PYTHON

by

lokesh

sathya technologies

PYTHON :

→ Python is a powerful and general purpose programming language, developed by the guido van rossum (1989)

→ guido van rossum developed python language at mathematical research institute called 'CWI'

→ CWI is located at Netherlands

→ Guido developed python language by taking the different language features like

i. procedure oriented programming language → C
ii. Object oriented programming language → C++, Java
iii. Scripting language → shell script, perl
iv. Modular programming language → modula-3

NOTE: Guido van Rossum make it available python language to the public in 1991

→ By using python language we can do the following things

1. GUI application development
2. Web application development
3. Data analytics
4. task automations
5. Test cases implementations
6. scientific application development
7. Network application development
8. Gaming / animation application development

NOTE: Most of the people are calling python is a scripting language because the way of developing python applications and execution of python applications are similar to the scripting languages.

* Differences between programming and scripting languages

9515922459

scripting language	programming language
→ scripting language are interpreter based languages	→ programming language are compiler based languages
→ scripting language programs (or) applications explicit compilation is not required	→ It requires explicit compilation
→ S.L.P (or) applications directly we can run	→ P.L. programs(or) applications we can not run without compiling
→ S.L. programs (or) applications takes longer time to execute	→ P.L. programs (or) applications takes less time to execute.
Ex: shell Script, perl, ---	Ex: C, C++, Java, onet ---

* FEATURES OF PYTHON:

python supports the following features

1. simple and easy to learn:-

→ The syntaxes of the python language are very simple
→ Anybody can remember the python language syntax's, rules and regulations very easily
→ By developing the python programs (or) applications programers need not to focus on the syntax's.
→ instead of focusing on syntax's they can focus on the business logic implementation.
→ The elegant syntax's of the python language makes the people to learn python in easiest manner.
→ without having any other programming language knowledge directly anybody can learn python language.
→ The simple and powerful syntax of the python language makes the programers to express their business logic is less lines of code
→ Because of simple feature of python language project development cost, development time and maintenance cost will become less.

2. Portable :

→ The python applications which are developed on one platform are going to execute on irrespective of platforms without making any changes inthe python applications

→ To achieve the portability feature with respect to every o.s seperate python software is developed for every version of python

3. Free & open source & Re-distribution:

→ Anybody can use the python software without purchasing license agrement of python

→ Anybody can read the python source code & they can do the modifications in the python source code and we can redistribute that code to others.

4. High-level language :

→ while developing python programers (or) developers need not to focus on the memory management and memory used by the program (Low level details)

5. supporting procedure oriented programing & object oriented programing language features:

→ python is language supports both p.o.p & o.o.p language features.

→ If we develop any application according to the oops principles then that application will get the security, Flexibility & reusability.

→ different oops principles are

1. encapsulation
2. polymorphism
3. inheritance
4. abstraction

→ python supports oops principles because of that reason python applications will get the security, Flexibility and reusability.

6. Interpreted Language:

→ python applications doesnot require explicit compilation so that compiler is not required for in python software

→ Directly we can run the python applications without compiling explicit.

→ python interpretar is responsible for execution of the python applications.

→ whenever we run the python applications python interpreter will check the syntax errors. If no syntax error python ~~imp~~ interpreter converts that code in the form of intermediate code in the form of low level format and executes it.

→ The intermediate code of the python applications is known as Byte code

→ The extension for the byte code file is '.pye (python compile code')

7. Extensible :-

→ python application execution is slower compared to C, C++ programs execution.

→ TO overcome the above problem we can implement some logics by using C, C++ language and we can use C, C++ programs into python application.

→ python source code doesnot contain security i.e., anybody can read python code and they can do the modifications in the source code

→ If we want to provide security to the some part (logic) or some algorithm of python application then we represent that code on logic (OR) algorithm by using, C (or) C++ languages and we use that code into python application.

8. Embedable :-

→ we can embedded the python code into the other language programs such as C, C++, Java ----

→ In order to provide the scripting capabilities to the other language programs / applications we use python code into those applications.

4. Extensive libraries:

→ python language is providing huge building libraries
→ python developers can use the built in libraries into their applications
→ By using builtin libraries application development will become faster
→ Third party people developed libraries, modules we can add to python software and used into python applications.

* Installation of python software:

→ Download the required python 2.7.11 software from the following website

https://www.python.org/downloads

→ click on the downloaded installer file
→ click on the run
→ click on the next
→ click on the next
→ click on the next
→ click on yes
→ click on finish

* Development of python applications (or) programs:

We can develop python application/programs in 2 modes. They are

1. interactive mode 2. Batch mode

1. Interactive mode:

→ Interactive mode is a command line shell
→ In command line shell if we write any python statement immediately that statement will execute and gives the result
→ Interactive mode is used to test the features of the python (what python can do)

NOTE: Interactive mode is not used for development of the business applications.

→ open command prompt

```
C:\user\ cd..
C:\> cd python 37 ⇒ C:\> cd python 37 > python
C:\ python
>>> 10+20
30
>>> 10/2
5
>>> i = 1000
>>> j = 2000
>>> print i+j
3000
```

2. Batchmode :-

→ In batch mode we write group of python statements in any one of the editors (or) IDE's

→ Different editors are

Notepad
Notepad++
editplus
nano
gedit
IDLE ----

→ Different IDE's are

pycharm
Eric
Eclipse
Netbeans ----

→ After writing the group of python statements in anyone of the editor/IDE we save the file with extension ".py" or ".pyw"

→ After developing the .py (or) .pyw files we submit those files to the python interpretor directly.

→ Batch mode is used for development of business applications.

→ open the notepad

→ write the following code

```
i = 1000
j = 200
print i+j
print i-j
```

→ save the above file with demo.py in c.drive python 27 folder (or) directory.

→ open the cmd prompt c:\python 27 > python demo.py ↵

```
1200
800
```

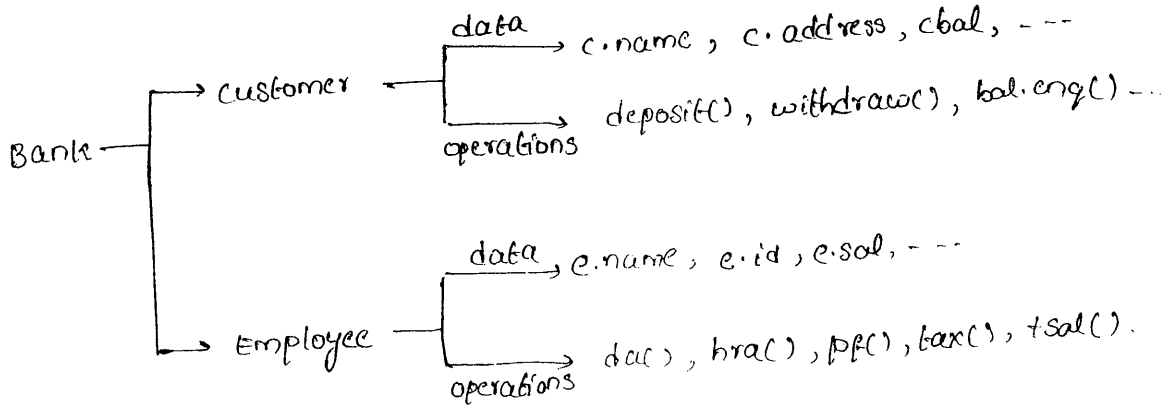
NOTE: whenever we install the python software in windows o.s along with the python software 'IDLE' or 'python GUI' editors will be installed

→ In IDLE editor we can develop the python programs in interactive and batch mode.

→ After developing the python application in idle editor directly we can run that python application by using shortcut key (F5)

→ within the IDLE editor we can perform the debugging operation of the python applications.

→ Any organisation contains 2 parts 1. data
2. operations



→ Data of the any organisation we can represent by using any program language.

→ To represent the data of organisations programming language are providing datatypes & variables.

→ Every programming language supports datatypes and variables but the datatypes & variables of one language are not going tobe same with another programming language.

→ operations of the organisations we can represent by using any programming language

→ To represent the operations of the organisations every programming language is providing functions or methods or both.

* Datatypes :-

→ Data types are nothing but some of the keywords of the programming languages, which are used to specify what type of data has tobe store into the variables.

→ without datatypes we cannot store the data into the variables

→ python supports the dynamic datatypes i.e., at the time of execution of program datatype of the variable will be desided based on the data which is assigned to that variable.

→ At the time of writing the program, programers should not specify datatype to the variables explicitly. otherwise we will get error.

→ python data types are categorized into two types

1. Fundamental datatypes
2. collections datatypes

1. Fundamental datatypes :-

→ the variables which are represented with fundamental datatypes stores the address of the object in which we can represent only one element

→ python supports the following fundamental datatypes

int
long
float
complex
bool
str

program:

→ python supports the somany build-in functions.

type():

→ This function is used to know the datatypes of the variables

id():

→ This function is used to know the address of the object which is pointed by the variables

print():

→ This function is used to display the data on the console.

```
i = 12345
print i
print type(i)
print id(i)
j = 123456789012345
print j
print type(j)
print id(j)
k = 123.123
print k
print type(k)
print id(k)
x = 3+4j
print type(x)
print id(x)
y = True
print y
print type(y)
print id(y)
```

o/p → F5

```
12345
<type 'int'>
40980256
123456789012345
<type 'long'>
27495192
123.123
<type 'float'>
40942608
3+4j
<type 'complex'>
4083652
True
<type 'bool'>
1798444108
```

* Number systems:-

According to the mathematics four number systems are available they are

1. Binary number system
2. octal number system
3. Decimal number system
4. hexadecimal number system.

1. Binary number system:-

→ The base (or) radix of the binary number system is '2'

→ The possible digits that are used in binary number system are 0 & 1

→ we can give the binary number system number as a input by giving 0b begining of number

Ex-1: i = 0b1010

$(1010)_2 = 0\times2^0 + 1\times2^1 + 0\times2^2 + 1\times2^3$

$= 0 + 2 + 0 + 8$

$= (10)_{10}$

Ex-2: j = 0b1020

it gives error message (because binary number only accepts)

2. octal number system:-

→ The base/radix of the octal number system is '8'

→ The possible digits that are used in octal number system are 0, 1, ---7.

→ we can give the input value in the form of octal number system format by giving '0' begining of the number.

Ex-1: x = 0183

$(183)_8 = 3\times8^0 + 8\times8^1 + 1\times8^2 + 0$

$= 3 + 16 + 64$

$= (83)_{10}$

Ex-2: 0182 ⟶ # error

3. Decimal number system:-

→ The base/radix of the decimal number system is 10.

→ The possible digits that are used in decimal number system are 0, 1, 2 --- 9

→ The default number system followed by python is decimal number system.

Ex-1: x = 1234

$(1234)_{10} = 4\times10^0 + 3\times10^1 + 2\times10^2 + 1\times10^3$

$= (1234)_{10}$

Ex-2: y = 12HP3

we get error because of P

4. Hexadecimal number system :-

→ The base/radix of the hexadecimal number system is 16.

→ The possible digits used in hexa-decimal number system are 0,1,---9, a, b, c---f

→ we can give the hexa-decimal system as input by giving '0x' begining of the number.

Ex-1: p = 0x25

$(25)_{16} = 5 \times 16^0 + 2 \times 16^1$

$= 5 + 32$

$= (37)_{10}$

Ex-2: q = 0x2c

$(2c)_{16} = c \times 16^0 + 2 \times 16^1$

$= 12 + 32$

$= (44)_{10}$

Ex-3: x = 0x4732 → # error

Note : We can give any number system number as input but output can be given in the form of decimal number system format.

```
x = 0b1010
print x
y = 0123
print y
z = 123H
print z
p = 0x25
print p
q = 0x2c
print q
```

o/p → 10
83
123H
37
44

bin() :- bin() is used to convert the any number system number in the form of binary format / binary number system

```
x = 0b1010
print bin(x)
y = 0123
print bin(y)
z = 1234
print bin(z)
```

```
p = 0x25
print bin(p)
```

o/p : 0b1010
0b10100011
0b1001101010
0b100101

oct():- oct() is used to convert any number system number in the form of octal number system format

```
x = 0b1010
print oct(x)
y = 0123
print oct(y)
z = 1234
print oct(z)
p = 0x25
print oct(p)
```

o/p:- 012
0123
02322
045

hex():- hex() is used to convert any ~~no~~ num-system number in the form of hexadecimal number system format.

```
x = 0b1010
print hex(x)
y = 0123
print hex(y)
z = 1234
print hex(z)
p = 0x25
print hex(p)
```

o/p:- 0xa
0x53
0x4d2
0x25

NOTE:- Non-decimal point numerical values we can represent in the form of any number system format but decimal point numerical values we can represent in the form of decimal number system format only.

```
x = 0b1010.1010 --> # error
y = 0x234.234 --> # error
```

NOTE:

→ All fundamental datatypes represents the objects are immutable objects

→ Immutable means once if we create the one object with some content later we can't modify the content of that object.

→ If we try to modify the content of the immutable objects without modifying the content of those objects it will create either new object (or) it will give error message.

→ If a fundamental datatypes represented variables are assigned with some data then internally those variables points the same object

Ex:
```
i = 1234
print i
print id(i)
j = 1234
print j
print id(j)
```

O/P:
```
1234
432777872
1234
432777872
```

String Handling :

→ The group/sequence of characters is known as a string

→ TO represent the string in python we use 'str' data type

→ We can create the string object in python in 2 ways they are

1. by using ' '
2. by using """ """

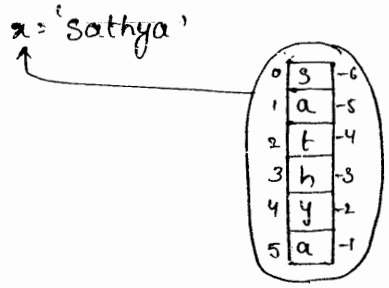
NOTE: by using ' ' we can represent only one line string
by using """ """ we can represent multiple lines string

→ string objects are Immutable objects

→ once if we create string object in some content later we cannot modify the content of that object.

→ every character in the string object is represented with unique index.

→ python supports both forward and backward indexes.



