If an exception raised at stmt-5 or at stmt-6 then it is always abnormal termination

Nested try-except-finally blocks:

We can take try-except-finally blocks inside try or except or finally blocks.i.e nesting of try-except-finally is possible.

```
try:
    ----------
    ----------
    ----------
    try:
        -------------
            ---------------
            ---------------
  except:
        ---------------
        ---------------
        ---------------
  ------------
except:
      ------------
      ------------
      ------------
```

General Risky code we have to take inside outer try block and too much risky code we have to take inside inner try block. Inside Inner try block if an exception raised then inner

except block is responsible to handle. If it is unable to handle then outer except block is responsible to handle.

Eg:

```
1)  try:
2)      print("outer try block")
3)      try:
4)          print("Inner try block")
5)          print(10/0)
6)      except ZeroDivisionError:
7)          print("Inner except block")
8)      finally:
9)          print("Inner finally block")
10) except:
11)     print("outer except block")
12) finally:
13)     print("outer finally block")
14)
15) Output
16) outer try block
17) Inner try block
18) Inner except block
19) Inner finally block
20) outer finally block
```

control flow in nested try-except-finally:

```
try:
        stmt-1
        stmt-2
        stmt-3
        try:
                stmt-4
                stmt-5
                stmt-6
        except X:
                stmt-7
        finally:
                stmt-8
        stmt-9
except Y:
        stmt-10
finally:
        stmt-11
stmt-12
```

case-1: If there is no exception
1,2,3,4,5,6,8,9,11,12 Normal Termination

case-2: If an exception raised at stmt-2 and the corresponding except block matched
1,10,11,12 Normal Termination

case-3: If an exceptiion raised at stmt-2 and the corresponding except block not matched
1,11,Abnormal Termination

case-4: If an exception raised at stmt-5 and inner except block matched
1,2,3,4,7,8,9,11,12 Normal Termination

case-5: If an exception raised at stmt-5 and inner except block not matched but outer except block matched

1,2,3,4,8,10,11,12,Normal Termination

case-6: If an exception raised at stmt-5 and both inner and outer except blocks are not matched

1,2,3,4,8,11,Abnormal Termination

case-7: If an exception raised at stmt-7 and corresponding except block matched
1,2,3,.,.,.,8,10,11,12,Normal Termination

case-8: If an exception raised at stmt-7 and corresponding except block not matched
1,2,3,.,.,.,8,11,Abnormal Termination

case-9: If an exception raised at stmt-8 and corresponding except block matched
1,2,3,.,.,.,.,10,11,12 Normal Termination

case-10: If an exception raised at stmt-8 and corresponding except block not matched
1,2,3,.,.,.,.,11,Abnormal Termination

case-11: If an exception raised at stmt-9 and corresponding except block matched
1,2,3,.,.,.,.,8,10,11,12,Normal Termination

case-12: If an exception raised at stmt-9 and corresponding except block not matched
1,2,3,.,.,.,.,8,11,Abnormal Termination

case-13: If an exception raised at stmt-10 then it is always abonormal termination but before abnormal termination finally block(stmt-11) will be executed.

case-14: If an exception raised at stmt-11 or stmt-12 then it is always abnormal termination.

Note: If the control entered into try block then compulsary finally block will be executed. If the control not entered into try block then finally block won't be executed.

else block with try-except-finally:

We can use else block with try-except-finally blocks.
else block will be executed if and only if there are no exceptions inside try block.

```
try:
        Risky Code
except:
        will be executed if exception inside try
else:
        will be executed if there is no exception inside try
finally:
        will be executed whether exception raised or not raised and handled or not
handled
```

Eg:

```
try:
        print("try")
        print(10/0)--->1
except:
        print("except")
else:
        print("else")
finally:
        print("finally")
```

If we comment line-1 then else block will be executed b'z there is no exception inside try. In this case the output is:

```
try
else
finally
```

If we are not commenting line-1 then else block won't be executed b'z there is exception inside try block. In this case output is:

try
except
finally

Various possible combinations of try-except-else-finally:

1. Whenever we are writing try block,compulsory we should write except or finally block.i.e without except or finally block we cannot write try block.

2. Wheneever we are writing except block,compulsory we should write try block. i.e except without try is always invalid.

3. Whenever we are writing finally block,compulsory we should write try block. i.e finally without try is always invalid.

4.We can write multiple except blocks for the same try,but we cannot write multiple finally blocks for the same try

5. Whenever we are writing else block compulsory except block should be there. i.e without except we cannot write else block.

