

## Overview of Python:

- Python is fastest growing language in terms of no. of developers who are using it, no. of companies who are using it, no. of libraries we have, in terms of area it is implemented (Machine Learning, AI, GUI, Software Development, Web Development --- Everywhere). That's why it is known as GENERAL PURPOSE LANGUAGE.
- It supports POP and OOP.
- Java came in 1995 whereas Python came in 1989
- Python is a high – level general purpose programming language.
- It is a high – level, interpreted, interactive and object – oriented scripting language.
- It is designed to be highly readable.
- Python is easy. It is much simpler than C, C++, Java etc.
- Google, YouTube, Dropbox, Yahoo, Nasa are using Python as main language or part of language.
- It uses English keywords frequently whereas other languages use punctuation, and it has fewer syntactical constructions than other languages.
- Python is interpreted: Python is processed at runtime by the interpreter. We do not need to compile a program before executing it. This is similar as PHP.
- Python is interactive: We can directly write Python programs @ Python prompt.
- Python is Object – Oriented: Supports OOP style or technique of programming that encapsulates code within objects.
- Python is a Beginner's Language: It is a great language for the beginner – level programmers and supports the development of a wide range of applications from simple text processing to www browsers to games.

## History of Python:

- Python was developed by Guido Van Rossum in the late eighties and early nineties at The National Research Institute for Mathematics and Computer Science in the Netherlands. He named it after the television comedy show Monty Python's Flying Circus.
- Python is derived from many other languages including ABC, Modula – 3, C, C++, Algol – 68, SmallTalk, Unix Shell and other scripting languages.

- Python is copyrighted. Python source code is available under the GNU General Public License (GPL).

### **Python Features:**

- Easy-to-learn: Python has few keywords, simple structure and a clearly defined syntax.
- Easy-to-read: Python code is more clearly defined and visible to eyes.
- Easy-to-maintain:
- Broad Standard Library: Python's bulk of the library is very portable and cross platform compatible on UNIX, Windows and Mac.
- Interactive Mode: Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- Portable: Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- Extendable: We can add low-level modules to the Python interpreter.
- Database: Python provides interfaces to all major commercial databases.
- GUI Programming: Supports GUI applications that can be created and ported to many systems.
- Scalable: Provides a better structure and support for large programs.
- Supports functional and structured programming as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for large applications.
- It supports automatic garbage collection.
- Easily integrated with C, C++, COM, CORBA, JAVA.

## The Basic Elements of Python:

- A Python program sometimes called a *script*, is a sequence of definitions and commands.
- These definitions are evaluated and the commands are executed by the Python interpreter in something called the *Shell*.
- Typically, a new shell is created whenever execution of program begins. In most cases, a window is associated with the shell.
- A *command*, often called a *statement*, instructs the interpreter to do something.

For Example: print 'Work Hard'.

In Python 3, print is a function not a command; so we need to write:

```
print('Work Hard').
```

### 1. Objects, Expressions and Numerical Types:

- Objects are the core things that Python program manipulate.
- Every object has a type.
- Types are scalar or non – scalar.
  - o Scalar objects are indivisible. [Indivisible means splitting them is not easy].
  - o Non – scalar objects – Strings, have internal structure.
- Python has four types of scalar objects:
  - o int: Represents integers.
  - o float: Represents real numbers. (Number with decimal point).
  - o bool: Represents Boolean values i.e. true / false.
  - o none: Represents Python's null.

[str – Represents sequence of characters → Scalar Object Type in some book]

- Objects and Operators can be combined to form expressions, each of which evaluates to an object of some type.
- Numeric Types: Python has separate type for integers and floating point numbers. Unlike other languages, Python's int type is unbounded (No such thing as integer overflow). Python will do implicit conversions from integer to float when required.

- The symbol `>>>` is a Shell Prompt indicating that the interpreter is expecting the user to type some Python code into the shell. The line below the line with the prompt is produced when the interpreter evaluates the Python code entered at the prompt.

:Examples:	
Python Code Entered @ Prompt	(Output) Evaluation By Interpreter
<code>&gt;&gt;&gt; 3 + 2</code>	5
<code>&gt;&gt;&gt; 3.0 + 2.0</code>	5.0
<code>&gt;&gt;&gt; 3 != 2</code>	True
<code>&gt;&gt;&gt; 3 + 2.5</code>	5.5
<code>&gt;&gt;&gt; a = 1 + 2j</code> <code>&gt;&gt;&gt; b = a * 2</code> <code>&gt;&gt;&gt; b</code>	<div style="display: flex; align-items: center;"> <div style="border: 1px solid black; padding: 5px; margin-right: 10px;">Complex Types</div> <div style="font-size: 2em;">←</div> <div>(2 + 4j)</div> </div>
<b>Python function type can be used to find out type of object.</b>	
<code>&gt;&gt;&gt; type(3)</code>	(class 'int')
<code>&gt;&gt;&gt; type(3.0)</code>	(class 'float')
<code>&gt;&gt;&gt; type('Mac')</code>	(class 'str')

Operators on types int and float are:

$i + j$	If both type int, then the result is int; if either of them is float, the result is float. Performs sum of i and j.
$i - j$	Performs $i - j$ . If both type int, then the result is int; if either of them is float, the result is float.
$i * j$	Performs product of i and j. If both type int, then the result is int; if either of them is float, the result is float.
$i / j$	Performs division of i and j. Check following possibilities: <div style="display: flex; justify-content: space-between;"> <div><code>&gt;&gt;&gt; 4 / 3</code></div> <div>1.333333333</div> </div> <div style="display: flex; justify-content: space-between;"> <div><code>&gt;&gt;&gt; 4 // 3</code></div> <div>1</div> </div> <div style="display: flex; justify-content: space-between;"> <div><code>&gt;&gt;&gt; 10 / 5</code></div> <div>2.0</div> </div> <div style="display: flex; justify-content: space-between;"> <div><code>&gt;&gt;&gt; 10 // 5</code></div> <div>2</div> </div> <div style="display: flex; justify-content: space-between;"> <div><code>&gt;&gt;&gt; 4.2 / 2.5</code></div> <div>1.680000000</div> </div>

	>>> 4.2 // 2.5	1.0
	>>> 4.2 / 2	2.0
	>>> 4.2 // 2	2.0
i % j	Performs i modulo j	
i ** j	Performs i raised to the power of j.	