→ We can access the characters of the string object by using their indexes

```
x = 'sathya'
print x
print type(x)
print id(x)
print x[2]
print x[4]
print x[-2]
print x[-5]
```

→ o/p:

```
sathya
<type 'str'>
40870032
t
y
y
a
```

→ If the given index is not available in the string we will get index error.

```
print x[-8] ⟶ # index error
print x[7] ⟶ # index error.
```

Len() :- Len() is used to know the length of string, list, tuple, set, dictionary and soon.

```
x = 'sathya'
print x
print len(x)
```

O/P :- sathya
6

→ If we try to modify the content of string object by using index we will get the type error.

```
x = 'sathya'
x[0] = 'a' ⟶ # type error
```

String slicing : column(:) is a slice operator, which is used to extract the require content from the given string

```
x = 'sathyatech'
print x
print type(x)
print id(x)
y = x[3]
print y
print type(y)
print id(y)
z = x[2:6]
print z
a = x[3:-2]
print a
b = x[-5:-2]
print b
c = x[5:-9]
print c
d = x[6:2]
print d
```

O/P :- Sathyatech
<type 'str'>
43182176
hyatech
<type str>
41840960
thya
hyate
ate

atec

```
e = x[-5:9]
print e
f = x[12:]
print f
```

Ex-2:

```
y = 'sathya\
    technologies\
    hyderabad'
print x
y = 'sathya\n\
    technologies\n\
    hyderabad'
print y
z = '''sathya
    technologies
    hyderabad'''
print z
```

o/p:

```
sathya    technologies
sathya
    technologies
    hyderabad
sathya
    technologies
    hyderabad
```

* String methods

x = 'Hello World'

- s.lower() ⇒ hello world
- s.upper() ⇒ HELLO WORLD
- s.swapcase() ⇒ hELLO WORLD
- s.find('World') ⇒ 6
- s.count('o') ⇒ 2
- s.capitalize() ⇒ Hello World ⇒ First Leter capital
- s.replace('Hello','Hi') ⇒ Hi world

print ('I have %d cats' %6)
⇒ I have 6 cats

%d, %3d, %f, %.2f, %03d

or

Print ('I have {0:d} cats".format(6))

hyderabad ⇒ I have 6 cats

* Reading the data from keyboard:

→ we can read the data from keyboard by using following functions

1. Raw_input
2. input()

1. Raw_input():- raw_input() read the data from the keyboard in the form of string format

Ex:-
```
x = raw_input('enter fno')
print x
print type(x)
```
⇒
```
o/p: enter no 1000
1000
<type 'str'>
```

* Type conversion functions:

→ These are used to convert one datatype represent data in the form of another datatype.

int():- int() converts the string represented data in the form of int format

* Write a python program to perform addition of two numbers by ~~using~~ reading those numbers from keyboard using raw_input().

Ex:
```
x = raw_input('enter fno')
y = raw_input('enter sno')
print type(x)
print type(y)
print x+y
i = int(x)
print type(i)
print i
j = int(y)
print type(j)
print j
print i+j
```
⇒
```
o/p:- enter fno123
enter sno456
<type 'str'>
<type 'str'>
123456
<type 'int'>
123
<type 'int'>
456
579
```

Shortcut:

```
print int(raw_input('enter fno')) + int(raw_input('enter sno'))
```

```
o/p: enter fno100
enter sno200
300
```

2. long():- long() is used to convert the string represented data in the form of long data type format

```
Ex:- x = raw_input('enter longno')
     print type(x)
     print x
     y = long(x)
     print type(y)
     print y
```

⇒

```
o/p:- enter longno1234567890123345
      <type 'str'>
      1234567890123345
      <type 'long'>
      1234567890123345
```

3. float():- float() used to convert string represented data in the form of float datatype format.

```
Ex:  x = raw_input('enter floatno')
     print type(x)
     print x
     y = float(x)
     print type(y)
     print y
```

⇒

```
o/p:- enter floatno 123.123
      <type 'str'>
      123.123
      <type 'float'>
      123.123
```

4. complex():- It is used to convert the string represented data in the form of complex datatype format.

```
Ex: x = raw_input('enter complexno')
    print type(x)
    print x
    y = complex(x)
    print type(y)
    print y
```

⇒

```
o/p:- enter complexno2+4j
      <type 'str'>
      2+4j
      <type 'complex'>
      (2+4j)
```

5. bool():- bool() is used to convert the string represented data in the form of bool datatype format.

Ex: x = raw_input('enter boolno')
print type(x)
print x
y = bool(x)
print type(y)
print y

o/p: enter boolnoTRUE
<type 'str'>
TRUE
<type 'bool'>
True

NOTE: TRUE / FALSE are capitals

Ex-2: x = raw_input('enter data')
print type(x)
print x
y = int(x)
print type(y)
print y

o/p: enter data Sathya
<type 'str'>
Sathya
we will get value error.

NOTE: We can't convert data to any other conversions

2. input() : input() used to read the data from keyboard directly in our required format.

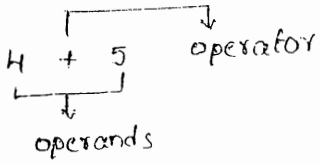
Ex-1: i = input('enter int no')
print type(i)
print i
j = input('enter long no')
print type(j)
print j
k = input('Now enter float no')
print type(k)
print k
x = input('enter complex no')
print type(x)
print x
y = input('enter bool no')
print type(y)
print y

o/p: enter int no 1234
<type 'int'>
1234
enter long no 12345678901234
<type 'long'>
12345678901234
enter float no 123.123
<type 'float'>
123.123
enter complex no 3+4j
<type 'complex'>
(3+4j)
enter bool noTrue
<type 'bool'>
True

Ex-2: fn = input ('enter first name')
ln = input ('enter last name')
print fn+ln

o/p: enter first name 'sathya'
enter last name 'tech'
sathyatech

* operators: operators are used to manipulate the values of the operands



Python supports the following types of operators. they are

1. Arithmatic operators
2. comparision (relational) operators
3. Logical operators
4. Bitwise operators
5. Assignment operators
6. special operators

1. Arithmatic operators:

→ These operators are used to perform the mathematical operations like addition, subtraction, multiplication and soon.

operator	Meaning
+	Add two operands (or) unary plus
-	subtract right operand from the left (or) unary minus
*	Multiply two operands
/	Divide left operand by right one (result float)
%	Modulus - remainder of the division of left operand by right.

//	Floor division - division that results into whole number adjusted to the left in number line
**	Exponent - left operand raised to the power of right

program:

```
x = 15
y = 4
print ('x+y = ', x+y)
print ('x-y = ', y-y)
print ('x*y = ', x*y)
print ('x/y = ', x/y)
print ('x//y = ', x//y)
print ('x **y = ', x **y)
```

⇒

o/p:
```
('x+y = ', 19)
('x-y = ', 11)
('x*y = ', 60)
('x/y = ', 3)
('x//y = ', 3)
('x **y = ', 506625)
```

Ex-2:

```
x = 15.0
y = 4
print ('x/y = ', x/y)
print ('x//y = ', x//y)
```

⇒

o/p:
```
('x/y = ', 3.75)
("x//y = ', 3.0)
```

Ex-3:

```
x = 'sathya'
y = 4
print ('x*y = ',x*y)
print ('y*x = ', y*x)
```

⇒

o/p:
```
('x*y = ', 'sathyasathyasathyasathya')
('y*x = ', 'sathyasathyasathyasathya')
```

2. comparision/ Relational operators:-

→ comparision operators are used to compare the values

→ comparision operators returns either True/False.

operator	Meaning
>	Greater than - True if left operator is greater than right one

<	Lessesthan → True if left operand is less than right one
==	Equal to → True if both operands are equal
!=	Not equal to → True if both not equal
>=	greater than (or) equal to → True if left operand is greater than or equal to right
<=	less than or equal to → True if left operand is less than or equal to right

Ex:

```
x = 10
y = 12
print('x>y is ', x>y)
print('x<y is ', x<y)
print('x==y is ', x==y)
print('x!=y is ', x!=y)
print('x>=y is ', x>=y)
print('x<=y is ', x<=y)
```

⇒

o/p:

```
('x>y is ', False)
('x<y is ', True)
('x==y is ', False)
('x!=y is ', True)
('x>=y is ', False)
('x<=y is ', True)
```

Ex-2:

```
x = 'sathya'
y = 'sathya'
print('x>y is ', x>y)
print('x<y is ', x<y)
print('x==y is ', x==y)
print('x!=y is ', x!=y)
print('x>=y is ', x>=y)
print('x<=y is ', x<=y)
```

⇒

o/p:

```
('x>y is', False)
('x<y is', False)
('x==y is', True)
('x!=y is', False)
('x>=y is', True)
('x<=y is', True)
```

3. Logical operators:

→ They are used to perform the logical operator like and, or, not

→ logical operators writtens either True/False values.

operator	Meaning
and	True if both the operands are True
OR	True if either of the operand is True
Not	True if operand is false (complements the operand)

Ex:

```
x = True
y = False
print('x and y is', x and y)
print('x or y is', x or y)
print('not x is', not x)
```

⇒

o/p: ('x and y is', False)
('x or y is', True)
('not x is', False)

4. Bitwise operator:-

→ These operators are used to perform the operations on the values based on the bits.

operator	Meaning
&	Bitwise AND
|	Bitwise OR
~	Bitwise NOT
^	Bitwise XOR
>>	Bitwise Right shift
<<	Bitwise left shift

Ex:
```
x = 10
y = 4
print ("x&y is : ", x&y)
print ("x|y is : ", x|y)
print ("~x is : ", ~x)
print ("x^y is : ", x^y)
print ("x>>2 is : ", x>>2)
print ("x<<2 is : ", x<<2)
```

⇒

o/p:
```
('x&y is : ', 0)
('x|y is : ', 14)
('~x is : ', -11)
('x^y is : ', 14)
('x>>2 is : ', 2)
('x<<2 is : ', 40)
```

NOTE:- Bitwise operators internally converts the operands values in the form of binary format and performs the operation and gives the results in the form of decimal format.

* Assignment operators :

→ Assignment operators are used to assign the values to the variables

→ It contains both bitwise and arithmetic operators

operator	example	equivalent to
=	x = 5	x = 5
+=	x += 5	x = x+5
-=	x -= 5	x = x-5
*=	x *= 5	x = x*5
/=	x /= 5	x = x/5
%=	x %= 5	x = x%5
//=	x //= 5	x = x//5
**=	x **= 5	x = x**5
&=	x &= 5	x = x&5
|=	x |= 5	x = x|5
^=	x ^= 5	x = x^5
>>=	x >>= 5	x = x>>5
<<=	x <<= 5	x = x<<5

Ex:

```
x = 10
print("x = 10 :", x)
x+=5
print("x+=5 :", x)
x-=5
print("x-=5 :", x)
x*=5
print("x*=5 :";x)
x **=5
print("x **=5 :", x)
x/=5
print("x/=5 :", x)
x %=5
print("x %=5", x)
x //=5
print("x //=5", x)
y=20
y&=10
print("y&=10 :", y)
y|=30
print("y|=30 :", y)
y^=10
print("y^=10 :", y)
y>>=3
print("y>>=3 :", y)
y<<=3
print("y<<=3 :", y)
```

⇒

o/p:-

```
('x=10 :', 10)
('x+=5 :', 15)
('x-=5 :', 10)
('x*=5 :', 50)
('x **=5 :', 312500000)
('x/=5 :', 62500000)
('x %=5 :', 0)
('x//=5 :', 0)
('y&=10 :', 0)
('y|=30 :', 30)
('y^=10 :', 20)
('y>>=3 :', 2)
('y<<=3 :', 16)
```

* Special operators:-

python supports the two types of special operators.

i. identity operators

ii. Membership operators

i. identity operators :-

→ identity operators are used to compare the addresses of the memory locations which are pointed by the operands

→ identity operators returns true (or) false

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

Ex:
```
x1 = 5
y1 = 5
x2 = 'hello'
y2 = 'hello'
print("x1 is y1 : ", x1 is y1)
print("x1 is not y1 : ", x1 is not y1)
print("x2 is y2 : ", x2 is y2)
print("x2 is not y2 : ", x2 is not y2)
```

o/p:
```
('x1 is y1 : ', True)
('x1 is not y1 : ', False)
('x2 is y2 : ', True)
('x2 is not y2 : ', False)
```

ii. Membership operators :-

→ Membership operators are used to search for a particular element in a string . list , tuple , set and soon

operator	Meaning
in	True if value/ variable is found in sequence
not in	True if value/ variable is not found in sequence

[redacted] operators returns True or False values

Ex: x = 'hello world'

```
print('h' in x)
print('h' not in x)
print('hello' in x)
print('hello' not in x)
```

⟹

o/p: True
False
True
False

* operators precedence :-

operator	description
**	Exponentiation (raise to the power)
~ + -	complement, uncary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right shift and left bitwise shift
&	Bitwise AND
^ |	Bitwise exclusive OR and regular OR
<= < > >=	Comparision operator
<> == !=	equality operator
= % = / = // = - = + = * = ** =	Assignment operator
is is not	Identity operators
in not in	membership operators
not or and	logical operators

•Ex:-
```
a = 20
b = 10
c = 15
d = 5
e = 0
e = (a+b) * c/d
print ("value of (a+b) * c/d is", e)
e = ((a+b) * c) / d
print ("value of ((a+b) *c)/d is", e)
e = (a+b) * (c/d)
print ("value of (a+b) * (c/d) is", e)
e = a+(b*c)/d
print("value of a+(b*c)/d is", e)
```

o/p:
```
('value of (a+b)*c/d is', 90)
('value of ((a+b) *c)/d is', 90)
('value of (a+b)* (c/d) is', 90)
('value of a+(b*c)/d is', 50)
```

* conditional statements:

→ conditional statements are used to decide whether code has to be execute (or) skip based on the evaluation of the condition.

→ After evaluating the condition, it should return either True or False value

Elements of conditional statements:-

In conditional statements we use two elements

i. condition

ii. Block

i. condition:- Any expression which returns either True (or) False value after evaluating that expression is known as "condition".

Block :- All the statements which are following same space indentation is known as "Block".

→ Blocks begins with when the indentation increases
→ Blocks can contain other blocks
→ Blocks end when the indentation decreases to zero (or) to a containing blocks indentation.

python supports three conditional statements, There are

i. if (or) simple if
ii. if else
iii. elif

i. if (or) simple if :-

syntax :- if condition : statement

(or)

if condition :

stmnt 1
stmnt 2
- - - - -

→ simple if executes the block if condition returns True. otherwise it will skip the execution of the block.

```
Ex: Name = raw_input('enter name :')
    if Name = 'sathya' :
        print('Hi, sathya')
    print('Hello world')
```

o/p-1 :

```
enter name : sathya
Hi, sathya
Hello world
```

o/p-2 :

```
enter name : lokesh
Hello world
```

ii. if - else :

Syntax: if condition :

    Stmnt 1
    Stmnt 2
    - - - - -
    - - - - -

else :

    Stmnt 1
    - - - - -
    - - - - -

→ In if - else condition returns True then it will execute if block otherwise it will execute else block.

