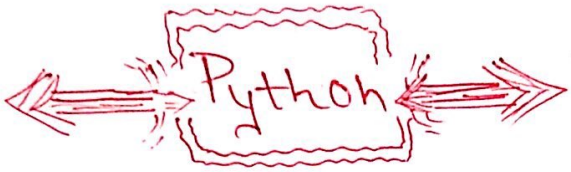


Python Features:-

* Simple & Easy to learn
* Freeware & Open source
* High level programming language
* Plateform Independent
* Portability
* Dynamically Typed.
* Both Procedure oriented & Object Oriented
* Interpreted
* Extensible
* Embedded
* Extensible Library

⇒ Dynamically Typed :-

Java :

```
a=10; //give error
class test
{
  psvm(String[] args)
  {
    a=10;
  }
}
```

Error :- Can't find the symbol
So use int a=10;

* If update the value gives error. as

```
int a=10;
a="durga" | gives error
```

python :-

```
a=10; //Take by default In
a=10.0; // float
a=600000; // double
print(type(a)); //give datatype
```

run :- py test.py

* If update variable value as override allow.

```
a=10;
print(type(a))
a="durga"
print(type(a)) //multiple
```

⇒ Python procedure oriented & Object Oriented :-

C → Procedure Oriented

Java → OOP

Python → (All Rounder) → procedure Oriented
→ OOP
→ scripting language
→ Modular

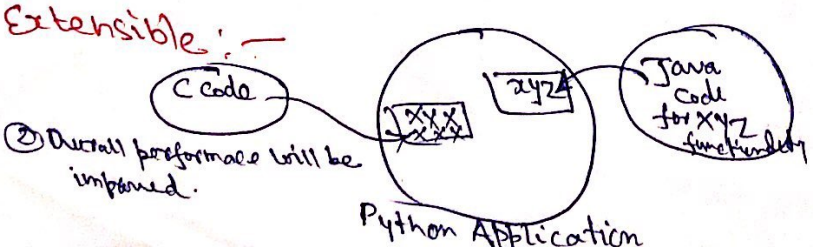
⇒ Interpreted :-

C → compile + execute

Java → compile + run

Python → Run.

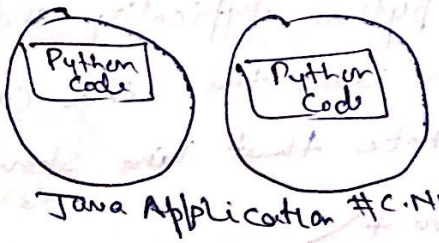
⇒ Extensible :-



② Overall performance will be improved.

① We can use Legacy Non-python code

⇒ Embeded :—



⇒ Extensive Library :—

Map to generate 6-digit OTP :-

random → randint(0,9)

Library.

test.py

```
from random import randint
print (randint(0,9),randint(0,9),randint(0,9),
                    ....(times=6,sep='')
```

Run :— by test.py

Keywords :—

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

Comment :—

Hash (#) → single line comments

Triple-quote (''' ''') → Multiline Comments

Lines and Indentation :—

Python provides no ~~bracket~~ braces to indicate block of code for class & function definitions or flow control.

*All statements within the block must be ~~indent~~ indented the same amount.

Eg :-

```
if True :
    print "True"
else
    print "False"
```

```
if True:
print "Answer"
print "True"
else :
```

→ Error

Multi Line Statements: —

* Statements in python typically end with a new line. python use the line continuation character (\) to denote that line should continue.

```
total = item_one +\
        item_two +\
        item_three
```

Multiple Statements on a Single Line: —

The semicolon (;) allows multiple statements on the single line. —

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Multiple Statement Group as Suites: —

A group of individual statements which make a single code block are called suites in python. Compound or complex stmt, such as if, while, def, and class requires a header § line and a suit.

```
if expression :
    Suit
elif expression :
    Suite
else :
    Suite
```

header line begin with (keyword) and terminate with a colon (:) and followed by one or more lines which make up suit....

Variable: —

```
Counter = 100   # int
miles = 1000.0  # float
name = "John"   # string
print counter
print miles
print name
```

Output :- 100
1000.0
John

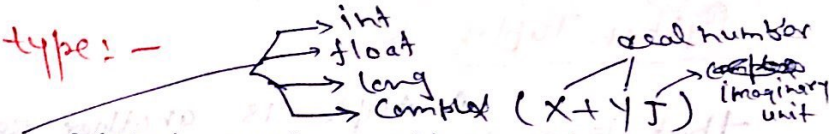
Multiple Assignment: —

```
a = b = c = 1
```

(Or)

```
a, b, c = 1, 2, "John"
```

Standard data type:-



- Number, var=1, var1=10 // also delete variable // del var
- String
- List
- Tuple
- Dictionary

String:- contiguous set of characters in the quotation marks. It allow either single (or) double quotes.

* subsets of string taking by slice operator ([ ], & [:]) with indexes starting at 0 in the begining of string
* (+) plus for concatenation & (*) asterisk for repetition of string.