- Other Comparison Operators are: Equal (==), Not Equal (!=), Greater (>), Less (<), At Least (>=), At most (<=).
- Boolean Operators:
  - and: Example → >>> 500 > 5 and 5 > 2 results *true*
  - or: Example → >>> 5 > 5 or 5 > 2 results *true*
  - not: Example → >>> not 5 > 2 results *false*

## 2. Variables and Assignment

- In Python, a **variable is just a name**, nothing more.
- Python variables do not need explicit declaration to reserve memory space.
- The declaration happens automatically when you assign a value to a variable.
- An **assignment** statement associates the name to the left of the = symbol with the object denoted by the expression to the right of the =.
- In Python, variable names can contain uppercase and lowercase letters, digits and the special character underscore (\_).
- Python variable must start with letter / underscore not digit.
- Python variable names are case-sensitive e.g., mac and Mac are different names.
- Python has following list of **reserved words** (also known as **keywords**) having built-in meanings and cannot be used as variable names.

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

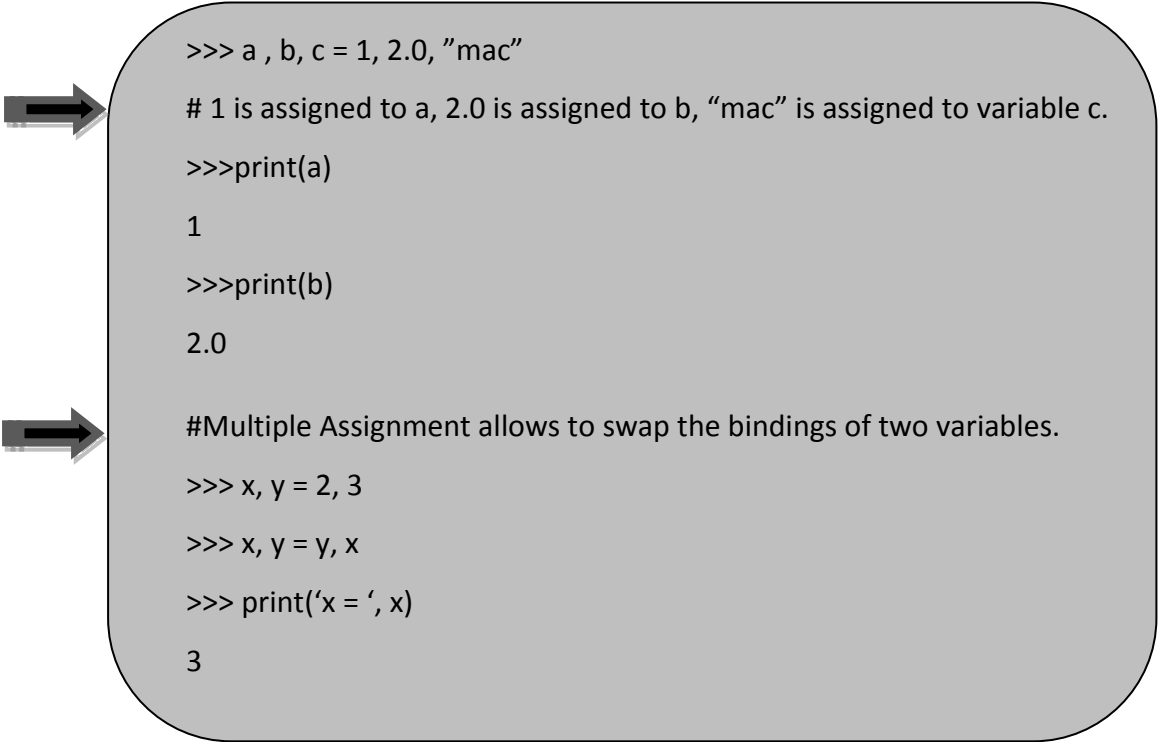
- *Simple assignment:* Python allows simple assignment of value to variable like other languages. Example:

```
>>> count = 100           #An Integer Assignment
>>> miles = 10.20         #A Floating Point Assignment
>>> name = "mac"          #A String Assignment.
>>> print(name)
mac                        #Output
```

- *Multiple assignment:* Python allows us to assign a single value to several variables simultaneously. Example:

```
>>> a = b = c = 1.        # value 1 is assigned to all three variables.
>>> print(a)
1
```

Python allows us to assign multiple values (objects) to multiple variables. Example:



```
>>> a, b, c = 1, 2.0, "mac"
```

# 1 is assigned to a, 2.0 is assigned to b, "mac" is assigned to variable c.

```
>>> print(a)
```

```
1
```

```
>>> print(b)
```

```
2.0
```

# Multiple Assignment allows to swap the bindings of two variables.

```
>>> x, y = 2, 3
```

```
>>> x, y = y, x
```

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

```
3
```

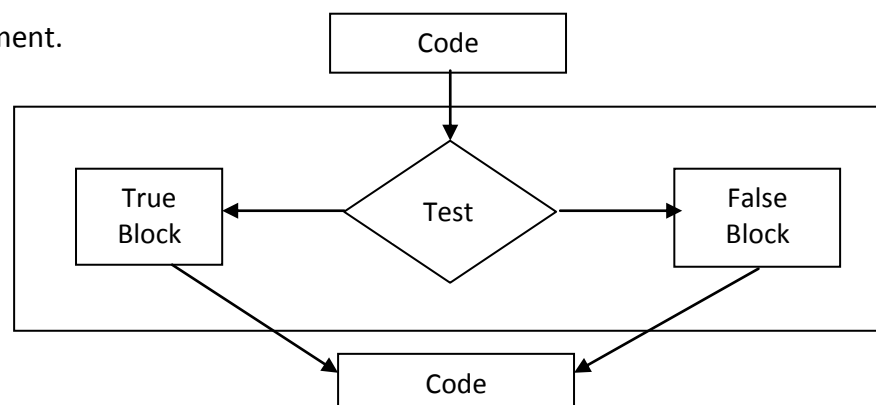
### 3) IDLE

- Typing programs directly to the shell is highly convenient.
- Most programmers prefer to use some sort of text editor that is part of an IDE (Integrated Development Environment).
- IDLE stands for "**Integrated Development and Learning Environment**" and since Van Rossum named the language Python partly to honor **British comedy group Monty Python**, the name IDLE was probably also chosen partly to honor **Eric Idle**, one of Monty Python's founding members.
- IDLE is an application. We can start it, the same way we would start any other application i.e. by double clicking on an icon.
- IDLE provides
  - o A text editor with syntax highlighting, auto completion, and smart indentation.
  - o A shell with syntax highlighting and an integrated debugger.
  - o Cross platform: Works mostly the same on Windows, Unix and Mac OS X.
  - o Colorizing of code input, output and error messages.
- IDLE has two main window types: The Shell Window and The Editor Window.