Ex:
```
Name = raw_input('enter name :')
if Name == 'sathya':
    print('Hi, sathya')
else:
    print('Hi, stranger')
print('Hello world')
```

o/p-1:
```
enter name : sathya
Hi, sathya
Hello world
```

o/p-2:
```
enter name : lokesh
Hi, stranger
Hello world
```

(or)

```
Name = raw_input('enter name :')
if Name == 'sathya' : print('Hi, sathya')
else:
    print('Hi, stranger')
print('Hello world')
```

elif :-

syntax: if condition :

stmnt 1

stmnt 2

elif condition :

stmnt 1

stmnt 2

(optional) else :

stmnt 1

Ex:

```
name = raw_input('enter name:')
age = input('enter age')
if age<15:
    print name, 'you are kid'
elif age<40:
    print name, 'you are young'
elif age<100:
    print name, 'you are old'
else:
    print name, 'you are alin'
print 'hello world'
```

o/p-1:

```
enter name: tom
enter age : 12
tom, you are kid
Hello world
```

o/p-2:

```
entername : priya
enter age : 20
priya, you are young
Hello world
```

o/p-3:

```
enter name : John
enter age : 45
John, you are old
Hello world
```

o/p-4:

```
enter name : sathya
enter age : 120
sathya, you are alien
Hello world
```

* write a program to find biggest number among 2 numbers.

```
num1 = input (' enter first number :')
num2 = input (' enter second number :')
if num1 > num2 :
    print num1, 'is big'
else :
    print num2, 'is big'
```

o/p-1:

```
enter first number : 20
enter second number : 10
20 is big
```

o/p-2:

```
enter first number : 50
enter second number : 70
70 is big
```

* write a program to find biggest number among 3 numbers.

```
num1 = input(' enter first number: ')
num2 = input (' enter second number :')
num3 = input (' enter third number :')
if num1 > num2 and num1 > num3 :
    print num1, 'is big'
elif num2 > num3 :
    print num2, 'is big'
else :
    print num3, 'is big'
```

o/p: enter first number : 10
enter second number : 30
enter third number : 20
30 is big.

* write a program to find smallest of two numbers.

```
num1 = input('enter first number :')
num2 = input('enter second number :')
if num1 < num2 :
    print num1, 'is small'
else:
    print num2, 'is small'
```

o/p: enter first number: 3
enter second number : 2
2 is small.

* write a program to find smallest of three numbers.

```
num1 = input('enter first number:')
num2 = input('enter second number :')
num3 = input('enter third number :')
if num1 < num2 and num1 < num3:
    print num1, 'is small'
elif num2 < num3 :
    print num2, 'is small'
else:
    print num3, 'is small'
```

o/p: enter first number 10
enter second number 20
enter third number 5
5 is small

Looping statements:-

→ Looping statements are used to execute the set of statements repeatedly

→ python supports 2 types of looping statements

i. for loop

ii. while loop

i. for loop:

→ for loop executes the set of statements (or) blocks with respect to every element of the string / collection objects

syntax:

for variablename in string / collection variable :

    - - - - -
    - - - - - } block
    - - - - -
    - - - - -

range(): range() generates the group of values based on the given range and gives it as a list.

```
Ex: print range (10)
    print range (1, 11)
    print range (0, 30, 5)
    print range (0, 10, 3)
    print range (0, -10, -1)
    print range (0)
    print range (1, 0)
```

```
o/p: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
     [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
     [0, 5, 10, 15, 20, 25]
     [0, 3, 6, 9]
     [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
     [ ]
     [ ]
```

→ In python for loop every time condition is not verified so performance will be faster

Ex:
```
for x in range(10):
    print x
for y in range(0,10,2):
    print y
for z in range(10):
    if z%2==0:
        print z
```

o/p:
```
0
1
2
3
4
5
6
7
8
9
0
2
4
6
8
0
2
4
6
8
```

Ex:
```
x='sathya'
print x
for p in x:
    print p
```

⇒

o/p:
```
sathya
s
a
t
h
y
a
```

(or)

```
x = 'sathya'
print x
for p in x:
    print p,
```

$\Rightarrow$ o/p: sathya

s a t h y a

Ex:
```
for x in range(1, 11):
    print 10, "*", x, "=", 10 * x
```

o/p:
```
10 * 1 = 10
10 * 2 = 20
10 * 3 = 30
10 * 4 = 40
10 * 5 = 50
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100
```

2. while loop:

while loop executes the set of statements or block repeatedly until condition will become false

syntax: while condition:

stmnt 1
- - - - -
- - - - -
- - - - -

Ex:
```
sum = 0
spam = 1
while spam < 11:
    sum = sum + spam
    spam = spam + 1
print sum
```

$\Rightarrow$ o/p: 55

Ex:

```
name = ' '
while name != 'sathya':
    name = raw_input('enter name:')
print 'thank you'
```

o/p: enter name : priya
enter name : Laxmi
enter name : sathya
thank you

Example for infinite loop:

```
while True:
    name = raw_input('please type your name:')
print 'thank you'
```

o/p: please type your name : lokesh
please type your name : priya
please type your name : sathya
⋮

Break:

→ break statements used to exit from the loops
→ generally we use the break statement in infinite loops.

Ex:

```
while True:
    name = raw_input('enter name:')
    if name == 'sathya'
        break
    print 'Thank you'
```

o/p: enter name : priya
enter name : sathya
Thank you

Continue :

→ continue statement can be used in the loops

→ whenever we use continue statement without executing the remaining part of the loop control will goes to starting of the loop and continues the next iteration.

Ex:-

```
while True:
    name = raw_input("enter your name")
    if name != 'sathya':
        continue
    print "hello sathya, enter password"
    password = raw_input('enter password')
    if password == 'priya':
        break
print ("Access granted")
```

O/P:

```
enter your name lokesh
enter your name sathya
hello sathya, enter password
enter password priya
Access granted
```

Ex:

```
count = 1
while True:
    if count == 5:
        count = count + 1
        continue
    print "hi", count
    count = count + 1
    if count == 10:
        break
```

⟹

o/p:

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

pass:

→ syntactically. block is required but we don't want to perform any action logically then we can replace the block with 'pass'

→ pass is a keyword it does nothing.

Ex:

```
i = 100
while i <= 100:
    pass
if i < 200:
    pass
for i in range(10):
    pass
```

del:

→ del is a keyword, which is used to remove the variables (or) elements from the memory location.

→ After using the variables, then it is recommended to remove those variables from the memory location by using 'del' keyword.

→ If you don't remove the variables, after usage is over then those variables available in the memory location until

execution of the program is over so that performance of the application will be degraded.

→ In garbage collector, it removes spaces after entire program is completed but del is deleted with in the program

Ex:

```
i = 1000
j = 2000
k = i+j
print k
del k
x = i-j
print x
del x
print k
print x
```

⟹

o/p: 3000
-1000
Name error.

Note: we can't remove the elements of the immutable objects by using 'del' keyword but we can remove the immutable objects by using 'del' keyword.

Ex:

```
i = 'priya'
del i
x = 'sathya'
del x[2]   # error
```

List:

→ List is used to represent group of elements into a single entity (or) object.

→ insertion order is preserved

→ duplicate elements are allowed.

→ hetrogeneous elements are allowed.

→ every element in the list object is represented with unique index.

→ List supports both positive and negative indexes

→ we can perform the operations on the elements of the list object by using "indexes".

→ we can create the list object by using "list()" function and by using square brackets "[ ]"

→ List objects are mutable objects.

Ex:
```
a = [ ]
print a
print type(a)
b = [10, 20, 30]
print b
print type(b)
x = list()
print x
print type(x)
y = list('sathya')
print y
print type(y)
```

⇒

o/p:
```
[ ]
<type 'list'>
[10, 20, 30]
<type 'list'>
[ ]
<type 'list'>
['s', 'a', 't', 'h', 'y', 'a']
<type 'list'>
```

Ex:
```
x = [10, 20, 'sathya', True, 123.123, 10, 'sathya']
print x
print id(x)
y = [10, 20, 30, 40, 50]
print y
print id(y)
print y[0]
print y[0:4]
print y[-2]
```

```
print  y[0:-3]
print  y[7] # index error
y[2] = 100
print  y
print  id(y)
del  y[3]
print  y
print  id(y)
```

o/p: [10, 20, 'sathya', True, 123.123, 10, 'sathya']

```
35466816
[10, 20, 30, 40, 50]
35479600
30
[30, 40]
40
[10, 20]
[10, 20, 100, 40, 50]
35479600
[10, 20, 100, 50]
35479600
```

Ex: x = [100, 123.123, 'sathya']

```
print x
print len(x)
a,b,c = x
print a
print type(a)
print b
print type(b)
print c
print type(c)
y = [10, 20, [100, 800, 300], 'sathya']
print y
```

o/p: [100, 123.123, 'sathya']

```
3
100
<type 'int'>
123.123
<type 'float'>
sathya
<type 'str'>
[10, 20, [100, 200, 300], 'sathya']
```

Ex:

```
x = ['sathya', 'hyderabad', 'ameerpet']
print x
for p in x:
    print p, " ---> ", len(p)
    for q in p:
        print q
```

o/p: ['sathya', 'hyderabad', 'ameerpet']

```
sathya ---> 6
s
a
t
h
y
a
hyderabad ---> 9
h
y
d
e
r
a
b
a
d
ameerpet ---> 8
a
m
e
e
r
p
e
t
```

Ex:

```
x = [[10, 20, 30], [40, 50, 60], [70, 80, 90]]
print x
for p in x :
    for q in p:
        print q ,
    print
```

o/p: [[10, 20, 30], [40, 50, 60], [70, 80, 90]]

```
10 20 30
40 50 60
70 80 90
```

Ex:

```
x = [10, 20, 30]
print x
print id(x)
y = [10, 20, 30]
print id(y)
print x == y
print x is y
z = y
print z
print id(z)
print y == z
print y is z
```

⟹

o/p:
```
[10, 20, 30]
31878520
31882256
True
False
[10, 20, 30]
31882256
True
True
```

Ex:

```
x = [10, 20, 30, 40, 50]
print x
print id(x)
print len(x)
x.append(60)
```

```
print x
print id(x)
x.insert(2,70)
print x
y = ['abc', 'def', 'xyz', 30]
print y
x.extend(y)
print x
print x.index('abc')
x.remove(30)
print x
x.sort()
print x
x.reverse()
print x
x.pop()
print x
x.pop(3)
print x
print id(x)
```

O/P:

```
[10, 20, 30, 40, 50]
34434424
5
[10, 20, 30, 40, 50, 60]
34434424
[10, 20, 70, 30, 40, 50, 60]
['abc', 'def', 'xyz', 30]
[10, 20, 70, 30, 40, 50, 60, 'abc', 'def', 'xyz', 30]
7
```

[10, 20, 70, 40, 50, 60, 'abc', 'def', 'xyz', 30]

[10, 20, 30, 40, 50, 60, 70, 'abc', 'def', 'xyz']

['xyz', 'def', 'abc', 70, 60, 50, 40, 30, 20, 10]

['xyz', 'def', 'abc', 70, 60, 50, 40, 30, 20]

['xyz', 'def', 'abc', 60, 50, 40, 30, 20]

34434424

List comprehension:

→ creating the list object by writing business logic is known as "List comprehension".

Ex:

```
a = [x for x in range(10)]
print a
y = [x**x for x in range(5)]
print y
i = [x for x in range(0,10,2)]
print i
j = [x*x for x in range(0,10,2)]
print j
k = [p for p in a if p%2==0]
print k
```

O/P: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[1, 1, 4, 27, 256]

[0, 2, 4, 6, 8]

[0, 4, 16, 36, 64]

[0, 2, 4, 6, 8]

Ex:

```
noprimes = [j for i in range(2,8) for j in range(i*2, 50, i)]
primes = [x for x in range(2,50) if x not in noprimes]
print primes
```

o/p:

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

Ex:

```
line = 'the quick brown fox jumps over the lazy dog'
words = line.split()
print words
stuff = [[w.upper(), w.lower(), len(w)] for w in words]
for i in stuff:
    print i
```

o/p:

['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']

['THE', 'the', 3]

['QUICK', 'quick', 5]

['BROWN', 'brown', 5]

['FOX', 'fox', 3]

['JUMPS', 'jumps', 5]

['OVER', 'over', 4]

['THE', 'the', 3]

['LAZY', 'lazy', 4]

['DOG', 'dog', 3]

tuple:

→ tuple is used to represent group of elements into a single entity

→ tuple objects are immutable objects i.e., once if you create tuple object with some element, later we cannot modify the elements of that object.

→ Insertion order is preserved

→ Duplicate elements are allowed

→ every element in the tuple object is represented with index

→ tuple supports both forward and backward indexes

Ex:

```
a = ()
print a
print type(a)
b = tuple()
print b
print type(b)
c = (1,2,3)
print c
print type(c)
d = (1,"hello", 3.4)
print d
print type(d)
e = ("mouse", [8,4,6], (1,2,3))
print e
print type(e)
f = 3, 4.6, "dog"
x,y,z = f
print x
print type(x)
print y
print type(y)
```

```
print z
print type(z)
```

O/P:
```
()
<type 'tuple'>
()
<type 'tuple'>
(1,2,3)
<type 'tuple'>
(1,'hello',3.4)
<type 'tuple'>
('mouse',[8,4,6],(1,2,3))
<type 'tuple'>
3
<type 'int'>
4.6
<type 'float'>
dog
<type 'str'>
```

Ex:
```
x = tuple('sathyatech')
print x
print len(x)
print x[5]
print x[-3]
y = x[2:7]
print y
print type(y)
print x[-7:-2]
del x[-2] # typeerror
```

O/P:
```
('s','a','t','h','y','a','t','e','c','h')
10
a
e
```

('t','h','y','a','t')

<type 'tuple'>

('h','y','a','t','e')

Ex:

```
x = (10, 20, 30, 10, 40, 50, 10)
print x
for p in x:
    print p
y = (1.1, True, 10, 'sathya')
print y
for q in y:
    print q
```

⇒

o/p: (10, 20, 30, 10, 40, 50, 10)

10
20
30
10
40
50
10

(1.1, True, 10, 'sathya')

1.1
True
10
Sathya

Ex:

```
x = ([10,20,30], (40, 50, 60), 'sathya')
print x
for p in x:
    print p, type(p), len(p)
    for q in p:
        print q
```

o/p: ([10,20,30], (40, 50, 60), 'sathya')

[10, 20, 30] <type 'list'> 3

10
20
30

(40, 50, 60) <type 'tuple'> 3

40
50
60

```
sathya < type 'str'> 6
S
a
t
h
y
a
```

Ex:
```
x = (10, 20, 30, 40, 50, 20, 10, 10)
print x
a = x.count(10)
print a
b = x.index(20)
print b
print 100 in x
```

O/P:
```
(10, 20, 30, 40, 50, 20, 10, 10)
3
1
False
```

Ex:
```
x = (10, 20, 30)
print x
print id(x)
y = (10, 40, 50)
print y
print id(y)
z = x+y
print z
print id(z)
a = (10, 20, 30)
print a
print id(a)
b = ('sathya', 'tech')*3
```

```
print b
print x == a
print x is a
```

O/P: (10, 20, 30)
31954144
(10, 40, 50)
33120544
(10, 20, 30, 10, 40, 50)
31812552
(10, 20, 30)
33120664
('sathya', 'tech', 'sathya', 'tech', 'sathya', 'tech')
True
False

NOTE: tuple comprehension is not supported in python, if we use it internally creates generator object.