Ex:-
```
str = 'Hello World!'
print str          # print complete string ⇒ Hello World!
print str[0]       # first character of string ⇒ H
print str[2:5]     # 3rd to 5th character ⇒ llo
print str[2:]      # string starting with 3rd character ⇒ llo World!
print str * 2      # prints string two times ⇒
print str + "TEST" # print concatenated string
```

Python List:- List is compound data types...

A list contains items separated by commas and enclosed within square brackets ([]). .. And all operation same as string.

Ex:-
```
list = ['abcd', 786, 2.23, 'john', 70.2]
tinylist = [123, 'john']
print list          → [abcd, 786, 2.23, 'john', 70.2]
print list[0]       → abcd
print list[1:3]     → [786, 2.23]
print list[2:]      → [2.23, 'john', 70.2]
print tinylist * 2  → [123, 'john', 123, john]
print list + tinylist → ['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Python Tuples ⇒

A tuple is another sequence data type that is similar to the list.. A tuple consists of a number of values separated by commas.. Unlike lists, tuples are enclosed within parenthesis..

List [ ]. elements & size can be changed while tuple ( ) cannot be updated. Tuples can be thought of as read-only lists...

Ex:—

```
tuple = ('abcd', 786, 2.23, 'John', 70.2)
tinytuple = (123, 'John')
print tuple ⇒ ('abcd', 786, 2.23, 'John', 70.2)
print tuple[0] ⇒ abcd
print tuple[1:3] ⇒ (786, 2.23)
print tuple[2:] ⇒ (2.23, 'John', 70.2)
print tinytuple*2 ⇒ (123, 'John', 123, 'John')
print tuple+tinytuple ⇒ ('abcd', 786, 2.23, 'John', 70.2, 123, 'John')
```

// tuple not allow to update

Ex:—

```
tuple = ('abcd', 786, 2.23, 'John', 70.2)
List = ['abcd', 786, 2.23, 'John', 70.2)
tuple[2] = 1000 // Invalid syntax with tuple
list[2] = 1000 // Valid syntax with list..
```

Dictionary:— are kind of hash table.. They work like associative arrays or hashes found in Perl and consists of key-value pair..

* Dictionary key can be almost any python type, but are usually numbers or strings..
* Dictionary are enclosed by curly braces ({}) and values can be assigned and accessed using square braces ([])

Ex:—

```
dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"
```

tinydict = {'name': 'John', 'code': 6734, 'dept': 'sales'}

print dict['one'] → This is one
print dict[2] → This is two
print tinydict → {'dept': 'sales', 'code': 6734, 'name': 'John'}
print tinydict.keys() → ['dept', 'code', 'name']
print tinydict.values() → ['sales', 6734, 'John']

Note: dictionaries have no concept of order among elements — it is 'out of order'...

Data type conversion:

* int(x[,base]) → convert x to an integer. base specify the base if x is string.
* long(x[,base]) → convert x to an long int
* float(x) → convert x to float
* complex(real [,imag]) → create complex no..
* str(x) → convert object x to string representation.
* repr(x) → convert object x to an expression string
* eval(str) → Evaluates a string and returns an object
* tuple(s) → convert s to a tuple
* list(s) → convert s to a list
* set(s) → convert s to a set
* dict(d) → create a dictionary. d must be a sequence of (key, value) tuples..
* frozenset(s) → convert s to a frozen set..
* chr(x) → convert integer to a character
* unichr(x) → convert integer to Unicode character
* ord(x) → convert a single character to its integer value..
* hex(x) → convert an integer to a hexadecimal string..
* oct(x) → convert an integer to an octal string

Type of operator:—

Arithmatic → +, −, *, /, %, ** (exponent) ⇒ a**b

// → Floor Division Ex:- 9//2 = 4

−11//3 = −4 9.0//2.0 = 4.0

−11.0//3 = −4.0

Python Comparision operator:—

==, !=, <> → similar to !=,

>, <, >=, <=, (Result is true (or) false).

Python Assignment operator:—

=, +=, −=, *=, /=, **=(exponent),

//= → (floor division)

a**=c ⇒ a=a**c

c//=a ⇒ c=c//a;

Python Bitwise operator:—

a=0011 1100, b=0000 1101

And:- a&b Ex:- 0000 1100

OR:- a|b Ex:- 0011 1101

XOR:- a^b Ex:- 0011 0001 → 0011

Not:- ~a Ex:- 1100 → 0011

<< Binary left shift: a<<2

>> Binary right shift: a>>2

Python Logical Operators:—

a=10, b=20

↳ Logical AND → (a and b) → true

↳ Logical OR → (a or b) → true

↳ Logical NOT → Not(a) → false.

Membership operators:—

↳ in → x in y, 1 if x is member of y

↳ not in → x not in y, 1 if x is not member of y.

Python operators Precedance :—

(order)

1. ** (Exponential → raise to the power)
2. ~ (Complement)
3. *, /, %, // (multiply, divide, modulus & floor division)
4. +, − (Addition, substraction)
5. >>, << (Right & left bitwise shift)
6. & (Bitwise AND)
7. ^, | (Bitwise (XOR and OR)
8. <=, >=, <, > (comparison operator)
9. <>, ==, != (Equality operator)
10. =, %=, /=, //=, −=, +=, *=, **= (Assignment operator)
11. Identity operator (is, is not)
12. Membership operator.
13. not, OR, and → logical operators

⇒ Python decision Making :—

* if stmt
* if ... else stmt
* nested if stmt

* **Single stmt Suit :—**

If the suite of an if clause consists only of a single line, it may go on same line as header stmt.

Ex :— var = 100
if (var == 100) : print "value of expression is 100"
print "Good bye!"

Output :— value of expression is 100
Good bye!

* loop → for
  → while
  → Nested.

⇒ Mathematical function:—

↳ abs(x) :- absolute value of x.

↳ ceil(x) → The ceiling of x: smallest int not less than x.

↳ cmp(x,y) :- -1 if x<y, 0 if x==y, 1 if x>y

↳ exp(x) → $x = e^x$

↳ fabs(x) → The absolute value of x.

↳ floor(x) → The largest int not greater than x.

↳ log(x).

↳ $\log_{10}(x)$ → log base 10;

↳ max($x_1, x_2, \ldots$)

↳ min($x_1, x_2, \ldots$)

↳ pow(x,y) → x**y ---

↳ ~~round~~ round(x,[,n])

↳ round(0.5) is 1.0

↳ round(-0.5) is -1.0

↳ sqrt(x)

⇒ Random Number Functions:—

Random numbers are used for games, simulation, testing, security, and privacy applications.
Python includes following funct...

* choice(seq,)
  ↳ A random item from a list, tuple or string

* randrange([start,], stop[, step])
  ↳ A random selected element from range(start, stop, step)

* random()
  ↳ A random float r, such that 0 is less than or equal to r and r is less than 1.

* seed([x])
  ↳ Sets the integer starting value used in generating random numbers.

* shuffle(lst) :- Randomizes the items of a list in place. Returns None ...

* uniform(x,y) :- A random float r, such that x is less than or equal to r and r is less than y ...

Trigonometric Function:—

* acos(x) → Return the arc cosine of x, in radians
* asin(x)
* atan(x)
* atan2(x,y) → Return atan(y/x), in radian
* hypot (x,y) → Return Euclidean norm
  ↳ sqrt(x*x + y*y).
* sin(x) → Return the sine of x radians
* tan(x) →
* degrees(x) →
* radians(x) →

Mathematical Constants:—

* Pi' → Mathematical constant pi.
* e → " e ..

String → String

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Accessing Value in String:—