- It is possible to have multiple editor windows simultaneously.
- IDLE Menus:
  - File Menu (Shell Window and Editor Window)
  - Edit menu (Shell and Editor)
  - Format menu (Editor window only)
  - Run menu (Editor window only)
  - Shell menu (Shell window only)
  - Debug menu (Shell window only)
  - Options menu (Shell and Editor)
  - Window menu (Shell and Editor)
  - Help menu (Shell and Editor)
  - Context Menus

### **Branching Programs:**

- The kinds of computations we have been looking at thus far are called **straight-line programs**.
- They execute one statement after another in the order in which they appear, and stop when they run out of statements.
- These types of straight-line programs are not very interesting.
- **Branching** programs are more interesting.
- The simplest branching statement is a **conditional**.
- A conditional statement has three parts:
  - A test i.e. an expression that evaluates to either True / False.
  - A block of code that is executed if the test evaluates to True and
  - An optional block of code that s executed if the test evaluates to False.
- After conditional statement is executed, execution resumes at the code following the statement.





- In a Python, a conditional statement has the form

```
if Boolean_expression:
    block of code
else:
    block of code
```

- In this, Boolean\_expression indicates any expression that evaluates to True or False can follow the reserved word if. Block of code indicates any sequence of Python statements. Example:

```
if n % 2 == 0:
    print('Even')
else:
    print('Odd')
    print('hello')
```

- **Indentation** is semantically meaningful in Python. For example, if the last statement in the above code were indented it would be part of the block of code associated with the else, rather than with the block of code following the conditional statement.
- Python is unusual (extraordinary) in using indentation this way.
- Most other programming languages use some sort of bracketing symbols to delineate (define) blocks of code. For Example, in C language, we have enclosed blocks in braces {}.
- An advantage of the Python approach is that it ensures that the visual structure of a program is an accurate representation of the semantic structure of that program.
- If any of the true / false block of a conditional contains another conditional, this is said to be **nested**.



```
if x < y and x < z:
    print ('x is least')
elif y < z:
    print('y is least')
else:
    print('z is least')
```

- The *elif* in the above code stands for “else if”.

- Python Programming language provides different types of decision making statements like
  - if statement
  - if...else statements
  - if...elif...else statement
  - nested if statement
  - single line if statement
- *Single line if Statement:*
  - Many programming languages have a ternary operator, which define a conditional expression but the Python creator Guido Van Rossum rejected it as non-Pythonic, since it is hard to understand for people not used to C.
  - In Python, single line if statement will be like:

```
<expression_1> if <condition> else <expression_2>
```

- In the above form, it first evaluates the condition; if it returns true then expression\_1 will be evaluated to give the result otherwise expression\_2 will be evaluated.
- We can write as follow:

```
>>> age = 15
>>> print('kid' if age < 18 else 'adult')
kid          #Output

>>> age = 15
>>> print('kid' if age < 13 else 'teenager'
if age < 18 else 'adult')
teenager     #Output
```

## **String and Input:**

- Objects of type str are used to represent string of characters.
- Unlike many programming languages, Python has no type corresponding to a character, instead it uses string of length 1.

- Literals of type str can be written using either single or double quotes (e.g. 'mac' or "mac").
- The literal '123' denotes a string of characters not the number.
- Example of different string expressions:

```
>>> 'a'
'a'          #Output
>>> 3*4
12           #Output
>>> 3*'a'
'aaa'        #Output
>>> 3+4
7            #Output
>>> 'a'+'a'
'aa'         #Output
```

- The operator + is said to be **overloaded**: It has different meanings depending upon the types of the objects to which it is applied.
- For example, it means addition when applied to two numbers and concatenation when applied to two strings.
- The operator \* is also overloaded. It means multiplication when applied to two numbers and if it is applied to an int and a str, it duplicates the str. For example, 2 \* 'a' has the value 'aa'. There is a logic to this. Just as the expression 3 \* 2 is equivalent to 2 + 2 + 2, the expression 3 \* 'a' is equivalent to 'a' + 'a' + 'a'.
- Check the followings @ Python Interpreter:

```
>>> a
>>> 'a' * 'a'
>>> '4' < 3
>>> 'a' / 5
>>> 'a' - 2
>>> 'a' + 2
>>> 2 + 'a'
```

- The **type checking** in Python is not as strong as in some other programming languages. The designer of Python has decided that it should be False while using str and int with operators like <, because all numeric values should be less than all values of type str. The designer of some other languages decided that since such expressions don't have an obvious meaning, they should generate an error message.
- Strings are one of several sequence types in Python. Following operations can be done with all sequence types:

- The **length** of a string can be found using the len function.

```
>>> len('mac')
3           #len() function output
```

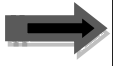
- **Indexing** can be used to extract individual characters from a string. In Python all indexing is zero-based. Python uses 0 to indicate the first element of a string. Negative numbers are used to index from the end of a string.

```
>>> 'mac'[0]
'm'   #Display string 'm'
>>> 'mac'[3]   #Generates Error
>>> 'mac'[2]
'c'       #Last Character as its index is 2
>>> 'mac'[-1]
'c'   #Negatives used to index from end of a str
>>> 'mac'[-2]
'a'
```

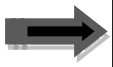
- **Slicing** is used to extract substrings of arbitrary (random) length. If s is a string, the expression s[start : end] denotes the substring of s that starts at index *start* and ends at index *end* - 1.



```
>>> 'mac'[1:3]
'ac'
#Shows string starts with index 1 upto index
#ends with 2 (3-1).
```



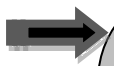
```
>>> 'mac'[1:]
'ac'
#If the value after the colon is omitted, it
#defaults to the length of the string.
```



```
>>> 'mac'[:2]
'ma'
#If the value before the colon is omitted, it
#defaults to 0.
```

### **Input:**

- Python 2.7 has `raw_input()` function to get input from user whereas Python 3 has `input()` function to get input directly from a user.
- `input()` takes a string as an argument and displays it as a prompt in the shell.
- It then waits for the user to type something, followed by hitting the enter key.
- `Input()` treats the typed line as a Python expression and assume a type.