EX:
```
S = (x**2 for x in range(10))
print S
print type(S)
for P in S:
    print P
```

O/P: <generator object <genexpr> at 0x08096198>
<type 'generator'>
0
1
4
9
16
25
36
49
64
81

* Differences between list and tuple.

List	Tuple
→ list objects are mutable objects	→ tuple objects are immutable objects
→ iterating the list is slower	→ iterating the tuple is faster
→ List objects are not used as a keys for the dictionary	→ the tuple objects which contains immutable objects can be used as a key for the dictionary
→ if the data changes frequently, then it is recommended to represent that data by using list	→ if the data does't change then it is recommended to represent the data by using tuple.

* Set:

→ set is used to represent group of immutable elements into a single entity.
→ set objects are mutable objects.
→ insertion order is not preserved.
→ Duplicate elements are not allowed
→ on the top of the set objects, we can perform the mathemati -cal set operations like union, inter-section, symmetric difference and soon.

Ex:
```
x = set()
print x
print type(x)
y = set([10, 20, 30])
print y
z = {100, 200, 300, 400, 100, 100}
print z
```

o/p:

```
set([])
<type 'set'>
set([10, 20, 30])
Set([400, 300, 100, 200])
```

→ set objects does not support "indexing".
→ set allows heterogeneous elements.

Ex:
```
x = {10, 20, 30}
print x
print x[0]
del x[1]
print x[1:2]
```

o/p:
```
set([10, 20, 30])
TypeError: 'set' object does not support indexing
```

Ex:
```
x = {10, 1.1, True, 'sathya', (20, 30, 40)}
print x
for p in x:
    print p
```

o/p:
```
set(['sathya', (20, 30, 40), 10, 1.1, True])
sathya
(20, 30, 40)
10
1.1
True
```

Ex:
```
x = {10, 20, 30, 40}
print x
x.add(50)
```

```
print x
x.update([30, 60, 70])
print x
x.update([80, 50], [15, 25, 35, 40])
print x
x.discard(30)
print x
x.remove(25)
print x
```

o/p: set([40, 10, 20, 30])
set([40, 10, 20, 50, 30])
set([70, 40, 10, 50, 20, 60, 80])
set([35, 70, 40, 10, 15, 80, 50, 20, 25, 60, 30])
set([35, 70, 40, 10, 15, 80, 50, 20, 60])
set([35, 70, 40, 10, 15, 80, 50, 20, 60])

Note: In discard method, if the specified element is not present in set. It doesn't gives error whereas in remove method, it gives error

Ex:

```
x = set('sathya')
print x
print x.pop()
print x
x.clear()
print x
print len(x)
```

o/p: set(['a', 'h', 's', 't', 'y'])
a
set(['h', 's', 't', 'y'])
set([])
0

* set mathematical operations :

Ex: A = {1, 2, 3, 4, 5}

```
print A
B = {4, 5, 6, 7, 8}
print B
print A|B
print A.union(B)
print A&B
print A.intersection(B)
print A-B
print A.difference(B)
print B-A
print B.difference(A)
print A^B
print A.symmetric_difference(B)
```

O/P:

```
set([1, 2, 3, 4, 5])
set([8, 4, 5, 6, 7])
set([1, 2, 3, 4, 5, 6, 7, 8])
set([1, 2, 3, 4, 5, 6, 7, 8])
set([4, 5])
set([4, 5])
set([1, 2, 3])
set([1, 2, 3])
set([8, 6, 7])
set([8, 6, 7])
set([1, 2, 3, 6, 7, 8])
set([1, 2, 3, 6, 7, 8])
```

Ex:

```
x = set('lokesh')
print x
print 'e' in x
print 't' in x
for letter in set("apple"):
    print (letter)
for p in set(range(1,5)):
    print p
```

o/p: set(['e', 'h', 'k', 'l', 'o', 's'])

True
False
a
p
e
l
1
2
3
4

Ex: `s = {x for x in range(10)}`

```
print s
t = {y**2 for y in range(10)}
print t
```

o/p: set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
set([0, 1, 4, 8, 64, 9, 16, 49, 25, 36])

* write a python program to eleminate the duplicate elements from list.

Ex: `x = [10, 20, 30, 10, 30, 40]`

```
print x
print set(x)
```

o/p: [10, 20, 30, 10, 30, 40]

set([40, 10, 20, 30])

(or)

```
values = [5,5,1,1,2,3,4,4,5]
print values
out = [ ]
seen = set()
for value in values:
    if value not in seen:
        output.append(value)
        seen.add(value)
print output
```

o/p: [5,5,1,1,2,3,4,4,5]

[5,1,2,3,4]

* Dictionary:

→ Dictionary is used to represent group of key, value pairs into a single entity.

→ insertion order is not preserved

→ Duplicate keys are not allowed but values can be duplicate

→ Heterogeneous keys and values are allowed

→ Immutable objects only allowed for keys.

Ex:

```
x = { }
print x
print type(x)
y = dict([(1,'apple'),(2,'ball')])
print y
print type(y)
z = {'java':90,'python':99,'hadoop':95}
print z
```

```
print type(z)
print len(z)
```

O/P: { }

<type 'dict'>

{1: 'apple', 2: 'ball'}

<type 'dict'>

{'python': 99, 'java': 90, 'hadoop': 95}

<type 'dict'>

3

Ex: x = {10: 'abc', 20: 'def', 30: 'ghi', 20: 'sathya', 40: 'abc'}

```
print x
print x[10]
print x[20]
print x[50]
```

O/P: {40: 'abc', 10: 'abc', 20: 'sathya', 30: 'ghi'}

abc

sathya

KeyError

Ex:

```
x = {10: 12.12, True: 100, 1.1: 'abc', 'def': False, 'names': ['hyd', 'bang',
        'vizi'], (1.1, 2.2, 3.3): 'india'}
print x
k = x.keys()
print k
v = x.values()
print v
```

o/p:

```
{True: 100, 1.1: 'abc', 10: 12.12, (1.1, 2.2, 3.3): 'india', 'names':
                                    ['hyd', 'bang', 'vis'], 'def': False}
[True, 1.1, 10, (1.1, 2.2, 3.3), 'names', 'def']
[100, 'abc', 12.12, 'india', ['hyd', 'bang', 'vis'], False]
```

Ex:
```
x = {[10, 20, 30]: 'sathya'}
print x
y = {['abc', 'def', 'xyz']: 'priya'}
print y
z = {(10, 20, [100, 200]): 'hyd'}
print z
```

o/p: type error.

Ex:
```
x = {10:'abc', 20: 'def', 30:'isk', 40:'def'}
print x
x[50] = 'hyd'
print x
x[40] = 'bang'
print x
x.pop(30)
print x
x.popitem()
print x
del x[20]
print x
x.clear()
print x
del x
```

o/p:

{40: 'def', 10: 'abc', 20: 'def', 30: 'ijk'}

{40: 'def', 10: 'abc', 20: 'def', 50: 'hyd', 30: 'ijk'}

{40: 'bang', 10: 'abc', 20: 'def', 50: 'hyd', 30: 'ijk'}

{40: 'bang', 10: 'abc', 20: 'def', 50: 'hyd'}

{10: 'abc', 20: 'def', 50: 'hyd'}

{10: 'abc', 50: 'hyd'}

{}

Ex:

```
x = {'rama' : 60, 'sita' : 50, 'sumith' : 30, 'likitha' : 25}
print x
y = x.items()
print Y
for P in Y :
    print P[0], '\t', P[1]
```

o/p:

{'sumith' : 30, 'sita' : 50, 'rama' : 60, 'likitha' : 25}

[('sumith', 30), ('sita', 50), ('rama', 60), ('likitha', 25)]

```
sumith   30
sita     50
rama     60
likitha  25
```

* Addition of two 3x3 matrices:

```
x = [[12,7,3], [4,5,6], [7,8,9]]
y = [[5,8,1], [6,7,3], [4,5,6]]
result = [[x[i][j] + y[i][j] for j in range (len(x[0]))]
                             for i in range (len(x))]
for r in result:
    print(r)
```

(or)

```
# iteration through rows
for i in range(len(x)):
    for j in range(len(x[0])):
        result[i][j] = x[i][j] + y[i][j]
for r in result:
        print(r)
```

o/p: [17, 15, 4]
[10, 12, 9]
[11, 13, 15]
[17, 15, 4]
[10, 12, 9]
[11, 13, 15]

Ex: marks = {}.fromkeys(['math', 'english', 'science'], 0)

```
print marks
for item in marks.items():
    print(item)
print sorted(marks.keys())
```

o/p: {'science': 0, 'math': 0, 'english': 0}
('science', 0)
('math', 0)
('english', 0)
['english', 'math', 'science']

Ex:

```
squares = {x * x * x for x in range(6)}
print squares
odd_squares = {x : x * x for x in range(11) if x % 2 == 1}
print odd_squares
d = {n : True for n in range(5)}
print d
```

O/P:

{0:0, 1:1, 2:4, 3:9, 4:16, 5:25}

{1:1, 3:9, 9:81, 5:25, 7:49}

{0:True, 1:True, 2:True, 3:True, 4:True}

Exception handling:- (1)

→ Generally we get the two types of errors in any programming language they are

1. syntax errors
2. runtime errors

1. syntax errors:-

→ The errors which occurs because of invalid syntax are knows as syntax errors.

→ whenever we run the python program then internally we will checks for the syntaxes. If any syntax is written wrongly bytecode will not be generated for the python program.

→ without generating the bytecode program execution will not takes place.

→ python programmers (or) developers has to provide the solutions to the syntax errors.

Ex:-
```
def m1():
    print "in m1"
```
o/p - syntax error.

2. Runtime errors:-

→ The errors which occurs at the time of execution of the program are known as runtime errors.

→ Generally we will get the run time errors because of program logic, invalid input, memory related problems, ----

→ with respect to runtime error corresponding class is available and we call those classes as a exception classes

i. key error, ii. Index error, iii. zerodivision error.

→ At the time of execution of the python program. If any runtime error is occured then internally corresponding runtime error representation class object will be created.

→ If the python program does not contain the code to handle that object then program will be terminated abnormally.

* Abnormal termination :-

→ The concept of terminating the program in the middle of its execution without executing last statement of the program is known as abnormal termination.

Ex:
```
print "hi"
i = input("enter fno")
j = input("enter sno")
k = i/j
print k
print "bye"
```

o/p:
```
hi
enter fno 100
enter sno 0
```
abnormal termination takes place

→ Abnormal termination is the undesirable situation in any programming language.

* Exception handling :-

→ The concept of identifying the runtime error representation class object which is created internally at the time of execution of the program, receiving that object and assigning that object to the reference variable of corresponding runtime error representation class is known as a exception handling.

→ Exception handling is used to stop the abnormal termination whenever runtime error is occured.

→ we can implement the exception handling in python by using try and except blocks.

* try blocks :-

→ The statements which ~~occurs~~ causes to runtime errors and other statements which depends on the execution of the runtime error occur statements are recommended to represent by using try block.

syntax:-
```
try
  :
```

→ At the time of execution of the try block statements if any runtime error representation class object is created then immediately that object is identified by the try block, received by the try block and forward it to the except block without executing remaining statements of the try block.

NOTE: If there is no runtime error occurs at the time of execution of the tryblock then control will not go to the except block.

* Except block:-

→ Except block should be priceeded by the try block.

→ Except block receives the runtime error representation class object which is given by the try block and assigns that object to the reference variable of corresponding runtime error representation class.

→ In except block we can define the statements to display the user friendly error messages.

Ex:
```
print "hi"
i = input("enter fno")
j = input("enter sno")
try:
k = i/j
print k
except(zero divisionerror):
print "second value can not be zero"
print "bye"
```

o/p:
```
hi
enter fno 1000
enter sno 0
second value can not be zero
bye
```

* single try with multiple except blocks:

→ whenever runtime error is occured while executing try block statements then control will go to the first except block.

→ If first except block is not handle that runtime error then control will goto the second except block.

→ If all except blocks are not handle that runtime error then program will be terminated abnormally.

Ex:
```
print "hi"
try:
i = input("enter fno")
j = input("enter sno")
k = i/j
print k
except(zeroDivisionError):
print "second value can not be zero"
except(NameError):
print "do not enter string values"
print "bye"
```

o/p: hi
enter fno abc
do not enter string values

* default except block:-

→ Default except block is used to handle the any type of runtime error

→ In default except block we write the statements to display the common error messages (General error messages)

Ex:
```
print "hi"
try:
    i = raw_input("enter fno")
    j = raw_input("enter sno")
    a = int(i)
    b = int(j)
    k = a/b
    print k
except(zeroDivisionError):
    print "second value can not be zero"
```

```
except(NameError):
    print "do not enter string values"
except:
    print "error occured"
    print "bye"
```

o/p:- hi
enter fno abc
enter sno 100
error occured
bye

Ex-9:

```
print "hi"
try:
    i = raw_input("enter fno")
    j = raw_input("enter sno")
    a = int(i)
    b = int(j)
    k = a/b
    print k
except:
    print "error occured"
    print "bye"
```

o/p:- hi
enter fno 100
enter sno 0
error occured
bye

→ default except block should be the last except block whenever we define single try with multiple except block otherwise we will get syntax error.

Ex:-

```
print "hi"
try:
    i = raw_input("enter fno")
    j = raw_input("enter sno")
    a = int(i)
    b = int(j)
    k = a/b
    print k
except:
    print "error occured"
except(zeroDivisionError):
    print "second value can not be zero"
except(NameError):
    print "do not enter string values"
    print "bye"
```

* Finally block:-

→ The set of statements which are compulsary to execute whether exception is occured (or) not occured. Even though exception is occured whether it is handle (or) not handle recommended to represent in finally block.