```
var1 = 'Hello World!'
var2 = "Python Programming"
print "var1[0]: ", var1[0]
print "var2[1:5]:" var2[1:5]
```

Output:— var1[0]: H
var2[1:5]: ytho

Updating String:—

```
var1 = 'Hello World!'
print "Updated String:—", var1[:6] + 'Python'
```

Output: Updated String:— Hello Python

Escape Characters: — List of escape or non-printable character that can be represented with backslash notation..

* \a → Bell or alert
\b → Back space
\C-x (or) \cx → control-x
\e → Escape
\f → formfeed
\n → newline
\r → Carriage return
\s → space
\t → Tab
\v → vertical Tab
\x → character x

String Formatting operator →

* Python's coolest features is the string format operator %. This operator is unique to string & makes up for pack of having function from C's printf() family...

↳ print "My name is %s and weight is %d kg!" % ('Zara', 21)

Output:- My name is Zara and weight is 21 kg!

List of format Symbol:-

%c → character
%s → string conversion via str() prior to
%i → signed decimal integer
%d → signed decimal "
%u → unsigned decimal int.
%o → octal int
%x → hexadecimal int (lowercase letters)
%X → hexadecimal int (Upper case letters)
%e → exponential notation (lowercase 'e')
%E → " " (Uppercase 'E')
%f → floating point real number
%g → shorter of %f and %e
%G → shorter of %f and %E

Triple Quotes:- Escape characters work on triple quotes but not work in single (or) double quotes —

Ex:- print ''' This is Tab(\t) and [\n]!''' → output:- This is Tab( ) and [ ]!...

Ex:- print 'C:\\nowhere' → output:- C:\nowhere..

Built-in String Methods:—

* capitalize() → Capitalizes first letter of String
* center (width, fillchar) → Return space-padded string with the original string centered to a total of width columns..
* count (str, beg=0, end=len(string)) → Count how many times str occurs in string or in a substring of string if starting index beg and ending index end are given..
* find (str, beg=0 end=len(string))
  ↳ find index of string otherwise -1.
* index (str, beg=0, end=len(string))
* isalnum() → True if string has atleast 1 character and all character are alphanumeric and false otherwise..
* isalpha() → Return True if string has atleast 1 character and all characters are alphabetic and false otherwise..
* isdigit() → Return true if string contain only digits and false otherwise..
* islower() → True if all characters are lowercase otherwise false..
* isnumeric() →
* isspace() → True if string contains only white space characters and false otherwise..
* join(seq) → Merges (concatenates) the string representation of elements in sequence seq. into a string, with separator string.
* len(string) → return length of string
* ljust(width [, fillchar]) → Return a space-padded string with the original string left-justified to the total of width columns.

* lower() → Converts all uppercase letters into lowercase..

* lstrip() → Removes all leading whitespace in string.

* max(str) → Returns the max alphabetical character from the string str..

* min(str) → Returns the min " " " ..

* replace(old, new[, max] →

* rfind(str, beg=0, end=len(string))

* rindex(str, beg=0, end=len(string))

* rjust(width,[,fillchar])

* rstrip()

* split(str="", num=string.count(str))

* splitlines(num=string.count('\n'))

* startswith(str, beg=0, end=len(string))

* swapcase()
  ↳ Inverts case for all letters in string..

* isdecimal()
  ↳ Returns true if a unicode string contains only decimal characters and false otherwise..

* Lists — Lists *

Lists → List is different comma separated value between square brackets;—

Ex:— List1 = ['Physics', 'chemistry', 1997, 2000];
List2 = [1,2,3,4,5];
List3 = ["a", "b", "c", "d"];

Accessing Values in List:—

print "List1[0]: ", list1[0] → List1[0]: physics

print list2[1:5] → [2, 3, 4, 5]

Updating Lists:—

list1[2] = 2001;
print list1; Output: ['Physics', 'chemistry', 2001, 2000]

Delete List Elements :— To remove the list element you can either the **del** stmt if you know exactly which element you are deleting or **remove()** method if you do not know...