6. In try-except-else-finally order is important.

7. We can define try-except-else-finally inside try,except,else and finally blocks. i.e nesting of try-except-else-finally is always possible.

Total 23 combinations
1. try:

Types of Exceptions:

In Python there are 2 types of exceptions are possible.
1. Predefined Exceptions
2. User Definded Exceptions

1. Predefined Exceptions:

Also known as Inbuilt exceptions

The exceptions which are raised automatically by Python virtual machine whenver a particular event occurs ,are called pre defined exceptions.

Eg 1: Whenever we are trying to perform Division by zero,automatically Python will raise ZeroDivisionError.

```
print(10/0)
```

Eg 2: Whenever we are trying to convert input value to int type and if input value is not int value then Python will raise ValueError automatically

```
x=int("ten")===>ValueError
```

2. User Defined Exceptions:

Also known as Customized Exceptions or Programatic Exceptions

Some time we have to define and raise exceptions explicitly to indicate that something goes wrong ,such type of exceptions are called User Defined Exceptions or Customized Exceptions

Programmer is responsible to define these exceptions and Python not having any idea about these. Hence we have to raise explicitly based on our requirement by using "raise" keyword.

Eg:

InSufficientFundsException
InvalidInputException
TooYoungException
TooOldException

How to Define and Raise Customized Exceptions:

Every exception in Python is a class that extends Exception class either directly or indirectly.

Syntax:

```
class classname(predefined exception class name):
    def __init__(self,arg):
        self.msg=arg
```

Eg:

```
1)  class TooYoungException(Exception):
2)      def __init__(self,arg):
3)          self.msg=arg
```

TooYoungException is our class name which is the child class of Exception

We can raise exception by using raise keyword as follows
raise TooYoungException("message")

Eg:

```
1)  class TooYoungException(Exception):
2)      def __init__(self,arg):
3)          self.msg=arg
4)
5)  class TooOldException(Exception):
6)      def __init__(self,arg):
7)          self.msg=arg
8)
9)  age=int(input("Enter Age:"))
10) if age>60:
11)     raise TooYoungException("Plz wait some more time you will get best match soon!!!")
12) elif age<18:
13)     raise TooOldException("Your age already crossed marriage age...no chance of getting ma
    rriage")
14) else:
15)     print("You will get match details soon by email!!!")
16)
17) D:\Python_classes>py test.py
18) Enter Age:90
19) __main__.TooYoungException: Plz wait some more time you will get best match soon!!!
20)
21) D:\Python_classes>py test.py
22) Enter Age:12
23) __main__.TooOldException: Your age already crossed marriage age...no chance of g
24) etting marriage
25)
26) D:\Python_classes>py test.py
27) Enter Age:27
28) You will get match details soon by email!!!
```

Note:

raise keyword is best suitable for customized exceptions but not for pre defined exceptions

Logging the Exceptions:

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:

1. We can use log files while performing debugging
2. We can provide statistics like number of requests per day etc

To implement logging,Python provides one inbuilt module logging.

logging levels:

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 information
6. NOTSET==>0==>Rrepresents that the level is not set.

By default while executing Python program only WARNING and higher level messages will be displayed.

How to implement logging:

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)
```

Q.Write a Python program to create a log file and write WARNING and higher level messages?

```
1) import logging
2) logging.basicConfig(filename='log.txt',level=logging.WARNING)
3) print("Logging Module Demo")
4) logging.debug("This is debug message")
5) logging.info("This is info message")
6) logging.warning("This is warning message")
7) logging.error("This is error message")
8) logging.critical("This is critical message")
```

log.txt:

```
1) WARNING:root:This is warning message
2) ERROR:root:This is error message
3) CRITICAL:root: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.

```
1) import logging
2) logging.basicConfig(filename='log.txt',level=logging.DEBUG)
3) print("Logging Module Demo")
4) logging.debug("This is debug message")
5) logging.info("This is info message")
6) logging.warning("This is warning message")
7) logging.error("This is error message")
8) logging.critical("This is critical message")
```

log.txt:

```
1) DEBUG:root:This is debug message
2) INFO:root:This is info message
3) WARNING:root:This is warning message
4) ERROR:root:This is error message
5) CRITICAL:root:This is critical message
```

Note: **We can format log messages to include date and time, ip address of the client etc at advanced level.**

How to write Python program exceptions to the log file:

By using the following function we can write exceptions information to the log file.

logging.exception(msg)