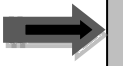


```
>>> name = input('Enter Your Name: ')
```

```
Enter Your Name: Mac
```

```
>>> print(name)
```

```
Mac
```



```
>>> print('Have a nice time ' + name + '!!!')
```

```
Have a nice time Mac!!!
```

```
>>> n = input('Enter Number: ')
```

```
Enter Number: 5
```



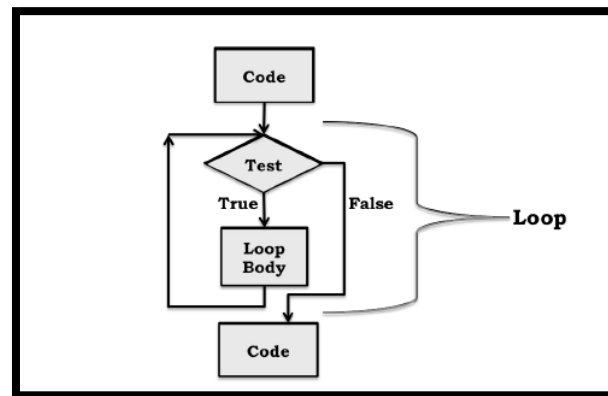
```
>>> print(type(n))
```

```
<class 'str'>
```

- **Type Conversions** (also called **type casts**) are used often in Python code. We use the name of a type to convert values to that type. For example, the value of `int('3')*4` is 12. When a float is converted to an int, the number is truncated (not rounded). For Example, the value of `int(3.9)` is the int 3.

### **Iteration:**

- A generic **iteration** (also called **looping**) mechanism is described in below diagram.



Flow chart for Iteration

- Like a conditional statement it begins with test. If the test evaluates to True, the program executes the loop body once, and then goes back to reevaluate the test. This process is repeated until the test evaluates to False, after which control passes to the code following the iteration statement.
- A loop statement allows us to execute a statement or group of statements multiple times.
- Python programming language provides following types of loops to handle looping requirements.

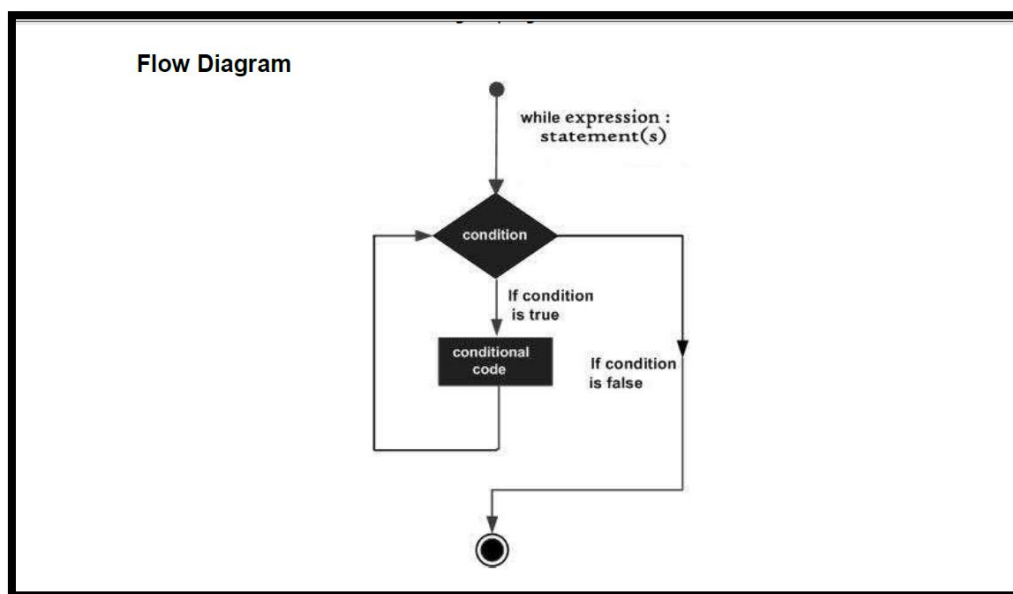
Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
nested loop	You can use one or more loop inside any other while, for.

- **while loop:** A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

**Syntax:**

```
while expression:  
    statement(s)
```

- Here, statement(s) may be a single statement or a block of statements. The condition may be an expression. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop. *Python uses indentation as its method of grouping statements.*



```
>>> count = 1  
>>> while(count < 6):  
    print(count)  
    count = count + 1  
  
1  
2  
3  
4  
5
```

- The Infinite Loop is loop in which a condition never becomes FALSE. We must use caution as this type of loop never ends. An infinite loop might be useful in

client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

```
>>> i = 1
>>> while i == 1:
        print('mac')
```

- When above code is executed, it prints 'mac' infinite times. We need to use *CTRL + C* to exit the program.
- Using else statement with loop: Python supports to have an **else** statement associated with a loop statement. If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

```
>>> i = 1
>>> while i <= 3:
        print('mac')
        i = i + 1
else:
        print('Haas!! Condition False')

mac
mac
mac
Haas!! Condition False
```

- Single Statement Suites: We can have single line **while** loop in Python as we have single line **if** statement. The single line while loop consists only of a single statement and it may be placed on the same line as the while header. It is better not to try this as it goes into the infinite loop and we need to press *CTRL + C* to exit.

```
>>> i = 1
>>> while (i): print('mac')
```

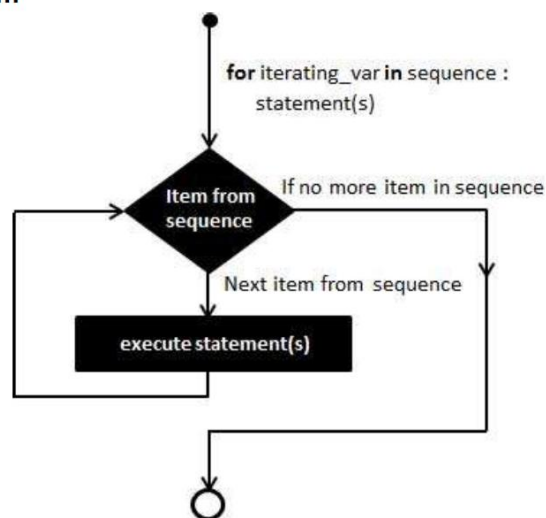
- **for loop**: A for loop statement in Python programming language has the ability to iterate over the items of any sequence, such as a list or a string.



**Syntax:**

```
for iterating_var in sequence:  
    statement(s)
```

- If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating\_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating\_var*, and the statement(s) block is executed until the entire sequence is exhausted.

**Flow Diagram**

```
>>> for i in 'Mac':  
    print(i)
```

```
M  
a  
c
```

```
>>> company=['Infosys' , 'Microsoft' , 'Sun']  
>>> for cmp in company:  
    print(cmp)
```

```
Infosys  
Microsoft  
Sun
```

- Iterating by Sequence Index: In Python, for loop has an alternative way to iterate over sequence – **Iterating Through Each Item by INDEX**.

```
>>> company=[ 'Infosys' , 'Microsoft' , 'Sun' ]
>>> for index in range(len(company)):
        print(company[index])

Infosys
Microsoft
Sun
```

- len() – Used to get total number of elements in tuple. and range() – Used to give the actual sequence to iterate over.
- Using else statement with loop: Python supports to have an **else** statement associated with a loop statement. If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.

```
>>> for i in range(1,5):
        print('mac')
else:
        print('Diversion Comes')

mac
mac
mac
mac
Diversion Comes
```

- **nested loop**: We can have one loop inside another loop in Python. Or we can say that we can have any type of loop inside of any other type of loop which is known as *Nested Loop*.