```
List1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];          →  "
print list1  ——→ ['physics', 'chemistry', 2000]
```

Basic List Operations :—

* len([1,2,3]) ——→ 3 → length
* [1,2,3]+[4,5,6] → [1,2,3,4,5,6] ——→ Concatenation
* ['Hi!']*4 ——→ ['Hi!', 'Hi!', 'Hi!', 'Hi!'] → Repetition
* 3 in [1,2,3] → True ——→ Membership
* for x in [1,2,3]; ——→ 1 2 3 ——→ Iteration
  print x,

Indexing, Slicing, and Matrixes :—

Lists are sequences, indexing and slicing work the same way for lists as they do for strings..

Assume following input :—

L = ['spam', 'Spam' 'SPAM']

* L[2] ——→ SPAM ——→ offsets start at zero
* L[-2] ——→ Spam ——→ Negative: count from the right
* L[1:] → ['Spam', 'SPAM'] → Slicing fetches sections.

Built-in List Functions & Methods :—

* cmp(list1, list2) → Compares elements of both lists
* len(list) ——→ Gives the total length of list.
* max(list) ——→ Returns item from the list with max value
* min(list) → " " " " " " min value
* list(seq) ——→ Converts a tuple into list

⇒ Python includes following List methods :-

* list.append(obj) → Appends object obj to list
* list.count(obj) →
* list.extend(seq) →
* list.index(obj) →
* list.insert(index, obj) →
* list.pop(obj = list[-1]) →
* list.remove(obj) → Remove objects obj from list..
* list.reverse() → Reverse objects of list in place..
* list.sort([func])
  └→ Sorts objects of list, use compare function if given..

* Tuples *

* A tuple is a sequence of immutable python objects..
* Tuple cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets..

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5);
tup3 = "a", "b", "c", "d";
```

* empty tuple :- tup1 = ();
* single value :- tup1 = (50,);

⇒ Accessing Values in Tuples :-

* To access values in tuple, use the square brackets for slicing among with the index or indices to obtain value available at that index..

Ex:-

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7);
print tup1[0]; → physics
print tup2[1:5]; → [2, 3, 4, 5]
```

⇒ Updating Tuples: —

Tuples are immutable which means you cannot update or change the value of tuple elements.. But create the new tuples i.e. —

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');
tup3 = tup1 + tup2;
print tup3; ⇒ (12, 34.56, 'abc', 'xyz')
```

⇒ Delete Tuples Elements: —

Removing individual tuple elements is not possible. There is, of course nothing wrong with putting together another tuple with the undesired elements discarded...

* To explicit remove an entire tuple, just use the **del** stmt....

Ex:—
```
tup = ('physics', 'chemistry', 1997, 2000);
del tup;
print tup; ⇒ Error: name 'tup' is not defined.
```

⇒ Basic Tuples Operations: —

Python Expression	Result	
* len((1,2,3))	⟶ 3	⟶ Length
* (1,2,3)+(4,5,6)	⟶ (1,2,3,4,5,6)	⟶ Concatenation
* ('Hi')*4	⟶ ('Hi','Hi','Hi','Hi')	⟶ Repetition
* 3 in (1,2,3)	⟶ True	⟶ Membership
* for x in (1,2,3): print x,	⟶ 1 2 3	⟶ Iteration

Indexing, Slicing and Matrixes: —

```
L = ('spam', 'Spam', 'SPAM')
```

* L(2) ⟶ 'SPAM' ⟶ offsets start at zero
* L[-2] ⟶ 'Spam' ⟶ Negative: count from the right
* L[1:] ⟶ ['Spam', 'SPAM'] ⟶ slicing fetches sections.