→ Resource releasing statements (File closing statements data base connection closing statements) are recommended to represent in finally block.

→ Finally block should be preceded by either try block (or) except block

syntax-1:
```
try:
    ≡
finally:
    ≡
```

syntax-2:
```
try:
    ≡
except():
    ≡
finally:
    ≡
```

Ex:-
```
try:
    x = input("enter fno")
    y = input("enter sno")
    z = x/y
    print z
except(zeroDivisionError):
    print "second no can not be zero"
finally:
    print "end"
    print "bye"
```

o/p:-
```
enter fno 100
enter sno 20
5
end
bye
```

```
enter fno 100
enter sno 0
second no can not be zero
end
bye
```

```
enter fno 100
enter sno abc
end
error: program is terminated abnormally
```

* Nested try blocks :-

→ The concept of defining one try block inside another try block is known as a nested try blocks.

syntax :-
```
try :
  ≡
  try:
    ≡
  except :
    ≡
except :
  ≡
```

→ A try block which contains an another try block is known as a outer try block.

→ A try block which is define in another try block is known as a inner try block.

→ If exception is occured in the outer try block then control will goto the outer try except block.

→ If outer try except block is not handle that exception then program will be terminated abnormally.

→ If exception is occured in the inner try block then control will goto the inner try except block.

→ If inner try except block is not handle that exception then control will goto the outer try except block.

Ex-1:-
```
try :
    print " in try1 "
  try:
    print " in try2 "
  try:
    print " in try3 "
  except:
    print " in try3 "
except:
    print " in except2 "
except:
    print " in except1 "
```

o/p :- in try1
in try2
in try3

Ex-2:

```
try:
  print "in try1"
try:
  x = input("enter fno")
  y = input("enter sno")
  z = x/y
  print z
except(zeroDivisionError):
  print "second value cannot be zero"
except:
  print "in except1"
```

O/P:

```
in try1
enter fno 100
enter sno 0
second value can not be zero
in try1
enter fno 100
enter sno abc
in except1
```

Ex-3:

```
print "in try1"
try:
  print "in try2"
except:
  print "in except2"
finally:
  print "in finally2"
except:
  print "in except1"
try:
  print "in try3"
except:
  print "in except3"
```

```
finally:
    print "in finally3"
finally:
    print "in finally1"
try
    print "in try4"
except:
    print "in except4"
finally:
    print "in finally4"
```

o/p:- in try1
in try2
in finally2
in finally1
in try4
in finally4

* Types of exceptions :-

Exceptions are categorized into two categories they are
1. pre-defined exceptions, 2. user defined exceptions.

1. pre-defined exceptions :-

→ The runtime errors representation class which are present in python software are known as pre-defined exceptions.
Ex: value error, zero division error, key error, index error, name error ---

→ pre-defined exceptions rised automatically whenever corresponding runtime error is occured.

→ After raising the pre-defined exceptions programmers handle those exceptions by using try and except blocks.

2. user defined exceptions :-

→ The runtime errors representation classes which are defined by the programmers explicitly according to their business requirements. are known as user defined exceptions.

→ Any user defined class which is extending any one of the pre-defined exceptions class is known as a user defined exception class.

syntax:- class classname (predefined exception classname)

_ _ _ _
_ _ _ _
_ _ _ _
_ _ _ _

→ user defined exceptions will not raise automatically so that we have to rise those exceptions explicitly.

Note:- creating the runtime error representation class object is known as a raising the exception.

→ we can raise the exceptions explicitly by using raise keyword.

syntax:- raise userdefinedexception.

Note:- we can also raise the pre-defined exceptions explicitly by using raise keyword.

→ After raising the exception we can handle that exception by using try and except blocks.

Ex:-
```
class Error(Exception):
    """Base class for other exceptions"""
    pass
class valueTooSmallError(Error):
    """Raised when the input value is too large"""
    pass
number = 10
while True:
try:
    i_num = input("Enter a number:")
    if i_num < number:
    raise valueTooSmallError
    elif i_num > number:
    raise valueTooLargeError
    break
    except valueTooSmallError:
    print("This value is too small, try again!")
    except valueTooLargeError:
    print("this value is too large, try again!")
    print("congratulations! you guessed it correctly")
```

O/P:- Enter a number : 8
This value is too small, try again!
Enter a number : 12
This value is too large, try again!
Enter a number: 10
congratulations! you guessed it correctly.

* File handling:-

→ programming languages programs memory allocation takes place in RAM area at the time of execution of the program.
→ RAM is a volatile so that after execution of the program the memory is going to be deallocated.
→ programming languages programs are good at processing the data but they can not store the data in permanent manner.
→ After processing the data we have to store the data in permanent manner.
→ If the data is stored in permanent manner we can use the data whenever we want.
→ we can store the data in permanent manner by using files.
→ file is a named location on disk, which is used to store the related data in permanent manner in the hard disk.
→ By using python program we can put the data into the file, we can get the data from the file and we can modify the data of the file.
→ Before going to read (or) write (or) modify the data of the file we have to open the file.
→ we can open the file by calling one predefined function called open
→ At the time of opening the file we have to specify the file modes
→ mode of the file indicates what purpose we are going to open the file.
→ python supports following file modes
r - open a file for reading (default)
w - open a file for writing create a new file if it does not exist (or) truncates the file if it exists.

x - open a file for exclusive creation. If the file already exists the operation fails.

a - open for appending at the end of the file without truncating it. creates a new file if it doesn't exist.

t - open in textmode (default)

b - open in binary mode

+ - open a file for updating.

Syntax:-

x = open("filename", "modes of the file")

→ after executing the open function it will create file object with the specified modes.

→ By calling the methods on the file object we can read the data from the file, we can write the data into the file and update the data of the file.

→ After performing the operations on the file object we have to close that file object.

→ we can close the file object by calling close method.

* write a python program to read the data from the file

```
x = None
try:
    x = open("test.txt")
    print "file is opened"
    print x
    data = x.read()
    print data
except:
    print "error occured"
finally:
    if x:
        x.close()
        print "file is closed"
```

o/p:- file is opened

<open file 'test.txt', mode 'r' at 0x01EAD198>

sathya technologies
ameerpet
hyderabad
telangana
file is closed

Ex:-
```
x = None
try:
    x = open("test.txt")
    print "file is opened"
    print x
    data = x.read(10)
    print data
    line = x.readline()
    print line
    lines = x.readlines()
    print lines
except:
    print "error occured"
finally:
    if x:
        x.close()
        print "file is closed"
```

O/P:-
```
file is opened
<open file 'test.txt', mode 'r' at 0x01E1D1A8>
sathya tec
hnologies
['ameerpet\n', 'hyderabad\n', 'telangana]
file is closed.
```

→ whenever we open the file by default file pointer points the zero location.

→ By using tell function we can know the current file pointer location.

→ By using seek() we can change the file pointer from one location to another location.

Ex:-

```
x = open('test.txt')
print x.tell()
data = x.read(3)
print data
print x.tell()
data = x.readline()
print data1
print x.tell()
x.seek(0)
print x.tell()
data2 = x.readline()
print data2
print x.tell()
x.seek(0)
print x.tell()
data3 = x.read()
print data3
print x.tell()
```

o/p:-

```
0
sathya
5
a technologies
21
0
sathya technologies
21
0
sathya technologies
ameerpet
hyderabad
telangana
51
```

* write a python program to write the data into the file.

```
x = None
try:
  x = open('demo.txt', 'w')
  x.writelines('python is a open source language \n python is
                a oopl \n python is used for data analytics')
  print 'data stored in file'
except:
   print 'error occured while writing the data into file'
finally:
   if x:
       x.close()
       print "end"
```

o/p:- data stored in file
end

* working with csv files:-

→ In order to work with the csv files we use built in module called as csv

→ Inorder to read (or) write (or) modify the data of the csv files we need to impliment following steps

1. open the file in required mode
2. create the csv file represented object
3. read (or) write (or) update the data
4. close the file

Ex:-
```
import csv
fl = open('abc.csv')
print fl
csvreader = csv.reader(fl)
print csvreader
```

```
data = list(csvreader)
for line in Data:
    for word in line:
        print word,
    print
    fi.close()
```

o/p: <open file 'abc.csv', mode 'r' at 0x0B6G4A70>
<_csv.reader object at 0x0B6A4970>

4/5/2015 13:34 Apples 73
4/5/2015 3:41 cherries 85

EX:-

```
import os
import string
def replace(file, search_for, replace_with):
temp = os.path.splittext(file)[0] + ".tmp"
try:
    os.remove(temp)
except os.error:
    pass
fi = open(file)
fo = open(temp, "w")
for s in fi.readlines():
fo.write(string.replace(s, search_for, replace_with)
fi.close()
fo.close()
file = "samples/sample.txt"
replace(file, "sathya", "lokesh")
```

Functions

Functions :

Functions is a syntax or structure, in which we represent the reusable bussiness logic.

Functions will not be Executed automatically.

Functions will be executed when ever we makes a function call.

We can call one function for 'n' number of times.

python supports two types of functions. They are

i) Built-in functions.

ii) user defined functions.

Built-in functions :

The functions which comes along with python software are known as pre defined functions (or) Built-in functions.

Eg: id()
input()
raw-input()
type()
len() ...

user defined functions :

The functions which are developed by the programmers Explicitly according to their business requirements are called user defined functions.

We can define the user defined functions by using following syntax.

Syntax:

```
def function-name (parameters):
    """
        doctstring
    """
    -----
    -----
    -----
    return value (optional)
```

Note: With in the function syntax, parameters and return statements are optional.

Eg:

```
print "hye"
def f1():
    print ("welcome")
    f1()
    f1()
Print ("bye"
```

O/p:

```
hye
welcome
welcome
bye
```

Parameters:

Parameters indicates input of the function i.e., if any function contains parameters at the time of calling that function we have to pass values to those parameters otherwise, we will get Error.

Eg:

```
def greet(name):
    """
        This function greets to the person passed in
    the parameters """
    print("Hello, "+name" Good mrng!")
    greet('lokesh')
    greet('sathya')
```

O/p:

Hello lokesh. Good mrng!

Hello sathya. Good mrng!

We can define n' numbers of parameters to a function.

Eg:

```
def add(a,b):
    print "sum of", a, "and", b, "is:", a+b
    add(100, 200)
    add(123, 456)
```

O/p:

Sum of 100 and 200 is : 300

Sum of 123 and 456 is : 579

Return Statement:

Return statement indicates what data is going to be given after execution of that fun is over.

If the function doesn't contain return statement then by default it returns none

If function contain return statement we can use the return value by storing that value into the other variable.

Eg:

```
def add(a,b)
    return a+b
x = add(a,b)
      100 200
print x.
```

o/p: 300

Eg:

```
def m1():
    print "hi"
print m1()
```

o/p: hi
None

Types of arguments:

In python language for functions we can pass the four types of arguments.

1) Required arguments
2) Default arguments
3) Key word arguments
4) Arbitary arguments

1. Required arguments:

```
def f1(a,b):
    print a+b
f1(10,20)
```

2. Default arguments:

```
eg: def f1 (a = "good morning"):
        print "hello satya", a
    f1()
    f1 ("good evening")
```

olp: Hello satya, good morning
Hello satya, good evening.

eg:

```
def f1 (a, b = "good morning"):
    print "hello", a, b
    f1 ("sathya")
    f1 ("lokesh")
    f1 ("susmitha", "good evening")
```

olp:
Hello sathya good morning
Hello lokesh good morning
Hello susmitha good evening

After default arguments we are not allowed to declare non default arguments

```
eg: def f1 (a = "sathya", b):
        print "hello", a, b
    f1 ("susmitha", "good evening")
```

olp:
Error msg.

3. Keyword arguments:

```
def f1 (name, msg)
    print "hello", name, msg
f1 (msg = "good evening", name = "sumith")
f1 (name = "sathya", msg = "good morning")
f1 ("lokesh", "good morning")
f1 ("good morning", "likitha")
```

o/p: Hello sumitha good evening
Hello sathya good morning
Hello lokesh good morning
Hello good morning vikitha

Eg:
```
def f1(name, msg):
    print "Hello", name, msg
f1("sumitha", msg = "good mrng")
```
↓ Non keyword ↓ keyword

o/p
Hello sumitha good mrng.

f1(msg = "good morning", "sumitha")

o/p error msg

Non keyword argument should not be preceded by keyword arguments.

3. Arbitrary arguments:

We can pass n number of values.

```
def greet(*names)
    print names
    for p in names
    print p
greet("sathya", "susmitha", "lokesh")
greet(10,20,30,40)
```

o/p: ("sathya", "susmitha", "lokesh

sathya
susmitha
lokesh
(10,20,30,40)
10
20
30
40

Global variables:

```
a=100
def f1():
    print a
def f2()
    print a
f1()
f2()
```

The variables which are declared outside of the function are said global variable.

Global variables of one module can be used throughout the module.

Local variables:

The variables which are declared with in a function are known as local variables.

local variables of one function cannot be accessed outside of that function

eg:

```
def f1():
    a=100
    print a
def f2():
    print a
    a=200
    f1()
    f2()
```

Recursive function calling: the concept of calling one function by the same function is known as a recursive function calling.

eg:

```
def recur_fact (x):
    """
    This is a recursive function to find the factorial of
    an integer """
    if x == 1
        return 1
    else
        return (x * recur_fact (x-1))

num = input ("Enter a number")
if num >= 1:
print ("The factorial of", num, "is", recur_fact (num))
```

o/p:

Enter the number : 5

'The factorial of', 5, 'is', 120

Infinite recursion is not supported by python.

eg:

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

o/p:

```
in h1
in h1
.
.
.
```

It will terminate with an error.

Lambda functions:

A function which doesn't contain any name explicitly is known as lambda functions. or anonamus functions.

Syntax:

lambda arguments : Expression

A lambda function can contain 'n' number of arguments.

A lambda function can contain only one expression.

After executing lambda function it will returns the expression value

Eg: double = lambda x : x * 2

print double (5)

The above lambda function is equal to the following code.

Eg:

```
def double (x):
    return x*2
def double (x):
    return x**2
double (10)
```

generally we use the lambda functions when ever we want to pass one function as parameters to another function

Eg: filter function.

```
my-list [1,2,3,5,6,8]
even-list = filter (lambda x : (x%2 == 0), my-list)
print even list
```

```
odd_list = filter(lambda x: (x%2 != 0), my list)
print(odd_list)
```

olp: [2, 6, 8]

[1, 3, 5]

MODULES:

Every python file itself is known as a module. A module can contain variables, function and classes.