Q. Python Program to write exception information to the log file

```
1)  import logging
2)  logging.basicConfig(filename='mylog.txt',level=logging.INFO)
3)  logging.info("A New request Came:")
4)  try:
5)     x=int(input("Enter First Number: "))
6)     y=int(input("Enter Second Number: "))
7)     print(x/y)
8)  except ZeroDivisionError as msg:
9)     print("cannot divide with zero")
10)    logging.exception(msg)
11) except ValueError as msg:
12)    print("Enter only int values")
13)    logging.exception(msg)
14) logging.info("Request Processing Completed")
15)
16)
17) D:\Python_classes>py test.py
18) Enter First Number: 10
19) Enter Second Number: 2
20) 5.0
21)
22) D:\Python_classes>py test.py
23) Enter First Number: 10
24) Enter Second Number: 0
25) cannot divide with zero
26)
27) D:\Python_classes>py test.py
28) Enter First Number: 10
29) Enter Second Number: ten
30) Enter only int values
```

mylog.txt:

```
1)  INFO:root:A New request Came:
2)  INFO:root:Request Processing Completed
3)  INFO:root:A New request Came:
4)  ERROR:root:division by zero
5)  Traceback (most recent call last):
6)    File "test.py", line 7, in <module>
7)      print(x/y)
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

```
8)  ZeroDivisionError: division by zero
9)  INFO:root:Request Processing Completed
10) INFO:root:A New request Came:
11) ERROR:root:invalid literal for int() with base 10: 'ten'
12) Traceback (most recent call last):
13)   File "test.py", line 6, in <module>
14)    y=int(input("Enter Second Number: "))
15) ValueError: invalid literal for int() with base 10: 'ten'
16) INFO:root:Request Processing Completed
```

Debugging Python Program by using assert keyword:

The process of identifying and fixing the bug is called debugging.
Very common way of debugging is to use print() statement. But the problem with the print() statement is after fixing the bug,compulsory we have to delete the extra added print() statments,otherwise these will be executed at runtime which creates performance problems and disturbs console output.

To overcome this problem we should go for assert statement. The main advantage of assert statement over print() statement is after fixing bug we are not required to delete assert statements. Based on our requirement we can enable or disable assert statements.

Hence the main purpose of assertions is to perform debugging. Usully we can perform debugging either in development or in test environments but not in production environment. Hence assertions concept is applicable only for dev and test environments but not for production environment.

Types of assert statements:

There are 2 types of assert statements

1. Simple Version
2. Augmented Version

1. Simple Version:

assert conditional_expression

2. Augmented Version:

assert conditional_expression,message

conditional_expression will be evaluated and if it is true then the program will be continued.
If it is false then the program will be terminated by raising AssertionError.

By seeing AssertionError,programmer can analyze the code and can fix the problem.

Eg:

```
1)  def squareIt(x):
2)      return x**x
3)  assert squareIt(2)==4,"The square of 2 should be 4"
4)  assert squareIt(3)==9,"The square of 3 should be 9"
5)  assert squareIt(4)==16,"The square of 4 should be 16"
6)  print(squareIt(2))
7)  print(squareIt(3))
8)  print(squareIt(4))
9)
10) D:\Python_classes>py test.py
11) Traceback (most recent call last):
12)   File "test.py", line 4, in <module>
13)     assert squareIt(3)==9,"The square of 3 should be 9"
14) AssertionError: The square of 3 should be 9
15)
16) def squareIt(x):
17)     return x*x
18) assert squareIt(2)==4,"The square of 2 should be 4"
19) assert squareIt(3)==9,"The square of 3 should be 9"
20) assert squareIt(4)==16,"The square of 4 should be 16"
21) print(squareIt(2))
22) print(squareIt(3))
23) print(squareIt(4))
24)
25) Output
26) 4
27) 9
28) 16
```

Exception Handling vs assertions:

Assertions concept can be used to alert programmer to resolve development time errors.

Exception Handling can be used to handle runtime errors.

File Handling:

As the part of programming requirement,we have to store our data permanently for future purpose. For this requirement we should go for files.

Files are very common permanent storage areas to store our data.

Types of Files:

There are 2 types of files

1. Text Files:

Usually we can use text files to store character data
eg: abc.txt

2. Binary Files:

Usually we can use binary files to store binary data like images,video files, audio files etc...

Opening a File:

Before performing any operation (like read or write) on the file,first we have to open that file.For this we should use Python's inbuilt function open()

But at the time of open,we have to specify mode,which represents the purpose of opening file.

f=open(filename,mode)

The allowed modes in Python are

1. r--->open an existing file for read operation. The file pointer is positioned at the beginning of the file.If the specified file does not exist then we will get FileNotFoundError.This is default mode.

2. w--->open an existing file for write operation. If the file already contains some data then it will be overridden. If the specified file is not already avaialble then this mode will create that file.

3. a--->open an existing file for append operation. It won't override existing data.If the specified file is not already avaialble then this mode will create a new file.

4. r+ --->To read and write data into the file. The previous data in the file will not be deleted.The file pointer is placed at the beginning of the file.

5. w+--->To write and read data. It will override existing data.

6. a+--->To append and read data from the file.It wont override existing data.

7. x--->To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.

Note: All the above modes are applicable for text files. If the above modes suffixed with 'b' then these represents for binary files.

Eg: rb,wb,ab,r+b,w+b,a+b,xb