Built-in Tuple Function: —

* cmp(tuple1, tuple2)
* len(tuple)
* max(tuple)
* min(tuple) ⟶
* tuple(seq) ⟶ converts a list into tuple

* Python-Dictionary *

Dictionary: → Each key is separated from its value by a colon (:), the items are separated by commas and whole thing is enclosed by curly braces...

Empty Dictionary = {}

Accessing Values in Dictionary:—

```
dict = {'Name': 'Zara', 'Age': 7, 'class': 'First'}
print dict['Name'] → Zara.
print dict['Age'] → 7
print dict['Alice'] → Error, Not part of dictionary.
```

Updating Dictionary:—

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry or deleting an existing entry --

Ex:—

```
dict = {'Name': 'Zara', 'Age': 7, 'class': 'First'}
dict['Age'] = 8 # update existing entry
dict['School'] = "DPS school"; # Add new entry.
print dict['Age'] → 8
print dict['School'] → DPS school
```

Delete Dictionary Elements:—

You can either remove individual dictionary elements or clear the entire contents of dictionary. You can also delete entire dictionary in a single operation...

* To explicitly remove an entire dictionary, just use the del stmt.

```
dict = {'Name': 'Zara', 'Age': 7, 'class': 'First'}
del dict['Name']; // remove entry with key 'Name'.
dict.clear(); // remove all entries in dict
del dict; // delete entire dictionary
print dict['Age'] → // Error
print dict['school'] → // Error
```

Property of Dictionary Keys:—

* More than one entry per key not allowed. which means no duplicate key is allowed. when duplicate keys encountered during assignment, the last assignment wins..

Ex:— dict = {'name': 'Zara', 'age': 7, 'Name': 'Manni'}
print dict['Name'] ——→ Manni

* Keys must be immutable. which means you can use string, numbers or tuples as dictionary keys but something like ['key'] is not allowed...

Ex:— dict = {['Name']: 'Zara', 'Age': 7}
print dict['Name'] // Error: unhashable type...—

Built-in Dictionary Function & Method:—

* cmp(dict1, dict2)
* len(dict) ——→ Gives total length of dictionary.
* str(dict) ——→ produce a printable string representation of a dictionary
* type(variable) ——→ Returns the type of the passed variable..

*Dictionary Methods:—

* dict.clear() ——→ remove all elements of dictionary dict..
* dict.copy()
* dict.fromkeys()
* dict.get(key, default = None)
* dict.has_key(key)
* dict.items() ——→ list of (key, value) tuple pairs..
* dict.setdefault(key, default = None)
* dict.update(dict2)
* dict.values()
  ↳ Return list of dictionary values.
* dict.keys()
  ↳ Returns list of dictionary keys..

Python - Date & Time

* There is a popular time module available in python which provide function for working with times ...

Ex:-
```
import time;
ticks = time.time()
print ticks
```

Ex:-
```
import time;
localtime = time.localtime(time.time())
print localtime
```

Output:- time.struct_time(tm_year=2013, tm_mday=17, tm_hour=21, tm_min=26, tm_sec=3, tm_wday=2, tm_yday=1, tm_isdst=0)

TimeTuple:- Python's time function handle time as a tuple of 9 numbers.

Time-tuple = (4 Digit year, Month(1), Day(2), Hour(3), Minute(4), Second(5), Day of Week(6), Day of Year(7), Daylight Saving(8))

** time tuple as struct_time structure...

Attributes:-

Attributes	values
tm_year	2008
tm_mon	1 to 12
tm_mday	1 to 31
tm_hour	0 to 23
tm_min	0 to 59
tm_sec	0 to 61 (60 or 61 are leap-seconds)
tm_wday	0 to 6 (0 is Monday)
tm_yday	0 to 366 (Julian days)
tm_isdst	-1, 0, 1, -1 means library determines DST.

**
```
import time;
localtime = time.asctime(time.localtime(time.time()))
print "Local current time :", localtime
```

Output:- Local current time : Tue Jan 13 10:17:09 2009

Ans

Getting Calender for a month:—

```
import calender
cal = calender.month(2008, 1)
print cal
```

⇒ January 2008

Python – Functions

* Function : is a block of organized, reusable code, that is used to perform a single, related actions.
* Function provides better modularity for your application and high degree of code reusing...

Defining a function:—

Syntax:—

```
def function_name(parameters):
    "function_docstring"
    function_suite
    return [expression]
```

Ex

```
def printme(str):
    "This prints a passed string into this func"
    print str
    return
```

Calling a Function:—

```
def printme(str):
    print str
    return;
```
} fun definition

```
printme("Ram")
printme("shyam")
```
} fun calling

Output:—
```
Ram
Shyam
```

⇒ Pass by reference Vs Value:—

All parameters (arguments) in the Python language are passed by reference...

Ex:—
```
def changeme(mylist):
    mylist.append([1,2,3,4]);
    print "value:" mylist
    return
mylist = [10,20,30];
changeme(mylist);
print "value outside fun:" mylist
```
|| fun def.—

Output:— value : [10,20,30,1,2,3,4]
value outside fun : [10,20,30,1,2,3,4]

⇒ Function Arguments:—

* Required argument
* Keyword argument
* Default argument
* variable-length arguments

↳ Required Arguments:—

The no. of arguments in the function call should match exactly with the function definition..