**Syntax for Loop:**

```
for iterating_var in sequence:
    for iterating_var in sequence:
        statement(s)
statement(s)
```

**Syntax while Loop:**

```
while expression:
    while expression:
        statement(s)
statement(s)
```

```
1
12
123
1234
12345
```

**for Loop:**

```
>>> for i in range(1,6):
        for j in range(1, i+1):
            print(j,end='')
        print()
```

**while Loop:**

```
>>> end=5
>>> i=1
>>> while i <= end:
        j = 1
        while j <= i:
            print(j,end='')
            j+=1
        i+=1
        print()
```

- **Loop Control Statements:** Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- Python supports the following control statements:

Control Statement	Description
break	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
pass	This statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

#### # break statement with for & while

```
>>> for letter in 'SauUni':
        if letter == 'U':
            break
        print(letter)
```

s  
a  
u

```
>>> count = 10
>>> while count > 0:
        print(count)
        count = count - 1
        if count == 5:
            break
```

10  
9  
8  
7  
6

```

# continue statement with for & while
>>> for letter in 'SauUni':
        if letter == 'U':
            continue
        print(letter)

s
a
u
n
i

```

```

>>> count = 5
>>> while count > 0:
        count = count - 1
        if count == 3:
            continue
        print(count)

4
2
1
0

```

```

# pass statement with for & while
>>> for letter in 'Sau':
        if letter == 'a':
            pass
        print('This is pass block')
        print(letter)

s
This is pass block
a
u

```

```

>>> var=5
>>> while var > 1:
        var = var - 1
        if var == 3:
            pass
        print('This is pass block')
        print(var)

4
This is pass block
3
2
1

```

## **Functions and Scoping:**

- So far, we have seen numbers, assignments, input / output, comparisons and looping constructs in Python. Such languages are called *Turing Complete*.
- In general, a programming language to be *Turing Complete* it needs:
  - o A form of conditional repetition or condition jumping statements (for, while, if, break, continue etc.)
  - o A way to read and write to some storage mechanism (variables)
- A function is a block of organized, reusable code that is used to perform a single, related action.
- Functions provide better modularity for application and a high degree of code reusing.
- Python has two types of functions: built-in functions and user-defined functions.
- To define function in Python:
  - o Function block begin with def followed by the function name and parentheses ( ).
  - o Any input parameters / arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
  - o The first statement of a function can be an optional statement – the documentation string of the function or docstring.
  - o The code block within every function starts with a colon (:) and is indented.
  - o The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

### **Syntax:**

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

```

#Example: Takes a string as input parameter & prints it.
#Python Function Declaration
def printme(str):
    "This is docstring"
    print (str)
    return

#Calling Function
>>> printme('mac')
mac
>>> printme("mac");
mac

```

```

#Example: Pass by Reference

>>> def printme(mylist):
        mylist.append([1,2,3,4]);
        print("Inside Function",mylist)
        return


>>> mylist=[10,20,30];
>>> printme(mylist);
Inside Function [10,20,30,[1,2,3,4]]
>>> print("Outside Function",mylist)
Outside Function [10,20,30,[1,2,3,4]]

```

- Function Arguments:
  - We can call a function by using the following types of formal arguments:
    - Required Arguments
    - Keyword Arguments
    - Default Arguments
    - Variable – Length Arguments

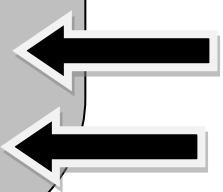
- Required Arguments: The arguments passed to a function in correct positional order. The number of arguments in the function call should match exactly with the function definition.

```
>>> def printme(str):  
    print (str)  
    return  
>>> printme('mac')  
mac  
>>> printme() #Generates an Error
```



- Keyword Arguments: These are related to the function calls. When we use keyword arguments in a function call, the caller identifies the arguments by the parameter name. This allows you to skip arguments to place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

```
>>> def printclg(name,number):  
    print ("Name:",name)  
    print ("Phone:",number)  
    return  
>>> printclg(number=2460012,name='rpbc')  
Name: rpbc  
Phone: 2460012  
>>> printclg('rp',11)  
Name: rp  
Phone: 11  
>>> printclg(11,'rp')  
Name: 11  
Phone: rp  
>>> printclg(number=11,'rp')  
Error  
>>> printclg(name='rp',11)  
Error
```





```
>>> def printme(str):
        print (str)
        return
>>> printme(str='mac')
mac
```

- **Default Arguments:** This is an argument that assumes a default value if a value is not provided in the function call for that argument.

```
>>> def printclg(number, name='rpbc'):
        print("Name:", name)
        print("Phone:", number)
        return
>>> printclg(2460012)
Name: rpbc
Phone: 2460012
>>> printclg(1234, 'rp')
Name: rp
Phone: 1234
```

- **Variable Length Arguments:** We may need to process a function for more arguments than we specified while defining the function. These arguments are called *Variable-Length arguments* and are not named in the function definition, unlike required and default arguments. An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments.

**Syntax:**

```
def function_name ([formal_arg,] *var_arg):
    "function_docstring"
    function_suite
    return [expression]
```

```
>>> def printclg(number, *var):  
        print(number)  
        for i in var:  
            print(i)  
        return  
  
>>> printclg(123)  
  
123  
>>> printclg(123, 'rp', 11, 420.22)  
  
123  
rp  
11  
420.22
```

- **The Anonymous Function:**

- These functions are called anonymous as they are not declared in the standard way with the *def* keyword.
- The *lambda* keyword is used to create small anonymous function.
- The anonymous function (Lambda forms) can take any number of arguments but returns just only one value in the form of an expression.
- They cannot contain multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Lambda's are a one-line version of a function.

**Syntax:**

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

```
>>> sum = lambda a1, a2: a1 + a2;
⇒ Check followings for Output / Error

>>> print (sum(10,20))
>>> print (sum)
>>>sum = lambda a1, a2:

>>> print (lambda(10,20))
>>> sum(10,20)
```

- The return Statement:

- The statement return [expression] exits a function in Python, optionally passing back an expression to the caller.
- A return statement with no arguments means nothing to return.