```
f=open("abc.txt","w")
```

We are opening abc.txt file for writing data.

Closing a File:

After completing our operations on the file,it is highly recommended to close the file. For this we have to use close() function.

```
f.close()
```

Various properties of File Object:

Once we opend a file and we got file object,we can get various details related to that file by using its properties.

name--->Name of opened file
mode---->Mode in which the file is opened
closed---->returns boolean value indicates that file is closed or not
readable()--->Retruns boolean value indicates that whether file is readable or not
writable()----->Returns boolean value indicates that whether file is writable or not.

Eg:

```
1)  f=open("abc.txt",'w')
2)  print("File Name: ",f.name)
3)  print("File Mode: ",f.mode)
4)  print("Is File Readable:  ",f.readable())
5)  print("Is File Writable:  ",f.writable())
6)  print("Is File Closed : ",f.closed)
7)  f.close()
8)  print("Is File Closed : ",f.closed)
9)
10)
11) Output
12) D:\Python_classes>py test.py
13) File Name:  abc.txt
```

```
14) File Mode:  w
15) Is File Readable:   False
16) Is File Writable:   True
17) Is File Closed :  False
18) Is File Closed :  True
```

Writing data to text files:

We can write character data to the text files by using the following 2 methods.

write(str)
writelines(list of lines)

Eg:

```
1. f=open("abcd.txt",'w')
2. f.write("Durga\n")
3. f.write("Software\n")
4. f.write("Solutions\n")
5. print("Data written to the file successfully")
6. f.close()
```

abcd.txt:

Durga
Software
Solutions

Note: In the above program,data present in the file will be overridden everytime if we run the program. Instead of overriding if we want append operation then we should open the file as follows.

```
f=open("abcd.txt","a")
```

Eg 2:

```
1) f=open("abcd.txt",'w')
2) list=["sunny\n","bunny\n","vinny\n","chinny"]
3) f.writelines(list)
4) print("List of lines written to the file successfully")
5) f.close()
```

abcd.txt:

sunny
bunny
vinny

chinny

Note: while writing data by using write() methods, compulsory we have to provide line seperator(\n),otherwise total data should be written to a single line.

Reading Character Data from text files:

We can read character data from text file by using the following read methods.

read()===>To read total data from the file
read(n)===>To read 'n' characters from the file
readline()==>To read only one line
readlines()==>To read all lines into a list

Eg 1: To read total data from the file

```
1)  f=open("abc.txt",'r')
2)  data=f.read()
3)  print(data)
4)  f.close()
5)
6)  Output
7)  sunny
8)  bunny
9)  chinny
10) vinny
```

Eg 2: To read only first 10 characters:

```
1)  f=open("abc.txt",'r')
2)  data=f.read(10)
3)  print(data)
4)  f.close()
5)
6)  Output
7)  sunny
8)  bunn
```

Eg 3: To read data line by line:

```
1)  f=open("abc.txt",'r')
2)  line1=f.readline()
3)  print(line1,end='')
4)  line2=f.readline()
5)  print(line2,end='')
6)  line3=f.readline()
7)  print(line3,end='')
```

```
f.close()

Output
sunny
bunny
chinny
```

Eg 4: To read all lines into list:

```
f=open("abc.txt",'r')
lines=f.readlines()
for line in lines:
    print(line,end='')
f.close()

Output
sunny
bunny
chinny
vinny
```

Eg 5:

```
f=open("abc.txt","r")
print(f.read(3))
print(f.readline())
print(f.read(4))
print("Remaining data")
print(f.read())

Output
sun
ny

bunn
Remaining data
y
chinny
vinny
```

The with statement:

The with statement can be used while opening a file.We can use this to group file operation statements within a block.
The advantage of with statement is it will take care closing of file,after completing all operations automatically even in the case of exceptions also, and we are not required to close explicitly.

Eg:

```
1)  with open("abc.txt","w") as f:
2)      f.write("Durga\n")
3)      f.write("Software\n")
4)      f.write("Solutions\n")
5)      print("Is File Closed: ",f.closed)
6)  print("Is File Closed: ",f.closed)
7)
8)  Output
9)  Is File Closed:  False
10) Is File Closed:  True
```

The seek() and tell() methods:

tell():

==>We can use tell() method to return current position of the cursor(file pointer) from beginning of the file. [can you plese telll current cursor position]

The position(index) of first character in files is zero just like string index.

Eg:

```
1)  f=open("abc.txt","r")
2)  print(f.tell())
3)  print(f.read(2))
4)  print(f.tell())
5)  print(f.read(3))
6)  print(f.tell())
```

abc.txt:

```
sunny
bunny
chinny
vinny
```

Output:

```
0
su
2
nny
5
```

seek():

We can use seek() method to move cursor(file pointer) to specified location.
[Can you please seek the cursor to a particular location]

f.seek(offset,fromwhere)

offset represents the number of positions

The allowed values for second attribute(from where) are
0---->From beginning of file(default value)
1---->From current position
2--->From end of the file

Note: Python 2 supports all 3 values but Python 3 supports only zero.

Eg:

```
1)  data="All Students are STUPIDS"
2)  f=open("abc.txt","w")
3)  f.write(data)
4)  with open("abc.txt","r+") as f:
5)      text=f.read()
6)      print(text)
7)      print("The Current Cursor Position: ",f.tell())
8)      f.seek(17)
9)      print("The Current Cursor Position: ",f.tell())
10)     f.write("GEMS!!!")
11)     f.seek(0)
12)     text=f.read()
13)     print("Data After Modification:")
14)     print(text)
15)
16) Output
17)
18) All Students are STUPIDS
19) The Current Cursor Position:  24
20) The Current Cursor Position:  17
21) Data After Modification:
22) All Students are GEMS!!!
```

How to check a particular file exists or not?

We can use os library to get information about files in our computer.
os module has path sub module,which contains isFile() function to check whether a particular file exists or not?

os.path.isfile(fname)

Q. Write a program to check whether the given file exists or not. If it is available then print its content?

```
1)  import os,sys
2)  fname=input("Enter File Name: ")
3)  if os.path.isfile(fname):
4)      print("File exists:",fname)
5)      f=open(fname,"r")
6)  else:
7)      print("File does not exist:",fname)
8)      sys.exit(0)
9)  print("The content of file is:")
10) data=f.read()
11) print(data)
12)
13) Output
14) D:\Python_classes>py test.py
15) Enter File Name: durga.txt
16) File does not exist: durga.txt
17)
18) D:\Python_classes>py test.py
19) Enter File Name: abc.txt
20) File exists: abc.txt
21) The content of file is:
22) All Students are GEMS!!!
23) All Students are GEMS!!!
24) All Students are GEMS!!!
25) All Students are GEMS!!!
26) All Students are GEMS!!!
27) All Students are GEMS!!!
```

Note:

sys.exit(0) ===>To exit system without executing rest of the program.
argument represents status code . 0 means normal termination and it is the default value.

Q. Program to print the number of lines,words and characters present in the given file?

```
1)  import os,sys
2)  fname=input("Enter File Name: ")
3)  if os.path.isfile(fname):
4)      print("File exists:",fname)
5)      f=open(fname,"r")
6)  else:
7)      print("File does not exist:",fname)
```

```
    sys.exit(0)
lcount=wcount=ccount=0
for line in f:
    lcount=lcount+1
    ccount=ccount+len(line)
    words=line.split()
    wcount=wcount+len(words)
print("The number of Lines:",lcount)
print("The number of Words:",wcount)
print("The number of Characters:",ccount)

Output
D:\Python_classes>py test.py
Enter File Name: durga.txt
File does not exist: durga.txt

D:\Python_classes>py test.py
Enter File Name: abc.txt
File exists: abc.txt
The number of Lines: 6
The number of Words: 24
The number of Characters: 149
```

abc.txt:

All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!
All Students are GEMS!!!

Handling Binary Data:

It is very common requirement to read or write binary data like images,video files,audio files etc.

Q. Program to read image file and write to a new image file?