Ex:—
```
def printme(str):
    "This prints a passed string into this fun"
    print str
    return;
printme() // fun call
```

Keyword Arguments: — When you use keyword arguments in a function call, the caller identifies the arguments by parameter name..

Eg:—
```
def printme(str):
    print str
    return
printme(str="My string")  // func calling
```
} func definition

Output:— My string

Eg:—
```
def printinfo(name, age):
    print "Name:", name
    print "Age", age
    return
printinfo(age=50, name="miki")  // func call
```
} func definition

Output:—
Name: miki
Age 50

Default Arguments: — It assumes a default value if the value is not provided in the function call for that argument.

Eg:—
```
def printinfo(name, age=35):
    print "Name:", name
    print "Age", age
    return;
printinfo(age=50, name="miki");
printinfo(name="miki")
```
} definition

Output:—
Name : miki
Age : 50
Name : miki
Age 35

Variable-length Arguments: —

```
def functionname([formal_args,] *var_args_tuple):
    "function_docstring"
    function_suite
    return [expression]
```

Ex:-

```
def printinfo (arg1, *vartuple):
    print "Output is:"
    print arg1
    for var in vartuple:
        print var
    return;
printinfo(10)
printinfo(70, 60, 50)
```

|| call printinfo function

Output:-

```
Output is:
10
Output is:
70
60
50
```

⇒ The Anonymous Functions:- are not declared in the standard manner by using def keyword. You can use the lambda keyword to create small anonymous funs...

* lambda form can take any number of argument but return just one value in the form of expression. They cannot contain commands or multiple expressions.

* An anonymous funs can not be a direct call to print because lambda requires an expression.

Syntax.- lambda funs contains only a single statement...

```
lambda [arg1 [,arg2, ... argn]] : expression
```

Ex:-

```
sum = lambda arg1, arg2: arg1+arg2;
print "value of total:", sum(10+20)
print "value of total:", sum(20,20)
```

Output:-

```
Value of total: 30
Value of total: 40
```

The Return Statement:—

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return stmt with no argument is the same as return None...

Ex:—

```
def sum(arg1, arg2):
    total = arg1 + arg2
    print "Inside func : " total
    return total;
total = sum(10,20);
print "Outside func : ", total
```

} func definition

Output:— Inside func : 30
Outside func : 30

Scope of Variables:—

- Global Variables
- Local Variables

Ex:—

```
total = 0;  ——> Global Variable
def sum(arg1, arg2):
    total = arg1 + arg2;
    print "Inside local total : ", total
    return total;
Sum(10,20);
print "Outside global total : ", total
```

} func definition

} func calling

Output:— Inside local total : 30
Outside global total : 0

Python Module :—

A Module allow you to ~~[illegible]~~ logic organize your python code.

* A Module can define funs, classes and variable
* A module can also include runnable code...

Ex :—

support.py

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

Module name : suppo

The import Statement :—

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

Ex :—

```
import support  // Import module suppo
support.print_func ("Zara")  // Now you
```

can call defined fun that module follows

Output :—

```
Hello : Zara
```

The from.... import Statement :⟹

Python's from statement lets you import specific attributes from a module into the current namespace ..

The from.... import has the following syntax

```
from modname import name1 [, name2 [, ...nam
```

Ex :— To import the fun fibonacci from the module fib, use the following stmt :-

```
from fib import fibonacci
```

The from.... import * Statement :⟹

It is also possible to import all names from a module into the current namespace by using the following import stmt :—

```
from modname import *
```

The dir() function — The dir() built-in func return a sorted list of strings containing the names defined by a module...

* "The list contains the name of all the modules, variable and functions that are define in a module...

Ex:—
```
import math
content = dir(math)
print content
```
→ import built in module math...

Output:— ['__doc__', '__file__', '__name__', 'asin', 'atan', ...]

⇒ The globals() and locals() functions ⇒ can be used to return the name in the global and local namespaces depending on the location from where they are called....

↳ The reload() Function:— The reload() function imports a previously imported module again...

```
reload(module_name)
```

Packages in Python:— A package is a hierarchical file directory structure that define a single Python application environment that consists of modules and subpackages and sub-subpackages and so on..

Consider a file Pots.py available Phone directory & Two other files

```
def Pots():
    print "I'm Pots Phone"
```

* Phone/Isdn.py having func Isdn()
* Phone/G3.py file having func G3()

Now create one more file __init__.py in phone directory.

↳ Phone/__init__.py.

To make all your func available, when you are imported Phone, you need to put explicit import statements in __init__.py as follows —

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

↳ After add all these lines to __init__.py, you have all of these classes available when you import the Phone package

```
import Phone
phone.Pots()
phone.Isdn()
phone.G3()
```

Output:

I'm Pots Phone
I'm 3G Phone
I'm Isdn Phone

Python - Files I/O

Printing to the Screen :—

↳ Simple use print stmt

Ex:- print "Hi how r you."