In order to use the properties of one module into another module we use the 'import' statment.

Eg 1: mod1.file

```
x = 100
def f1():
    print "in f1 of mod1"
def f2():
    print "in f2 of mod1"
```

mod2.file

```
import mod1
print x
print mod1.x
mod1.f1()
mod1.f2()
print "end"
```

olp: hi

100

in f1 of mod1

in f2 of mod 2

end

Note:

Whenever we import one module into another module then imported module compiled file will be generated

and stored that file into computer hard disc permenently. After generating the compiled file for python module without having the .py file of that module we can import module into other module.

Renaming a module at the time of input:

Whenever we import a module to access the properties of that module complsry we have to use module name . property name.

Ex: mod3.py

```
def add(a,b):
    return a+b
def sub(a,b):
    return a-b
def abs(a):
    if a>=0:
        return a
    else:
        return(-a)
```

mod4.py

```
import mod3 as m
sum = m.add(30,20)
print sum
sub = m.sub(30,20)
print sub
pv = m.abs(-10)
print pv
```

Olp:

```
50
10
10
```

Math module: [in built module, all properties are available]

```
import math as m
    print "the value of pi is " m.pi
```

olp: The value of pi is 3.14159265359

from ... import :

Inorder to import the specific properties of one module into another module we use from ... import.

eg: mod6.py

```
from math import pi, e
print "The value of pi is", pi
print " The value of e is", e
```

olp:

The value of pi is 3.14159265359

The value of e is 2.17182

whenever we import the properties of one module into another module by using from import then we can access the properties of that module directly without using module name or alias name.

* import :

* import is used to import all the properties of the module.

eg:

```
from math import *
print ("The value of pi is", pi)
print ("The value of e is", e)
```

olp: 'the value of pi is', 3.141592653589793

'the value of e is', 2.718281828459045

for every module by default some properties are added automatically.

[-builtins, --doc, --file, --package, --name]

eg:
```
x=1000
def f1()
  print "in f1 of mod8"
print -builtins-
print -doc-
print -file-
print -package-
print -name-
f1()
print x
```

olp:
```
<module '_builtin_' built-in>
None
C:\Python27\mod8.py
None
_main_
in f1 of mod8
1000
```

Dir functions:

Dir function is used to know the properties of the current module or specified or specified module.

eg:
```
x=100
y=200
def f1():
  print "in f1 of mod9"
print dir()
```

olp:
```
['_builtins_', '_doc_', '_file_',
'_name_', '_package_',
't_', 'path', 'pi', 'x', 'y']
```

eg: from math import *

from os import pi

x = 1000

y = 2000

def f1():

print "in f1 of mod q"

print dir()

o/p: ['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'pi', 'path', 'f1', 'x', 'y']

Module search path:

Whenever we import a module in python, python interpreter will search for that module in the following locations.

1) current directory
2) python path
3) installation dependent default directory.

Note:

We can find out the python path by using path variable of sys module.

from sys import *

print path.

Reloading a module:

Whenever we import one module more than ones then by default that module will be imported only ones.

eg: file1: mod10.py

```
print "begining of mod10"
print "hi"
print "bye"
print "end of mod10"
```

o/p:

```
begining of mod10
hi
bye
end of mod10
```

file2: mod11.py

```
import mod10
import mod10
import mod10
import imp
imp.reload(mod10)
```

whenever we want than we can reload the module by calling reload function of imp module.

Ex:

```
import mod10
import mod10
import mod10
import imp
imp.reload
```

o/p:

```
beging of mod10
hi
bye
end of mod10
begining of mod10
hi
bye
End of mod10
```

1)

Eg:

```
import os
# where are we ?
cwd = os.getcwd()          (current directory)
print "1", cwd

# go down
os.chdir("samples")
print "2", os.getcwd()

# go backup
os.chdir(os.pardir) - previous directory
print "3", os.getcwd()
```

Olp:

```
C:\Python\os module
C:\Python\os module\samples
C:\Python\os module
```

2) Eg:

```
import os
samp = os.listdir("samples")
for x in samp:
    print x
```

Olp:

```
sample.tmp
sample.txt
subsamp
```

3)

Eg:

```
import os
os.makedirs("test3\multiple\levels")
fp = open("test3\multiple\levels\myfile", w)
```

```
fp.write("Sathyatech")
fp.close()
# Remove the file
os.remove("tests\multiple\levels\myfiles")
# and all empty directories above it
os.removedirs("tests\multiple\levels")
```

Ex:4

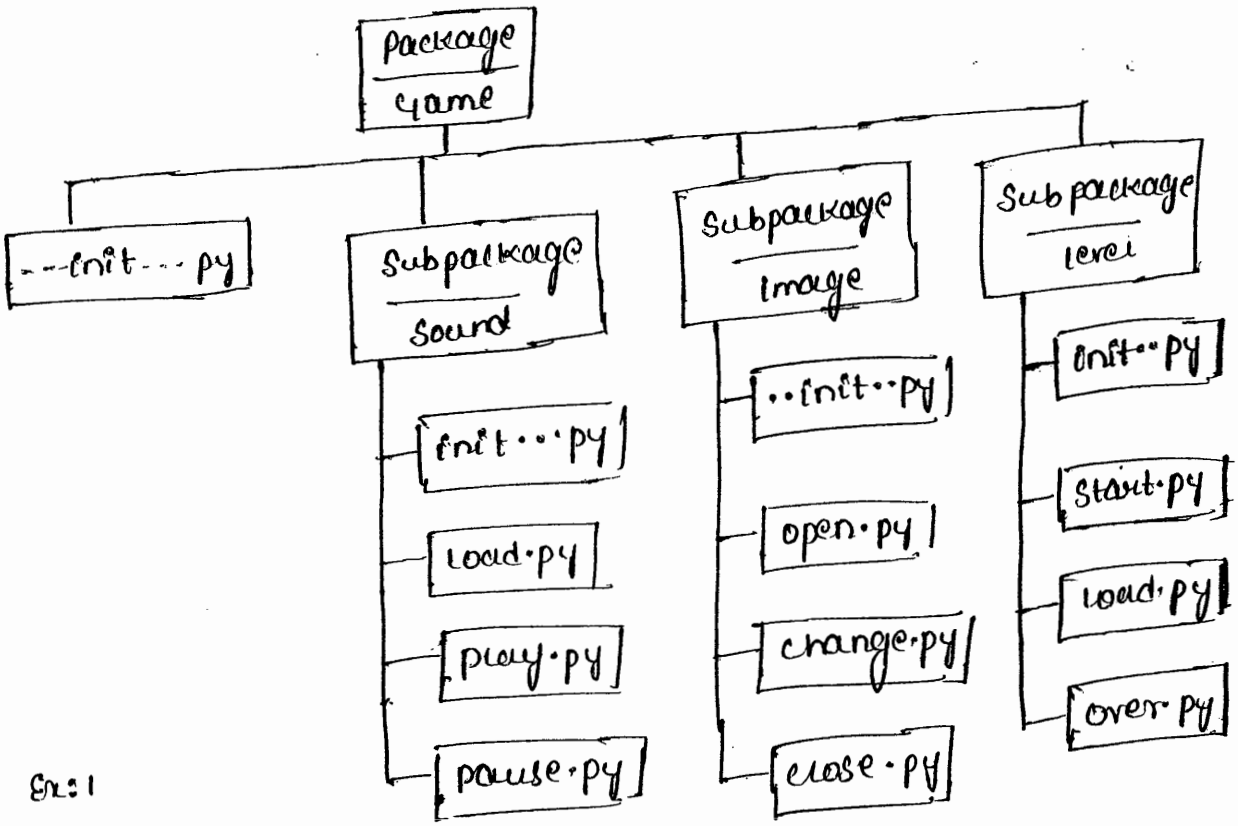
```
import os
if os.name == "nit"
    command = "ip config"
else
    command = "ls -l"
os.system(command)
```

Packages:

Packages is nothing but folder or directory which represents collection of python modules.

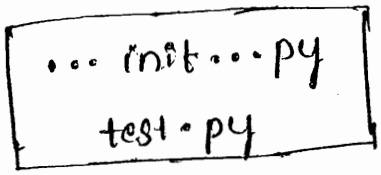
Any floder / directory which contains "__init__.py" file. Then we call that folder / directory as a 'python package'.

A package can contain subpackages.



Ex:1

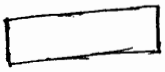
c:| python 27 > satya



file1 :- test.py

```
def greet()
print "welcome to satyatech"
```

c:| python 27 > demo.py



→ import satya.test

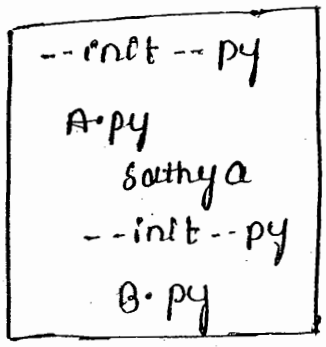
satya.testgreet

o|p: welcome to satyatech

Ex2:

c:\python 2#>

com



c:\python 2#> test.py

A.py

```
def m1():
    print "in m1 of A of com"
```

B.py

```
def m2():
    print "in m2 of B of com.sathya"
```

test.py

```
import com.A
import com.sathya.B
com.A.m1()
com.sathya.B.m2()
```

o/p:

in m1 of A of com

in m2 of B of com.sathya

④ More Avinash M.
✆ 07875175313

ADVANCED PYTHON:

oop's principles:

→ oops are the rules (or) guidelines which are supposed tobe satisfied by any programming language in order to call that programming language as oopL.

→ Different oops principles are

1. Encapsulation
2. polymorphism
3. Inheritance
4. Abstraction.

1. Encapsulation:

→ The concept of binding (or) grouping related data members along with its related functionalities is known as encapsulation.

customer

```
cname, cadd, cacno, cbal ...
def deposit(self):
    ===
def withdraw(self):
    ===
def transfer(self):
    ===
```

Employee

```
ename, eadd, eid, esal, ...
def da(self):
    ===
def hra(self):
    ===
def tax(self):
    ===
```

class:

→ class is a syntax (or) structure used to bind (or) group the related data members alongwith its related functionalities.

syntax:

```
class classname:
    - - - -
    - - - -
    - - - -
    - - - -
```

same space indentation called suit/block/class

→ within the class we can represent data by using static variables and non static variables.

Static variables (class variables):

→ The variables which are declared within the class, outside of all the methods are known as static variables (or) class variables.

→ The data which is common for every object is recommended to represent by using static variable

→ For all the static variables of a class, memory will be allocated only once

→ static variable of one class we can access within the same class (or) outside of the class by using class name

Ex:
```
class x:
    i = 1000        # static variable
    def m1(self):
        print x.i
x1 = x()    # object creation statement
x1.m1()     # 1000
print x.i   # 1000
```

→ we can modify value of a static variable

Non-static variables:

→ The variables which are declared with self (or) reference variable are known as non-static variable (or) instance variable

→ Non static variables we can define within the constructors (or) within the method.

→ The data which is separate for every object is recommended to represent by using non-static variable

→ for all the non-static variables of a class, memory will be allocated whenever we create object.

→ with respect to every object creation statement separate copy of the memory will be allocated for non-static variable.

→ Non static variable of one class, we can access within the same class by using 'self'.

→ Non-static variables of one class we can access outside the class by using reference variable name.

Ex:

```
class y:
    def m1(self):
        self.i = 100
        self.j = 123.123
    def display(self):
        print self.i
        print self.j

y1 = y()
y1.m1()
y1.display()
y1.i = 200
y1.j = 234.234
y1.display()
y2 = y()
y2.m1()
y2.display()
y2.i = 300
y2.j = 345.345
y2.display()
y1.display()
```

⟹

o/p:

```
100
123.123
200
234.234
100
123.123
300
345.345
200
234.234
```

* constructor:

→ It is a special kind of method which executes automatically whenever we create object. (i.e., without calling it)

→ constructors are used to initialize the non-static variables of the class at the time of creation of object with the user defined value

Ex:

```
class Z:
    def __init__(self):
        print "in const of Z"

Z1 = Z()
```

⇒ o/p: in const of Z

Ex-2:

```
class Z:
    def __init__(self, i, j):
        self.i = i
        self.j = j

Z1 = Z(100, 12.12)
print Z1.i
print Z1.j
Z2 = Z(100, 23.23)
print Z2.i
print Z2.j
```

⇒ o/p: 100
12.12
100
23.23

* object:

→ The concept of allocating memory space for non-static variable of a class at run time dynamically is known as an object

→ we can create 'n' no-of objects for one class.

→ After creating the object, address of object will be stored in variable and we call that variable as a reference variable

→ From that variable we can put the data into object,

we can get data and also call the methods on that object.

Ex:

```
def display():
    print "hi"
class x:
    def __init__(self, i, j):
        self.i = i
        self.j = j
    def display(self):
        print self.i
        print self.j
x1 = x(100, 200)
print x1
print id(x1)
print hex(id(x1)).upper
x1.display()
x2 = x(200, 300)
print x2
print hex(id(x2)).upper
x2.display()
display()
```

o/p:

```
<__main__.x instance at 0x01FD0B48>
33360712
<built-in method upper of str object at 0x01EEEFE0>
100
200
<__main__.x instance at 0x01FD6CB0>
<built-in method upper of str object at 0x01EEEFE0>
200
300
hi
```

Ex:

```
class InsufficientFunds(Exception):
    """ bal not available """

class cust:
    cbname = "sbi"
    def __init__(self, cname, cadd, cacno, cbal):
        self.cname = cname
        self.cadd = cadd
        self.cacno = cacno
        self.cbal = cbal
    def deposit(self, damt):
        self.cbal = self.cbal + damt
    def withdraw(self, wamt):
        raise InsufficientFunds
        self.cbal = self.cbal - wamt
    def display(self):
        print self.cname
        print self.cadd
        print self.cacno
        print self.cbal
        print cust.cbname
c1 = cust('ravana', 'srilanka', 1001, 100000.00)
c1.display()
c1.deposit(10000.00)
try:
    c1.withdraw(10000.00)
except:
    print "insufficient funds in your account"
c1.display()
c2 = cust('sitha', 'india', 1002, 1000.00)
c2.display()
c2.deposit(1000.00)
```

```
try:
    ca.withdraw(4000.00)
except:
    print "insufficient funds in your account"
ca.display()
ca.deposit(1000.00)
```

O/P:

```
ravana
srilanka
1001
100000.0
sbi
ravana
srilanka
1001
100000.0
sbi
sitha
india
1002
1000.0
sbi
Insufficient funds in your account
sitha
india
1002
2000.0
sbi
```

* Adding static and non-static variables to the class and object from outside of class.