- Scoping:

- Each function defines a new namespace, also called a scope.
- All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.
- The scope of a variable determines the portion of the program where you can access a particular identifier.
- There are two basic scopes of variables in Python: Global and Local.
- Variables that are defined inside a function body have a *local scope*, and those defined outside have a *global scope*. That is, the local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body.

```
>>> sum = 0;    #Global Variable
>>> def printme(arg1, arg2):
            sum = arg1 + arg2;
            print("Inside:" , sum);
            return;

>>> printme(10,20)    #Output- Inside:30
>>> print (sum)      #Output - 0
```

### *For Information:*

- ⇒ At the top level (the level of the shell), a Symbol Table keeps track of all names defined at that level and their current bindings.
- ⇒ When a function is called, a new symbol table (also known as Stack Frame) is created. This table keeps track of all names defined within the function and their current bindings. If a function is called from within the function body, yet another stack frame is created.
- ⇒ When the function completes, its stack frame goes away.
- ⇒ In Python, one can always determine the scope of a name by looking at the program text which is known as Static or Lexical Scoping.

```
>>> def f(x):
    def g():
        x = 'abc'
        print('x = ', x)
    def h():
        z = x
        print('z = ', z)
    x = x + 1
    print('x = ', x)
    h()
    g()
    print('x = ', x)
    return g
>>> x = 3
>>> z = f(x)
x = 4
z = 4
x = abc
x = 4
>>> print('x = ', x)
x = 3
>>> print('z = ', z)
z = <function f.<locals>.g at 0x029E2198>
>>> z()
x = abc
```

[Explanation](#)

## **Specification:**

- A specification of a function defines a contract between the implementer of a function and those who will be writing programs that use the function. This contract can be thought of as containing two parts:
  - Assumptions: Conditions that must be met by users of function. Typically constraints on parameters, such as type, and sometimes acceptable ranges of values. For example, the first line of the docstring of `isVowel()` describe the assumptions that must be satisfied by clients (users) of `isVowel()`.
  - Guarantees: Conditions that must be met by function, provided that it has been called in way that satisfies assumptions. The last two lines of the docstring of `isVowel()` describe the guarantees that the implementation of the function must meet.

```
>>> def isVowel(char):  
    '''  
    char: a single letter of any case  
    returns: True if char is a vowel and  
    False otherwise.  
    '''  
    if char=='a' or char=='e' or char  
    == 'i' or char == 'o' or char == 'u':  
        return 'true'  
    else:  
        return 'false'
```

- Experienced programmers know that an investment in writing testing code often pays big dividends. It certainly beats typing test cases into the shell over and over again during **debugging** (the process of finding out why a program does not work, and then fixing it).
- The text between the triple quotation marks is called a **docstring** in Python.

## **Recursion:**

- The world's simplest recursive definition is probably the factorial function (typically written in mathematics using  $!$ ) on natural numbers (positive integers or non-negative numbers).

- The classic inductive definition is,

$$1! = 1$$

$$(n + 1)! = (n + 1) * n!$$

- The first equation defines the base case whereas the second equation defines factorial for all natural numbers, except the base case, in terms of the factorial of the previous number.
- In general, a recursive definition is made up of two parts. There is at least one base case that directly specifies the result for a special case and there is at least one recursive (inductive) case that defines the answer in terms of the answer to the question on some other input, typically a simpler version of the same problem.

```
>>> def fib(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        return fib(n-1) + fib(n-2)  
>>> def testfib(n):  
    for i in range(n+1):  
        print('fib of ',i,'=',fib(i))  
>>> testfib(3)          #check output of this.
```

### **Global Variables:**

- If we tried to call above testfib() with a large number, it will take long time to run.
- Suppose, we want to know how many recursive calls are made? For this, we could do a careful analysis of the code and figure it out. Another approach is to add some code that counts the number of calls. One way to do that uses global variables.
- In each function, the line of code *global numFibCalls* tells Python that the name *numCalls* should be defined at the outermost scope of the module in which the line of code appears rather than within the scope of the function.
- If the line *global numFibCalls* is omitted, then it would have been local to each fib and testFib functions.

```

>>> numFibCalls = 0
>>> def fib(n):
        global numFibCalls
        numFibCalls += 1
        if n==0 or n==1:
            return 1
        else:
            return fib(n-1) + fib(n-2)

>>> def testfib(n):
        for i in range(n+1):
            global numFibCalls
            numFibCalls = 0
            print('fib of ',i,'=',fib(i))
            print('fib called', numFibCalls)

>>> testfib(3)           #check output of this.

```

### **Setting Path at Windows:**

- To add the Python directory in Windows:  
At the command prompt type, ***path %path%;C:\Python*** and press Enter. (C:\Python is Python Directory)
- Add **PYTHONPATH** variable for modules:
  - Got to the Windows menu, right-click on “Computer” and select “Properties”. From the computer properties dialog, select “Advanced system settings” on the left.
  - From the advanced system settings dialog, choose the “Environment variables” button. In the Environment variables dialog, click the “New” button in the top half of the dialog, to make a new user variable.
  - Give the variable name as PYTHONPATH and the value is the path to the code directory. Choose OK and OK again to save this variable.
  - Now open a command prompt. Type following to confirm the environment variable is correctly set.  
***echo %PYTHONPATH%***
- To access module from other drive / directory
 

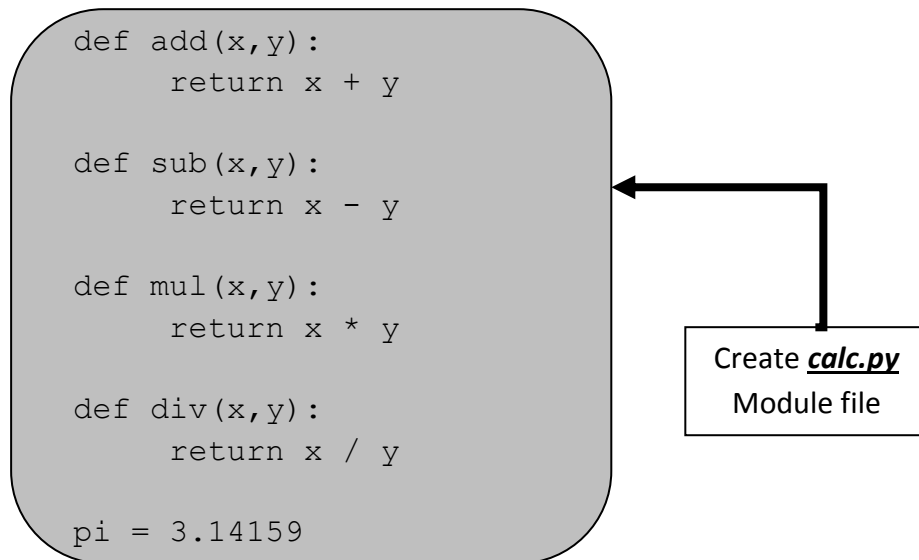
```

>>> import sys
>>> sys.path
>>> sys.path.append(r'D:\RP\Python')
>>> sys.path

```

## **Modules:**

- A module is a .py file containing Python definition and statements.
- A module allows us to logically organize our Python code. Grouping related code into a module makes the code easier to understand and use.
- A module is a Python object with arbitrarily named attributes that we can bind and reference.
- Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.