Output:- Hi how r you.

Reading Keyboard Input :—

Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. These functions are —

↳ raw_input
↳ input

The raw_input Function :—

The raw_input([Prompt]) function reads one line from standard input & returns it as a string (removing the trailing newline).

```
Ex:- str = raw_input("Enter your Input :")
     print "Received input is :", str
```

Output:- When typed Hello Python then results
Enter your Input : Hello Python
Received input is : Hello Python

The input Function :—

The input([prompt]) function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you..

Ex:— str = input("Enter your input: ")
print "Recieved input is :", str

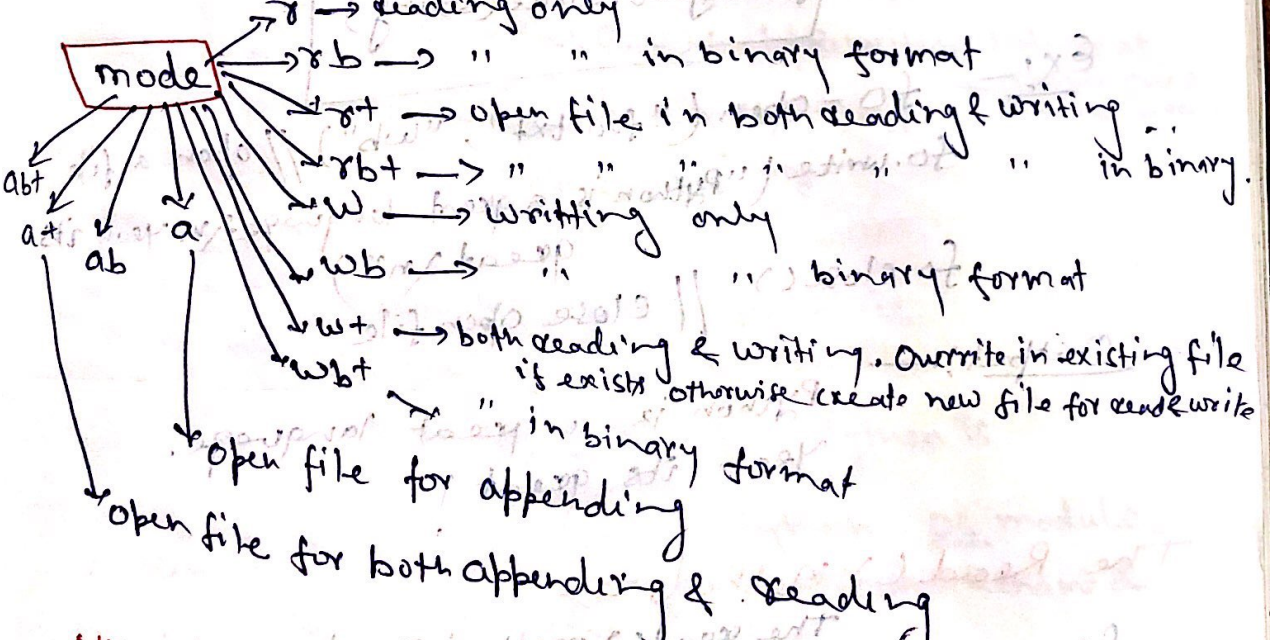
Output:— This would produce the following result against the entered input —

Enter your input : [x*5 for x in range(2,10,2)]
Recieved input is : [10, 50, 10]

⇒ Opening and Closing Files :—

The open Function:— built-in open() function.

file object = open(file_name [, access_mode][, buffering])



file Object Attributes:

- file.closed → return true if file is closed, otherwise false
- file.mode → return access mode with which file was opened..
- file.name → Return name of file
- file.softspace → Return false if space explicitely required with print, otherwise true...

foo.txt
Python is a great language :
yeah its great!

Ex:—
```
fo = open("foo.txt", "wb")
print "Name of file:", fo.name
print "Closed or not:", fo.closed
print "Opening mode:", fo.mode
print "Softspace flag:", fo.softspace
```
// Open a file

Output:—
Name of file: foo.txt
Closed or not : false
opening mode : wb
softspace flag : 0

The close() method :—

fileObject.close()

Ex :- fo = open("foo.txt", "wb") // open files
print "Name of the file:", fo.name
fo.close() // close opend file

Output :— Name of the file: foo.txt

⇒ Reading & Writing files :⇒

The write() Method ⇒ The write() method writes any string to an open file. The write() method does not add a newline character (\n) to the end of the string.

Syntax :—

```
fileObject.write(string)
```

Ex :— fo = open("foo.txt", "wb") // open a file
fo.write("Python is a great language.\n Yeah its great\n")
fo.close() // close open file

Output :—

Python is a great language.
Yeah its great !

The Read() Method ⇒

The read() method reads a string from an open file.

Syntax :—

```
fileObject.read([count])
```

Ex :— fo = open("foo.txt", "r+") // open a file
str = fo.read(10);
print "Read String is :", str
fo.close() // close open file.