```
f1=open("rossum.jpg","rb")
f2=open("newpic.jpg","wb")
bytes=f1.read()
f2.write(bytes)
print("New Image is available with the name: newpic.jpg")
```

Handling csv files:

CSV==>Comma seperated values

As the part of programming,it is very common requirement to write and read data wrt csv files. Python provides csv module to handle csv files.

Writing data to csv file:

```
1)  import csv
2)  with open("emp.csv","w",newline='') as f:
3)      w=csv.writer(f)  # returns csv writer object
4)      w.writerow(["ENO","ENAME","ESAL","EADDR"])
5)      n=int(input("Enter Number of Employees:"))
6)      for i in range(n):
7)          eno=input("Enter Employee No:")
8)          ename=input("Enter Employee Name:")
9)          esal=input("Enter Employee Salary:")
10)         eaddr=input("Enter Employee Address:")
11)         w.writerow([eno,ename,esal,eaddr])
12) print("Total Employees data written to csv file successfully")
```

Note: Observe the difference with newline attribute and without

with open("emp.csv","w",newline='') as f:
with open("emp.csv","w") as f:

Note: If we are not using newline attribute then in the csv file blank lines will be included between data. To prevent these blank lines, newline attribute is required in Python-3,but in Python-2 just we can specify mode as 'wb' and we are not required to use newline attribute.

Reading Data from csv file:

```
1)  import csv
2)  f=open("emp.csv",'r')
3)  r=csv.reader(f) #returns csv reader object
4)  data=list(r)
5)  #print(data)
6)  for line in data:
7)      for word in line:
8)          print(word,"\t",end='')
9)      print()
10)
11) Output
12) D:\Python_classes>py test.py
```

```
ENO  ENAME  ESAL  EADDR
100  Durga  1000  Hyd
200  Sachin 2000  Mumbai
300  Dhoni  3000  Ranchi
```

Zipping and Unzipping Files:

It is very common requirement to zip and unzip files.
The main advantages are:

1. To improve memory utilization
2. We can reduce transport time
3. We can improve performance.

To perform zip and unzip operations, Python contains one in-bulit module zip file.
This module contains a class : ZipFile

To create Zip file:

We have to create ZipFile class object with name of the zip file,mode and constant ZIP_DEFLATED. This constant represents we are creating zip file.

f=ZipFile("files.zip","w","ZIP_DEFLATED")

Once we create ZipFile object,we can add files by using write() method.

f.write(filename)

Eg:

```
from zipfile import *
f=ZipFile("files.zip",'w',ZIP_DEFLATED)
f.write("file1.txt")
f.write("file2.txt")
f.write("file3.txt")
f.close()
print("files.zip file created successfully")
```

To perform unzip operation:

We have to create ZipFile object as follows

f=ZipFile("files.zip","r",ZIP_STORED)

ZIP_STORED represents unzip operation. This is default value and hence we are not required to specify.

Once we created ZipFile object for unzip operation,we can get all file names present in that zip file by using namelist() method.

names =f.namelist()

Eg:

```
1)  from zipfile import *
2)  f=ZipFile("files.zip",'r',ZIP_STORED)
3)  names=f.namelist()
4)  for name in names:
5)      print(   "File Name: ",name)
6)      print("The Content of this  file is:")
7)      f1=open(name,'r')
8)      print(f1.read())
9)      print()
```

Working with Directories:

It is very common requirement to perform operations for directories like

1. To know current working directory
2. To create a new directory
3. To remove an existing directory
4. To rename a directory
5. To list contents of the directory
etc...

To perform these operations,Python provides inbuilt module os,which contains several functions to perform directory related operations.

Q1. To Know Current Working Directory:

```
import os
cwd=os.getcwd()
print("Current Working Directory:",cwd)
```

Q2. To create a sub directory in the current working directory:

```
import os
os.mkdir("mysub")
print("mysub directory created in cwd")
```

Q3. To create a sub directory in mysub directory:

```
cwd
  |-mysub
      |-mysub2
```

```
import os
os.mkdir("mysub/mysub2")
print("mysub2  created inside mysub")
```

Note: Assume mysub already present in cwd.

Q4. To create multiple directories like sub1 in that sub2 in that sub3:

```
import os
os.makedirs("sub1/sub2/sub3")
print("sub1 and in that sub2 and in that sub3 directories created")
```

Q5. To remove a directory:

```
import os
os.rmdir("mysub/mysub2")
print("mysub2 directory deleted")
```

Q6. To remove multiple directories in the path:

```
import os
os.removedirs("sub1/sub2/sub3")
print("All 3 directories sub1,sub2 and sub3 removed")
```

Q7. To rename a directory:

```
import os
os.rename("mysub","newdir")
print("mysub directory renamed to newdir")
```

Q8. To know contents of directory:

os module provides listdir() to list out the contents of the specified directory. It won't display the contents of sub directory.

Eg:

```
1)  import os
2)  print(os.listdir("."))
3)
4)  Output
5)  D:\Python_classes>py test.py
6)  ['abc.py', 'abc.txt', 'abcd.txt', 'com', 'demo.py', 'durgamath.py', 'emp.csv', '
7)  file1.txt', 'file2.txt', 'file3.txt', 'files.zip', 'log.txt', 'module1.py', 'myl
8)  og.txt', 'newdir', 'newpic.jpg', 'pack1', 'rossum.jpg', 'test.py', '__pycache__'
9)  ]
```

The above program display contents of current working directory but not contents of sub directories.

If we want the contents of a directory including sub directories then we should go for walk() function.

Q9. To know contents of directory including sub directories:

We have to use walk() function
[Can you please walk in the directory so that we can aware all contents of that directory]

os.walk(path,topdown=True,onerror=None,followlinks=False)

It returns an Iterator object whose contents can be displayed by using for loop

path-->Directory path. cwd means .
topdown=True --->Travel from top to bottom
onerror=None --->on error detected which function has to execute.
followlinks=True -->To visit directories pointed by symbolic links

Eg: To display all contents of Current working directory including sub directories:

```
1)  import os
2)  for dirpath,dirnames,filenames  in os.walk('.'):
3)      print("Current Directory Path:",dirpath)
4)      print("Directories:",dirnames)
5)      print("Files:",filenames)
6)      print()
7)
8)
9)   Output
10)  Current Directory Path: .
11) Directories: ['com', 'newdir', 'pack1', '__pycache__']
12) Files: ['abc.txt', 'abcd.txt', 'demo.py', 'durgamath.py', 'emp.csv', 'file1.txt'
13) , 'file2.txt', 'file3.txt', 'files.zip', 'log.txt', 'module1.py', 'mylog.txt', '
```

```
14) newpic.jpg', 'rossum.jpg', 'test.py']
15)
16) Current Directory Path: .\com
17) Directories: ['durgasoft', '__pycache__']
18) Files: ['module1.py', '__init__.py']
19)
20) ...
```

Note: To display contents of particular directory,we have to provide that directory name as argument to walk() function.

os.walk("directoryname")

Q. What is the difference between listdir() and walk() functions?

In the case of listdir(), we will get contents of specified directory but not sub directory contents. But in the case of walk() function we will get contents of specified directory and its sub directories also.

Running Other programs from Python program:

os module contains system() function to run programs and commands.
It is exactly same as system() function in C language.

os.system("commad string")
The argument is any command which is executing from DOS.

Eg:

```
import os
os.system("dir *.py")
os.system("py abc.py")
```

How to get information about a File:

We can get statistics of a file like size, last accessed time,last modified time etc by using stat() function of os module.

stats=os.stat("abc.txt")

The statistics of a file includes the following parameters:

st_mode==>Protection Bits
st_ino==>Inode number
st_dev===>device

st_nlink===>no of hard links
st_uid===>userid of owner
st_gid==>group id of owner
st_size===>size of file in bytes
st_atime==>Time of most recent access
st_mtime==>Time of Most recent modification
st_ctime==> Time of Most recent meta data change

Note:

st_atime, st_mtime and st_ctime returns the time as number of milli seconds since Jan 1st 1970 ,12:00AM. By using datetime module fromtimestamp() function,we can get exact date and time.

Q. To print all statistics of file abc.txt:

```
1) import os
2) stats=os.stat("abc.txt")
3) print(stats)
4)
5) Output
6) os.stat_result(st_mode=33206, st_ino=844424930132788, st_dev=2657980798, st_nlin
7) k=1, st_uid=0, st_gid=0, st_size=22410, st_atime=1505451446, st_mtime=1505538999
8) , st_ctime=1505451446)
```

Q. To print specified properties:

```
1) import os
2) from  datetime  import *
3) stats=os.stat("abc.txt")
4) print("File Size in Bytes:",stats.st_size)
5) print("File Last Accessed Time:",datetime.fromtimestamp(stats.st_atime))
6) print("File Last Modified Time:",datetime.fromtimestamp(stats.st_mtime))
7)
8) Output
9) File Size in Bytes: 22410
10) File Last Accessed Time: 2017-09-15 10:27:26.599490
11) File Last Modified Time: 2017-09-16 10:46:39.245394
```

Pickling and Unpickling of Objects:

Sometimes we have to write total state of object to the file and we have to read total object from the file.

The process of writing state of object to the file is called pickling and the process of reading state of an object from the file is called unpickling.

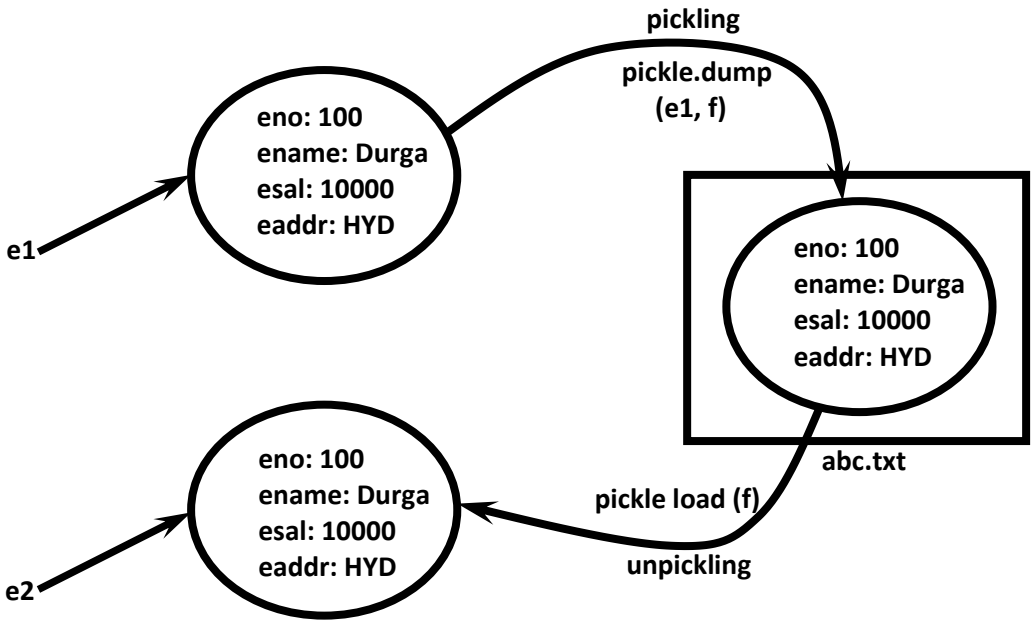
We can implement pickling and unpickling by using pickle module of Python.

pickle module contains dump() function to perform pickling.

```
pickle.dump(object,file)
```

pickle module contains load() function to perform unpickling

```
obj=pickle.load(file)
```



Writing and Reading State of object by using pickle Module:

```
1)  import pickle
2)  class Employee:
3)      def __init__(self,eno,ename,esal,eaddr):
4)          self.eno=eno;
5)          self.ename=ename;
6)          self.esal=esal;
7)          self.eaddr=eaddr;
8)      def display(self):
```

```
9)          print(self.eno,"\t",self.ename,"\t",self.esal,"\t",self.eaddr)
10) with open("emp.dat","wb") as f:
11)     e=Employee(100,"Durga",1000,"Hyd")
12)     pickle.dump(e,f)
13)     print("Pickling of Employee Object completed...")
14)
15) with open("emp.dat","rb") as f:
16)     obj=pickle.load(f)
17)     print("Printing Employee Information after unpickling")
18)     obj.display()
```

Writing Multiple Employee Objects to the file:

emp.py:

```
1)  class Employee:
2)      def __init__(self,eno,ename,esal,eaddr):
3)          self.eno=eno;
4)          self.ename=ename;
5)          self.esal=esal;
6)          self.eaddr=eaddr;
7)      def display(self):
8)
9)
10)     print(self.eno,"\t",self.ename,"\t",self.esal,"\t",self.eaddr)
```

pick.py:

```
1)  import emp,pickle
2)  f=open("emp.dat","wb")
3)  n=int(input("Enter The number of Employees:"))
4)  for i in range(n):
5)      eno=int(input("Enter Employee Number:"))
6)      ename=input("Enter Employee Name:")
7)      esal=float(input("Enter Employee Salary:"))
8)      eaddr=input("Enter Employee Address:")
9)      e=emp.Employee(eno,ename,esal,eaddr)
10)     pickle.dump(e,f)
11) print("Employee Objects pickled successfully")
```

unpick.py:

```
1)  import emp,pickle
2)  f=open("emp.dat","rb")
3)  print("Employee Details:")
4)  while True:
5)      try:
6)          obj=pickle.load(f)
```

```
7)          obj.display()
8)      except EOFError:
9)          print("All employees Completed")
10)         break
11) f.close()
```