- A program gets access to a module through an *import* statement. We can use any Python source file as a module by executing an import statement in some other Python source file.

### **Syntax:**

```
import module1 [, module2 [, ... module n]
```

- When the interpreter encounters an import statement, it imports the module if the module is present in the PYTHONPATH. A PYTHONPATH (An environment variable) is a list of directories that the interpreter searches before importing a module.



- We can use module as follow:

```
>>> import calc
>>> print(calc.pi)
3.14159
>>> print(calc.add(5,2))
7
>>> print(calc.sub(5,2))
3
```

- A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.
- The use of dot notation may seem cumbersome (burdensome). On the other hand, the use of dot notation to fully qualify names avoids the possibility of getting burned by an accidental name clash. For example, we can have pi = 3.0 (as local variable) and pi = 3.14159 (as module variable).
- There is a variant of the import statement that allows the importing program to omit the module name when accessing names defined inside the imported module.
- The *from...import* Statement is used to import specific attributes from a module into current namespace.

**Syntax:**

```
from modname import name1[, name2[, .. namen]
```

In the following example, Line #1 only import add from calc module. It does not import entire module to current name space.

**Example:**

```
>>> from calc import add
>>> print(add(5,2))
7
>>> print(calc.add(5,2)) #Generates an Error
```

- The *from...import\** Statement is used to import all names from a module into the current namespace.

**Syntax:**

```
from modname import *
```

In the following example, Line #1 import all names from module calc into current namespace. This statement should be used sparingly (carefully).

**Example:**

```
>>> from calc import *
>>> print(add(5,2))
7
>>> print (sub(5,2))
3
```

- A module is imported only once per interpreter session. If we start up IDLE, import a module, and then change the contents of that module, the interpreter will still be using the original version of module. We can force the interpreter to reload all imported modules by restarting shell by pressing CTRL + F6.

## **Files:**

- Every computer system uses *files* to save things from one computation to the next.
- Python provides many facilities for creating and accessing files.
- Each operating system comes with its own file system for creating and accessing file.
- Python provides basic functions / methods necessary to manipulate files.
- Most of the file manipulation in Python can be done with *file* object.

○ File Manipulation Functions in Python:

1	<b><u>open(filename, filemode):</u></b> To read / write a file, we have to open it using Python's built-in open() function. This function creates <i>a file</i> object, which would be utilized to call other support methods associated with it.
2	<b><u>fh.close():</u></b> Close the file associated with file handle fh.
3	<b><u>fh.flush():</u></b> Used to flush the buffer of file handle fh like stdio's fflush().
4	<b><u>fh.read():</u></b> Returns a string containing the contents of the file associated with the file handle fh
5	<b><u>fh.readline():</u></b> Returns the next line in the file associated with the file handle fh.
6	<b><u>fh.readlines():</u></b> Returns a list each element of which is one line of the file associated with the file handle fh.
7	<b><u>fh.write(s):</u></b> Write the string s to the end of the file associated with the file handle fh.
8	<b><u>fh.writeLines(S):</u></b> S is a sequence of strings. Writes each element of S to the file associated with the file handle fh.
9	<b><u>fh.seek(offset [,whence]):</u></b> Sets the file handle fh's current position.
10	<b><u>fh.tell():</u></b> Returns the file handle fh's current position.

○ File Modes:

r	Opens a file for reading only. (Default Mode) The File Pointer is at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if exists and if does not exist, creates a new file for writing.
a	Opens a file for appending. The file pointer is at the end of the file if the file exists (in append mode).

**:Example:**

```
f = open('MyData', 'w')
f.write('Be Positive\n')
f.write('Stay Positive\n')
f.close()

print("\nReading file:")
f = open('MyData', 'r')
for line in f:
    print (line[:-1])
f.close()

f = open('MyData', 'a')
f.write('All plays PUBG\n')
f.write('How they remember RUBG\n')
f.close()

print("\nReading file after appending:")
f = open('MyData', 'r')
for line in f:
    print (line[:-1])
f.close()
```

**Tuples:**

- Python has three structured types: Tuple, list and dict.
- Like strings, *tuples* are ordered sequences of elements.
- The difference is that the elements of a tuple need not be characters.
- The individual elements can be of any type, and need not be of the same type as each other.
- Literals of type tuple are written by enclosing a comma-separated list of elements within parentheses.
- In short, a tuple is a sequence of immutable Python objects (Immutable means we cannot change or update values of tuple elements).

### ***For Information:***

- ⇒ Tuples are immutable Python Objects.
- ⇒ To create tuple, we have to put comma-separated values.
- ⇒ Tuple with single value only, must include a comma.
- ⇒ Like string indices, tuple indices start at 0, and they can be sliced, concatenated and so on.
- ⇒ Tuple can be accessed with the use of square bracket for slicing along with the index / indices to obtain value(s).
- ⇒ We cannot change / update the values of tuple elements.
- ⇒ We can take portions of existing tuple(s) to create new tuple(s).
- ⇒ We cannot remove individual elements of tuple but we can remove entire tuple with *del* statement.

### **#Creating Tuples**

```
>>> t1 = ()      #Empty Tuple
>>> t2 = (33,)   #Tuple with single value must have comma
>>> t3 = (1,'two',3)
>>> t4 = ('goal', 'achieve', 'success')
```

### **#Accessing Tuples**

```
>>> print(t1)      #()
>>> print(t2)      #(33,)
>>> print(t3)      #(1,'two',3)
>>> print(t4)      #('goal', 'achieve', 'success')
>>> print(t3[1])    #two
>>> print(t4[1:3])  # ('achieve', 'success')
>>> t3[0] = 100     # Generates Error as t3 is immutable
>>> t5 = t2 + t3
>>> print (t5)      # (33, 1, 'two', 3)
>>> del t5          # Deleting Tuple
>>> print(t5)       # Generates Error as t5 not exist
```

- Tuples respond to + and \* operators like strings. Tuples respond to all basic operations (Concatenation, Length, Repetition, Membership, Iteration, Indexing, Slicing) which we have used on strings.
- Consider following operations in which tuple t1 has (1, 'hi', 2.5) values.