Output :— Read string is : Python is

The File Positions ⇒

. The tell() method tell you the current position within the file. in other word the next read or write will occur at that many bytes from beginning of the file. ... → seek(offset[, from]) method change the current file position...

* If from is set to 0 → means beginning of the file... 1 means use the current position Set 2 means end of file...

Ex:—

```
fo = open("foo.txt", "r+") ||Open file
str = fo.read(10)
print "Read string is : ", str
position = fo.tell()  ||Check current Position
print "current file position : ", position
position = fo.seek(0,0); ||Reposition pointer at beginning once again
str = fo.read(10)
print "Again read String is : ", str
fo.close() ||close open file
```

Output:—

```
Read String is : Python is
Current file position : 10
Again read String is : Python is
```

⇒ Rename() Method:— Python os module & remove() method helps to perform Rename & deleting the files.

Syntax:—

```
os.rename(current_filename, new_file_name)
os.remove(file_name)
```

Ex:— existing file text1.txt

```
os.rename("text1.txt", "text2.txt") ||rename..
os.remove("text2.txt")
```

⇒ Directories in Python ⇒

All the files are contained within various directory. The OS module has several method that helps you create, remove and change directory.

↳ The mkdir() Method → `os.mkdir("newdir")`

↳ The chdir() Method → change the current directory

`os.chdir("newdir")`

↳ The getcwd() method → display the current working directory...

`os.getcwd()`

↳ the rmdir() Method :- deletes the directory which is passed as argument in the method.

`os.rmdir('dirname')`

Code :-

```
import os
os.mkdir("test") ||Create a directory.
os.chdir("/home/newdir") ||go to home directory
os.getcwd() ||give the current directory
os.rmdir("/tmp/test")
```

Python - Object Oriented :⇒

OOP Terminology :⇒

* Class → A user define prototype for an object that define a set of attribute that characterize any object of class..

The attribute of data member (class variable & instance variable) and methods, accessed via dot notation...

* class variable
* Data member
* Function Overloading
* Inheritance
* Instance
* Instantiation
* Method
* Object
* Operator overloading

Creating Classes ⇒

Syntax:

```
class ClassName:
    'Optional class documentation string'
    class_suite
```

* The class has documentation string, which can be accessed via ClassName.__doc__.
* Class_suite consists of all the component statements defining class members, data attributes and functions...

Ex!-

```
class Employee:
    empCount=0
def __init__(self, name, salary):
    self.name= name
    self.salary = salary
```

```
        Employee.empCount += 1

    def displayCount(self):
        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):
        print "Name: ", self.name, ", Salary:" self.salary
```

* `__init__()` → is a special method, which is called class constructor or initialization method that python calls when create a new instance of this class.....

— Creating Instance Objects ⇒

To create instances of a class, you call the class using classname and pass in whatever arguments its `__init__` method accepts --

↳ This would create first object of Employee class

```
emp1 = Employee("Zara", 2000)
```

↳ 2nd object of Employee class

```
emp2 = Employee("Manni", 5000)
```

Accessing Attributes: —

```
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

Built-in-Attributes: ⇒

* `__dic__` → dictionary containing the class's namespace —
* `__doc__` → a class documentation string or none, if undefined;
* `__name__` → class name
* `__module__` → Module name in which the class is defined
* `__base__` → empty tuple containing the base class --

Destroying Objects (Garbage Collection):—

Python deletes unneeded objects (built-in type or class instances) automatically to free the memory space..

* Python's garbage collector runs during program execution and is triggered when an object's reference count reaches zero...

```
a = 10      # Create object <40>
b = a       # Increase ref. count of <40>
c = [b]     #   "      "     "   "
del a       # decrease ref. count of <40>
b = 100     #   "      "     "   "
c[0] = -1   #   "      "     "   "
```

You normally will not notice when the garbage collector destroys an orphaned instance and reclaims its space. But a class can implement the special method —del—(), called a destructor, that is invoked when instance about to destroy..

Ex:— This —del—() destructor prints the class name of an instance that is about to be destroyed..

```
class point:
  def __init__(self, x=0, y=0):
      self.x = x
      self.y = y
  def __del__(self):
      class_name = self.__class__.__name__
      print class_name, "destroyed"
pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # prints id of objects
del pt1
del pt2
del pt3
```

Class Inheritance :—

Derived classes are declared much like their parent class; however list of base classes to inherit from is given after the class name—

```
Class SubClassName (ParentClass1, ParentClass2,...):
    class-suite
```

```
Class Parent:  # define parent class
    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 "Calling child Constructor
        print "Parent attribute :" Parent.parentAttr

Class child (Parent): // define child class
    def __init__(self):
        print "Calling child constructor"
    def childMethod(self):
        print "Calling child method"

c = child()      #Instance of child
c.childMethod()  # child call its method
c.parentMethod() # calls parent's method
c.setAttr(200)   # again call parent's method
c.getAttr()      # again call parent's method
```

Output:

```
Calling child constructor
Calling child method
Calling parent method
Parent attribute : 200
```

Multiple parent classes:—

```
class A:    # define class A
class B:    # define class B
class C(A,B):    # Subclass of A & B
```