Ex:

```
class demo:
    x = 100
    def __init__(self):
        self.y = 2000
```

```
        self.z = 3000

demo.p = 4000
print demo.x
print demo.p
d1 = demo()
d1.q = 5000
d1.r = 6000
print d1.Y
print d1.Z
print d1.q
print d1.r
d2 = demo()
print d2.Y
print d2.Z
```

o/p:

```
100
4000
2000
3000
5000
6000
2000
3000
```

* Removing static and non-static variables from class and object:

```
class demo:
    x = 1000
    def __init__(self):
        self.Y = 2000
        self.Z = 3000
print demo.x
del demo.x
d1 = demo()
print d1.Y
print d1.Z
del d1.Y
del d1.Z
d2 = demo()
print d2.Y
print d2.Z
```

o/p:

```
1000
2000
3000
2000
3000
```

→ If static variable is removed then it is removed for all objects of that class

→ but non-static variable is removed only for that particular class.

* Default attributes (or) namespaces:

→ w.r.t every class some attributes are added automatically and we call those attributes as a default attributes.

→ The default attributes which are added to each and every class are.

i. ___dict___
ii. ___doc___
iii. ___name___
iv. ___module___
v. ___bases___

Ex:

```
class demo:
    """ test prog """
    a = 1000
    def __init__(self):
        self.b = 200
print demo.a
print demo.__name__
print demo.__module__
print demo.__dict__
print demo.__bases__
```

O/P:

```
1000
demo
__main__
{'a': 1000, '__module__': '__main__', '__doc__': 'test prog',
 '__init__': <function __init__ at 0x020H0D70>}
```

* Garbage collection:

→ The concept of removing the unused, unreferenced object from the memory location is known as a Garbage collection.

→ once garbage collection is over memory will become free and rest of the program execution takes place in faster manner

→ There are two types of garbage collection supported by python they are

1. automatic garbage collection
2. explicit garbage collection

1. Automatic garbage collection:

→ After starting execution of program periodically garbage collector program runs internally

→ whenever garbage collector program is running if any unused unreferenced objects are available in memory location then it will remove those objects from memory location.

→ whenever any object is going tobe removed from memory location then distructor of that class is going tobe executed.

→ In distructor we write the resource deallocation statement.

Ex:

```
class x:
    a = 1000
    def __init__(self):
        print " in constructor"
    def __del__(self):
        print " in distructor"

x1 = x()
x1 = x()
```

o/p: in constructor
in constructor
in distructor

2. Explicit G.C:

- The concept of executing the garbage collection program explicitly whenever we required is known as explicit garbage collection.

→ By using 'del' keyword we can run garbage collector explicitly.

eg:
```
# class Demo:
    a = 1000
    def __init__(self):
        print "in constructor"
    def __del__(self):
        print "in distructor"

d1 = Demo()
del d1          # explicit garbage c.
```
O/P:
```
in constructor
in distructor
```

eg:
```
# class x:
    a=1000
    def __init__(self):
        print "in constructor"
    def __del__(self):
        print "in distructor"

x1 = x()
x2 = x
x3 = x
del x3
del x2
del x1
```
separate object is not created but the address of x_1 is given to x_2 or x_3

O/P:
```
in constructor
in distructor
```

* Inheritance:

The concept of using properties of one class into another class without creating object of that class explicitly is known as a inheritance.

→ A class which is extended by another class is known as 'super class'.

→ A class which is extending another class is known as 'sub class'.

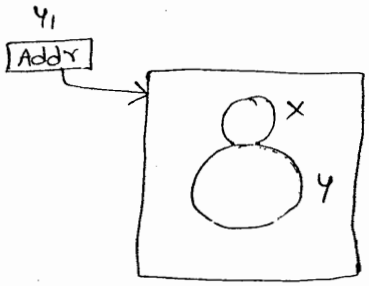
→ Both super class prop and sub class prop can be accessed through subclass ref. variable.

```
# class x:
    def m1(self):
        self.i=1000

class y(x):
    def m2(self):
        self.j = 2000

y1=y()
y1.m1()
y1.m2()
print y1.i
print y1.j
```

o/p:

1000

2000



→ super class properties directly we can use within the subclass.

```
# class x:
    def m1(self):
        self.i=1000

class y(x):
    def m2(self):
        self.j = 2000
    def display(self):
```

```
        print self.i
        print self.j

Y1 = Y()
Y1.m1()
Y1.m2()
Y1.display()
```

O/P:

1000
2000

```
# class Parent:
    parentAttr = 100
    def __init__(self):
        print "calling parent constructor"
    def parentMethod(self):
        print "calling parent Method"
    def setAttr(self,attr):
        parent.parentAttr = attr
    def getAttr(self):
        print "parent Attribute", Parent.ParentAttr
class child(parent):
    def __init__(self):
        print "calling child constructor"
    def childMethod(self):
        print "calling child Method"

P = Parent()
P.ParentMethod()
P.getAttr(800)
P.getAttr()
C = child()
C.childMethod()          # overloading of constructor hence only
```

child class constr. will execute.

```
C.parentMethod()
C.setAttr(300)
C.getAttr()
```

O/P: calling parent constructor
calling parent Method
parent attribute : 200
calling child constructor
calling child Method
calling parent Method
parent Attribute : 300

```
# class x:
    def __init__(self):
        print "in const of x"
class y(x):
    def m1(self):
        print "in m1 of y"
y1 = y()
y1.m1()
```

O/P: in const. of x
in m1 of y

```
# class x:
    def m1(self):
        print x.__bases__
class y(x):
    def m2(self):
        print y.__bases__
```

```
y1 = y()
y1.m1
y1.m2
```

O/P: ( )

(<class __main__ x at 0x03AA998>)

* Types of Inheritances:

1. Single Inheritance:

The concept of inheriting properties from only one class into another class is known as a single inheritance.

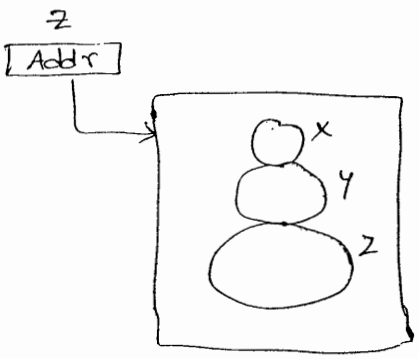
```
# class x:
    def m1(self):
        print "in m1 of x"
class y(x):
    def m2(self):
        print "in m2 of y"
y1 = y()
y1.m1()
y1.m2()
```

O/P: in m1 of x
in m2 of y

2. Multilevel Inheritance:

The concept of Inheriting prop. from multiple classes into single class with the concept of "one after another" is known as a multilevel inheritance.

```
# class x :
    def m1(self):
        print "in m1 of x"
class y (x):
    def m2(self):
        print "in m2 of y"
class z(x):
    def m3(self):
        print "in m3 of z"
z1 = z()
z1.m1()
z1.m2()
z1.m3()
```



O/P: in m1 of x
in m2 of y
in m3 of z

3. Hirorchial Inheritance:

The concept of inheriting prop from one class into multiple classes is known as a heirarical inheritance.

```
# class x :
    def m1(self):
        print " in m1 of x"
class y(x):
    def m2(self):
        print "in m2 of y"
class z(x)
    def m3(self):
```

```
        print "in m3 of z"

y1 = y()
y1.m1()
y1.m2()
z1 = z()
z1.m1()
z1.m3()
```

o/p:

```
in m1 of x
in m2 of y
in m1 of x
in m3 of z
```

4. Multiple Inheritance:

The concept of inheriting prop. from multiple classes into single class at a time is known as a multiple inheritance.

```
# class x:
    def m1(self):
        print "in m1 of x"
class y:
    def m2(self):
        print "in m2 of y"
class z(x,y):
    def m3(self):
        print "in m3 of z"
```

```
z1 = z()          |  y1 = y()
z1.m1()           |  y1.m2()
z1.m2()           |  x1 = x()
z1.m3()           |  x1.m1()
```

o/p:

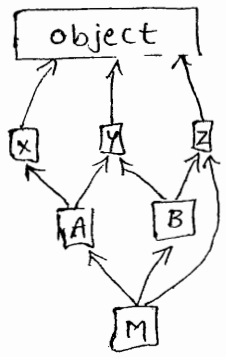
```
in m1 of x
in m2 of y
in m3 of z
in m2 of y
in m1 of x
```

5. Cyclic Inheritance:

The concept of inheriting the prop. from subclass to superclass is known as a cyclic inheritance.

→ python doesn't support cyclic inheritance.

```
# class x(z):
    def m1(self):
        print "in m1 of x"

class y(x):
    def m2(self):
        print "in m2 of y"

class z(y):
    def m3(self):
        print "in m3 of z"
```



* Object:

→ object class is a predefined class.

→ object class is super class for every class in python

→ If any class is not extending other class then by default it extends

```
# print object.__name__
  print object.__doc__
  print object.__dict__
  print object.__sizeof__
```

3. Polymorphism:

→ Poly means many and morphism means forms

→ Forms means functionalities or logics.

→ The concept of defining multiple logics to perform same operation is known as a polymorphism.

→ polymorphism can be implemented in python by using method overriding.

Note: Python does n't support the method overloading.

* Method overloading:

The concept of defining multiple methods with the same name, diff no. of parameters with in a same class is known as a method overloading.

```
# Class X:
    def m1(self, a):
        print "in one param m1 of x"
    def m1 (self, a, b):
        print "in two param m1 of x"

x1 = X()
x1.m1(100, 200)
x1.m1(100)
```

O/P: in two param m1 of x

error.

```
# class Y:
    def m1(self, *args):
        for p in args:
            print p

Y1=Y()
Y1.m1(10, 20)
Y1.m1(100, 200, 300)
Y1.m1('sathya', 'hyd', 'amp')
```

o/p:

```
10
20
100
200
300
sathya
hyd
amp
```

```
# class x:
    def add(self, instance of, *args)
        if instance of == 'int':
            result = 0
        if instance of == 'str':
            result = ''
        for i in args:
            result = result + i
        print result

x1 = x()
x1.add('int', 10, 20, 30)
x1.add('str', 'sathya', 'tech')
```

(*) This is not method overloading

o/p:

```
60
Sathyatech
```

* Method overriding:

The concept of defining multiple methods with the same name with the same no. of parameters. one is in superclass and another in sub class is known as method overriding.

```
# class parent :
    def myMethod(self):
        print 'calling parent method'
class child (Parent):
    def myMethod (self):
        print 'calling child method'
c = child()
c.myMethod()
P = parent()
P.myMethod()
```

O/P:

calling child.method
calling parent method

super() → when method overriding

```
# class Parent (object):
    def altered (self):
        print "Parent altered"
class child (Parent):
    def altered(self):
        print "CHILD. Before Parent altered"
        super (child, self). altered()
        print "CHILD. After Parent altered"
dad = Parent()
son = child()
dad. altered()
son. altered()
```

O/P: Parent altered

CHILD. Before Parent altered

Parent altered

CHILD. After parent altered

4. Abstraction (or) Data Hiding:

The concept of hiding the prop. of one class from the other classes or other programs directly is known as data-hiding or Abstraction. (__ is prefix)

eg:

```
# class x:
    __p = 1000
    def m1 (self)
        print "in m1 of x"
x1 = x()
x1.m1()
print x.p
```

O/P:
in m1 of x
error (p not found)

```
# class JustCounter:
    __secretcount = 0
    def count (self):
        self.__secretcount += 1
        print self.__Secretcount
counter = JustCounter()
counter.count()
counter.count()
print counter._JustCoustmer__Secretcount
```

After data hiding the variable is not static or non-static it just get hided.

(5) More Avinash M.
(8) 07875175313

* Regular Expression:

→ Regular expressions are used to extract the required information from the given data by following patterns.

→ Regular expressions are also used to check whether the given input data is in proper format or not.

eg: email Id verification, mobile no. verification
↳ @ is there or not ↳ 10 digit no or not

Password verification

→ The regular expressions which are supported by the 'pearl' are also supported by the python.

→ python is providing the built-in-functions to work with the regular expression easily.

→ All the predebined functions which are related to regular expression are present in re module

* Special characters which are used in regular expression are:

1. * → It matches zero or more occurances of preceding character.

eg: ab*c

AC
Abc
Abbc
Abbbbbbc
} Valid

Bbdc # error

2. + → It matches one or more occurances of preceding character.

eg: ab+c

AC # error
Abc
Abbc
Abbbbbc

3. ? → it matches zero or one occurance of preceding character

eg: i) ab?c
ac
Abc
AbbC #error

ii) Pea?rl
perl
pearl

iii) colou?r
color
colour

4. . → it matches any single character

eg: a.c
Agc
asc
a$c
Abcd #error

5. [ ] → it matches any single character in given list.

eg: b[aeiou]d
Bad
bed
Bid
bod
b8d #error

6. [^] → it matches any single character other than in the given list.

[^xyz....] → b[^aeiou]d

eg: Bad #error
Bed #error
Bid #error
Bpd

7. [-] → it matches any single character in the given range.

x[a-e]y

eg: xay
xby
xey
xfy #error

[0-9] → any single digit

[a-z] → any one lowercase alphabet

[A-Z] → any one uppercase alphabet

[a-zA-Z] → any one alphabet

[a-zA-Z0-9] → any one alphanumeric

[^0-9] → any single non digit

[^A-Z] → any one non uppercase alphabet

[^a-z] → any one non lowercase "

[^a-zA-Z] → any one non alphabet

[^a-zA-Z0-9_] → any one non alphanumeric (special character)

8. (|) → match anyone string in the list

eg: (java | Hadoop | python)

java
Hadoop
python
.net # error

9. {m} → it matches exact occurance of preceeding character

eg: ab{3}c

⇒ abbbc

abc # error

abbc # error

Abbbbc # error

10. {m,n} → it matches min m occurance and max n occurance of preceding character

A.{3,5}C

Abbc #error

Abbbc

Abbbbc

11. {m,} → it matches minimum m occurance and max no limit of preceding character.

Ab{3,}c

Abc #error

Abbbbbc

12. ^ → start of the line

eg: ^perl

^[abc]

^[^abc]

13. $ → end of the line

perl $

[0-9] $

14. \d or [0-9] → any single digit

eg: [0-9][0-9][0-9][0-9] or [0-9]{4} or \d\d\d\d or \d{4}

15. \D or [^0-9] → any single non digit

16. \w or [a-zA-Z0-9_] → any alphanumeric

17. \W or [^a-zA-Z0-9_] → any non alphanumeric (special characters)

18. '\s' → ' ', '\t', '\n'
↓
lowercase

19. \b → word boundary

eg:

* use google for re module documentation

```
# import re
regex = r"[a-zA-Z]+\d+"
matches = re.findall(regex, "June 24, August 9, Dec 12")
for match in matches:
    print "Full match: %s" % (match)
regex = r"([a-zA-Z]+)\d+"
matches = re.findall(regex, "June 24, August 9, Dec 12")
for match in matches:
    print "match month: %s" % (match)
regex = r"[a-zA-Z]+\d+"
matches = re.finditer(regex, "                    ")
for match in matches
    print "Match at index: %s, %s" % (match.start(),
                                      match.end())
```

* import re

```
regex = r"([a-zA-Z]+)(\d+)"
print re.sub(regex, r"\2 of \1", "June 24, August 9, Dec 12")
```

O/P: 24 of June, 9 of August, 12 of Dec

* `rex = r"^[\w.+-]+@[\w]+.([a-zA-Z]{2,}|com|org)*(.[a-zA-Z]{2,}|in)*$"`

```
# import re
    regex = re.compile (r"(\w+) World")
    result = regex.search ("Hello World is the easiest")
    if result :
        print result.sear(), result.end()
    for result in regex.findall ("Hello World, Bonjour World")
        print result
    print regex.sub (r"\1 Earth", "Hello World")
```

if some data is present True (or) None means false

O/P:

```
0, 11
Hell o
Bonjour
Hello earth
```

* Multithreading:

Thread is a functionality or logic which can executed simultaneously along with the other part of the program.

→ we can define the functionality or logic as a thread by overriding 'run' method of thread class.

→ 'Thread' is a predefined class which is defined in 'threading' module.

→ After defining the functionality or logic as a thread we can execute that logic or functionality as a thread by calling 'start()' method of thread class.

eg:

```
# import threading
class Thread1 (threading.Thread):
    def run (self):
        i=1
        While i<=100:
            print i
            i=i+1
class Thread2 (threading.Thread):
    def run (self):
        i=101
        While i<=200:
            print i
            i=i+1

t1 = Thread1()
t2 = Thread2()
t1.start()
t2.start()
```

* Scheduling:

Among multiple threads which thread has to start the execution first how much time that thread has to execute, after allocated time is over which thread has to continue the execution next is comes under scheduling.

→ scheduling is a dynamic process.

```
# import threading
import time
exitFlag=0
class myThread (threading.Thread):
```

```
def __init__ (self, threadId, name, counter):
    threading.Thread.__init__(self)
    self.threadId = threadID
    self.name = name
    self.counter = counter
def run(self):
    print "Starting" + self.name
    print_time (self.name, self.counter, 5)
    print "exiting" + self.name
def print_time (threadName, delay, counter):
    while counter:
        if exitflag:
            threadName.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1
thread1 = myThread(1, "Thread1", 1)
thread2 = myThread(2, "Thread-2", 2)
thread1.start()
thread2.start()
print "Exiting main Thread"
```

* Synchronisation:

The concept of avoiding multiple threads to access the same functionality at a time is known as a synchronisation.

→ synchronisation we apply by calling acquire() method and release() methods of thread lock.

```
# import threading
  import time
  class MyMethod (threading.Thread):
      def __init__ (self, threadID, name, counter):
          threading.Thread.__int__(self)
           self.threadID = threadID
           self.name = name
           self.counter = counter
      def run(self):
          print "Starting" + self.name
           threadLock.acquire()
           print_time (self.name, self.counter, 3)
           threadLock.release()
      def print_time (threadName, delay, counter):
          while counter:
              time.sleep (delay)
              print "%s, %s" % (threadName, time.ctime (time.time()))
               counter -= 1
  threadLock = threading.Lock()
   threads = [ ]
   thread1 = MyThread (1, "Thread-1", 1)
   thread2 = MyThread (2, "Thread-2", 2)
```

```
thread1.start()
thread2.start()
threads.append(thread1)
threads.append(thread2)
for t in threads:
    t.join()
print "Exiting main thread"
```

* Old Version Multithreading:

```
# import thread        # note thread was module
import time
def print_time (threadName, delay):
    count = 0
    while count < 5:
        time.sleep(delay)
        count += 1
        print "%s, %s" % (threadName, time.ctime(time.time()))
try:
    thread.start_new_thread(print_time, ("Thread-1", 2))
    thread.start_new_thread(print_time, ("Thread-2", 4))
except:
    print "Error: unable to start thread"
while 1:
    pass
```

* Python – Data Base Communication:

Programming languages are good at processing the data but they can't store the data in permenant manner because programming languages memory will be allocated in the RAM.

→ RAM is a Volatile i.e., once the program execution is over the data which is present inside the RAM will be deleted.

→ In order to store the data in permenant manner we can use the files (or) data bases.

→ Data bases are nothing but softwares which stores the data in hard-disk in permenant manner & provides the security for the data.

Different databases are

1. oracle
2. My SQL
3. SQL Server
4. DB2
5. postgre SQL- - - -

→ Download the oracle 11g enterprise edition from oracle website. (www.oracle.com)

→ extract the downloaded jar files into a single folder.

→ open the data base folder click on setup file.

→ deselect the check box called i wish to receive security updates via my oracle support.

→ click on Next

→ click on Yes.

→ click on Next

→ click on Next

Note: Default global database name for oracle 11g enterprise edition

is ORCL & don't change the global database name.

→ enter the administrative password as a sathya123.

→ enter the confirm password as sathya123.

→ click on Next.

→ click on Yes.

→ click on finish.

→ click on password management.

→ unlock the scott user, enter the password as tiger & enter the confirm password as tiger.

→ click on OK.

→ click on OK.

Note: Default port no. for the oracle database is 1521.

→ Inorder to provide the communication b/n oracle database to the python program we required one external module called

cx_Oracle

→ we can install the external modules by using PIP.

→ PIP is a application which will be installed automatically along with the python software.

→ C:\Python27\Scripts > PIP install cx_Oracle.

→ whenever we run the above program if any error is occured then upgrade the pip by using upgrade command, download & install the microsoft visual c++ compiler for python 2.7 & run the above command once again.

export ORACLE_HOME = db_Home

PATH = $PATH:

*oracle support password avi123

* global database name - orcl

administrative password - Avimore123

* establishing the connection with the database:

We can establish the connection with the oracle database by calling connect function of cx_Oracle module.

db = cx_Oracle.Connect ("dbusername", "dbpwd", "ip address of the computer where database is installed : portno/database service name")

eg:
```
import cx_Oracle
db = cx_oracle.connect ("scott", "tiger", "localhost :1521/orcl")
print db
if db == None :
    print "connection is not established"
else :
    print " connection is established"
db.close()
print db
print "end"
```

O/P: <cx_oracle.connection to scott@localhost

eg:
```
import cx_Oracle
db = None
try :
    db = cx_oracle.connection ("scott", "tiger123", "localhost :1521/orcl")
except (cx_oracle.Database error):
    print "invalid user name /password"
    print db
if db == None :
```

```
print " connection is not established"
else:
    print " connection is established"
if db != None:
    db.close()
    print " connection is not established"
    print "end"
```

O/P: Invalid user
None
connection is not established
end

* Executing the Queries:

In order to execute the SQL queries, first we have to get the cursor object.

→ We can get the cursor object by calling cursor() method of connection object.

```
cursor = conn.cursor()
```

→ After getting the cursor object, we can execute the SQL queries by calling execute method of cursor object.

```
cursor.execute('SELECT * FROM dept')
```

→ We can display the records from the cursor object by using for loop.

eg:
```
import cx_Oracle
conn = cx_oracle.connect("scott", "tiger", "localhost:1521/orcl")
cursor = conn.cursor()
cursor.execute('SELECT * FROM dept')
```

```
for row in cursor:
    print row
cursor.close()
conn.close()
```

O/P: (10, 'ACCOUNTING', 'NEW YORK')

(20, 'RESEARCH', 'DALLAS')

(30, 'SALES', 'CHICAGO')

(40, 'OPERATIONS', 'BOSTON')

SQL > select empno, ename, sal, d.deptno, dname, loc

eg:

```
import cx_Oracle
conn = cx_Oracle.connect("scott", "tiger", "localhost:1521/orcl")
cursor = cursor.execute('SELECT empno, ename, sal, d deptno, dname,
                         loc from emp e, deptd where e.deptno =
                         d.deptno')
for row in cursor:
    for col in row:
        print col
cursor.close()
conn.close()
```

O/P:

7782	CLERK	2450.0	10	Accounting	NewYork
7839	KING	5000.0	10	"	"
—	—	—	—	—	—
—	—	—	—	—	—

eg: import cx_oracle

```
db = cx_oracle.connect('scott', 'tiger', 'localhost:1521/orcl')
cursor.execute ('SELECT empno, ename, sal FROM emp order by sal
                                                    decrement')
row = cursor.fetchone()
print row
row = cursor.fetchone()
print row
cursor.close()
db.close()
```

O/P: (7839, 'KING', 5000.0)
(7902, 'FORD', 3000.0)

eg: import cx_oracle

```
db = cx_oracle.connect('scott', 'tiger', 'localhost:1521/orcl')
cursor = db.cursor()
Cursor.execute ('SELECT empno, ename, sal FROM emp order by sal desc')
rows = cursor.fetchmany(3)
for row in rows:
    print row
cursor.close()
db.close()
```

O/P: (7839, 'KING', 5000.0)
(7902, 'FORD', 3000.0)
(7788, 'SCOTT', 3000.0)

* Improving the query performance by changing array size of the cursor:

Whenever cursor object is created then by default cursor array size is set to 50 . i.e., at a time 50 records of the table will be stored into the cursor object.

→ If the table contains huge no. of records then performance of the query will be ↓sed. because of less array size of cursor.

→ To improve the performance of the query we have to ↑se the array size of the cursor.

→ we can ↑se the array size of the cursor by using following syntax.

syntax: cur. arraysize = 100

→ create table bigtab (mycol varchar2(20));

→ begin

→ for i in 1----20000

→ loop

→ insert into bigtab (mycol) values

→ (dbms_random.string ('A', 20));

→ end loop;

→ end;

→ /

→ commit; It means permenantly store the data

eg: import time

import cx_oracle

con = cx_oracle.connect ('scott', 'tiger', 'localhost:1521/orcl')

start = time.time()

cursor = con.cursor()

```
cursor.arraysize = 100
cursor.execute ('select * from bigtab')
res = cursor.fetchall()
print res
elapsed = (time.time() - start)
print elapsed
cursor.close()
con.close()
```

* Bind Variables:

Whenever we execute the same query for multiple times without using bind variables then every time 3 operations will takes place at database side. They are

1. query compilation
2. query plan generation
3. query execution

→ If we send the query to the database with bind variables 1st time query compilation, query plan generation & query execution will takes place at database side.

→ from 2nd time onwards only query execution will takes place.

→ If we want to execute the same query for multiple times then it is recommended to use bind variables.

→ In order to implement the bind variables concept we use prepare method of cursor object.

eg:
```
import cx_Oracle
con = cx_Oracle.connect('scott', 'tiger', 'localhost:1521/orcl')
cur = con.cursor()
cur.prepare('select empno, ename, sal from emp where empno = :id')
```

```
cur.execute (None, {'id' : 7788})
res = cur.fetchall()
print res
cur.execute (None, {'id' : 7369})
res = cur.fetchall()
print res
cur.close()
con.close()
```

O/P: [(7788, 'SCOTT', 3000.0)]

[(7369, 'SMITH', 800.0)]

eg:

```
import cx_oracle
con = cx_oracle.connect ('scott', 'tiger', 'localhost:1521/orcl')
rows = [ (1, "first"),
         (2, "second"),
         (3, "Third"),
         (4, "fourth"),
         (5, "fifth"),
         (6, "sixth"),
         (7, "Seventh") ]
cur = con.cursor()
cur.bindarraysize = 7
cur.setinputsizes(int,20)
cur.executemany ("insert into mytab (id, data) values (:1, :2)", rows)
con.commit()
# Now query the results back
cur2 = con.cursor()
```

```
cur2.execute ('select * from mytab')
res = cur2.fetchall()
print res
cur.close()
cur2.close()
con.close()
```

O/P:

[(1, 'first'), (2, 'second'), (3, 'Third'), (4, 'fourth'), (5, 'fifth'), (6, 'sixth'), (7, 'seventh'), (1, 'first'), (2, 'second'), (3, 'third'), (4, 'fourth'), (5, 'fifth'), (6, 'sixth'), (7, 'seventh')]

* Executing PL/SQL functions:

A PL/SQL function is a named block in which we represents the reusable business logic.

→ A PL/SQL function should return the value.

→ PL/SQL function stored in the data base permenantly.

→ whenever we call a PL/SQL function then only PL/SQL function will be executed.

→ set serveroutput on

→ create table ptab (mydata varchar(20), myid number);

→ create or replace function myfunc (d-P in varchar2, i-P in number) return number as

→ begin

→ insert into ptab (mydata, myid) values (d-P, i-P);

→ return (i-P * 2);

→ end;

→ /

→ we can execute the PL/SQL functions by calling call function method of cursor object

eg:
```
import cx_oracle
con = cx_oracle.connect('scott', 'tiger', 'localhost:1521(orcl')
cur = con.cursor()
res = cur.callfunc('myfunc', cx_oracle.NUMBER, ('sathya', 1))
print res
cur.close()
con.commit()
con.close()
```

select * from ptab

it will display the result

o/p:

2.0

sathya 1

* Executing PL/SQL procedures:

A PL/SQL procedure is a named PL/SQL block in which we represent the reusable business logic.

→ A PL/SQL procedure doesn't contain return statement.

→ A PL/SQL procedure will be executed whenever we call a PL/SQL procedure

→ create or replace procedure

→ Myproc (V1_P in number, V2_P out number) as

→ begin

→ V2_P := V1_P * 2;

→ end;

→ /

→ Procedure is created

eg: import cx_Oracle

```
con = cx_oracle.connect ('scott', 'tiger', 'localhost:1521/orcl')
cur = con.cursor()
myvar = cur.var(cx_oracle.Number)
cur.callproc ('myproc', (123, myvar))
print myvar.getvalue()
cur.close()
con.close()
```

o/p:

246.0

* Web scrapping:

Connecting to the website, accessing the data from the website & performing the required operations on the data is known as web scrapping.

eg: import urllib2

```
count = 0
x = urllib2.urlopen ('http://sathyatech.com/index.php?cid=8')
for line in x:
    if 'PYTHON' in line:
        count = count + 1
    print count
```

o/p:

1