Python Expression	Output
len(t1)	3
('good',100) + t1	('good', 100, 1, 'hi', 2.5)
t1 * 3	(1, 'hi', 2.5, 1, 'hi', 2.5, 1, 'hi', 2.5)
2.5 in t1	True
for i in t1 : print (i)	1 hi 2.5
t1[2]	2.5
t1[-2]	'hi'
t1[1:]	('hi', 2.5)
We can use <i>min()</i> and <i>max()</i> same as <i>len()</i> with tuple but for that it requires tuple elements of same type.	

### **Lists and Mutability:**

- Like a tuple, a *list* is an ordered sequence of values, where each value is identified by an index.
- The syntax for expressing literals of type *list* is similar to that used for tuples; the difference is that we use square brackets rather than parentheses.
- The empty list is written as [], and singleton list (a list with single value only) are written without comma before the closing bracket.

```
>>> list1 = ['Work Hard', 4, 'Success']
>>> for i in range(len(list1)):
        print(list1[i])

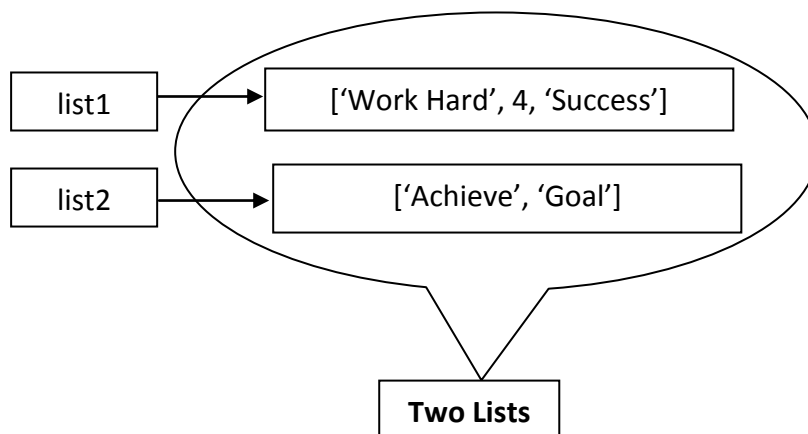
#Output
Work Hard
4
Success
```

Example of List

- Items in a list need not be of the same type – This is the most important thing about list.
- Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.
- Lists differ from tuples in one hugely important way: lists are *mutable*. In contrast, tuples and strings are *immutable*.
- There are many operators that can be used to create objects of these immutable types, and variables can be bound to objects of these types. But objects of immutable types cannot be modified.
- On the other hand, objects of type *list* can be modified after they are created.
- The distinction between mutating an object and assigning an object to a variable may appear subtle (slightly differ).
- For Example,

```
>>> list1 = ['Work Hard', 4, 'Success']
>>> list2 = ['Achieve', 'Goal']
```

- When above statements are executed, the interpreter creates two new lists and binds the appropriate variables to them as below:



- The assignment statements,

```
>>> st1 = [list1, list2]
>>> st2 = [['Work Hard', 4, 'Success'], ['Achieve', 'Goal']]
```

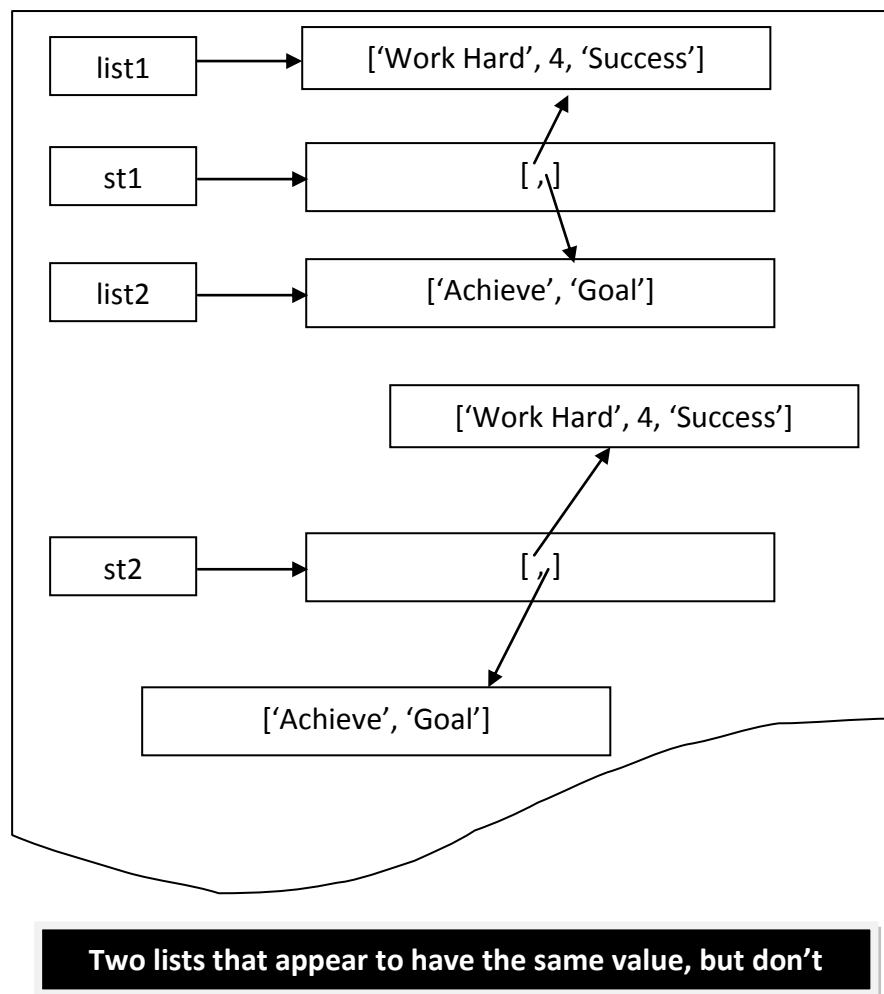
- The above statements create new lists and bind variables to them. The elements of these lists are themselves lists.

```
>>> print (st1)
[['Work Hard', 4, 'Success'], ['Achieve', 'Goal']]

>>> print (st2)
[['Work Hard', 4, 'Success'], ['Achieve', 'Goal']]

>>> print (st1 == st2)
True
```

- It appears as if *list* and *list1* are bound to the same value. But appearances can be deceiving (unreliable). From the following picture, *list* and *list1* are bound to quite different values.





- That `st1` and `st2` are bound to different objects can be verified using the built-in Python function `id`, which returns a unique integer identifier for an object. This function allows us to test for *Object Equality*.

```
>>> print (st1 == st2)
True

>>> print (id(st1) == id(st2))
False

>>> print(id(st1))
42035960

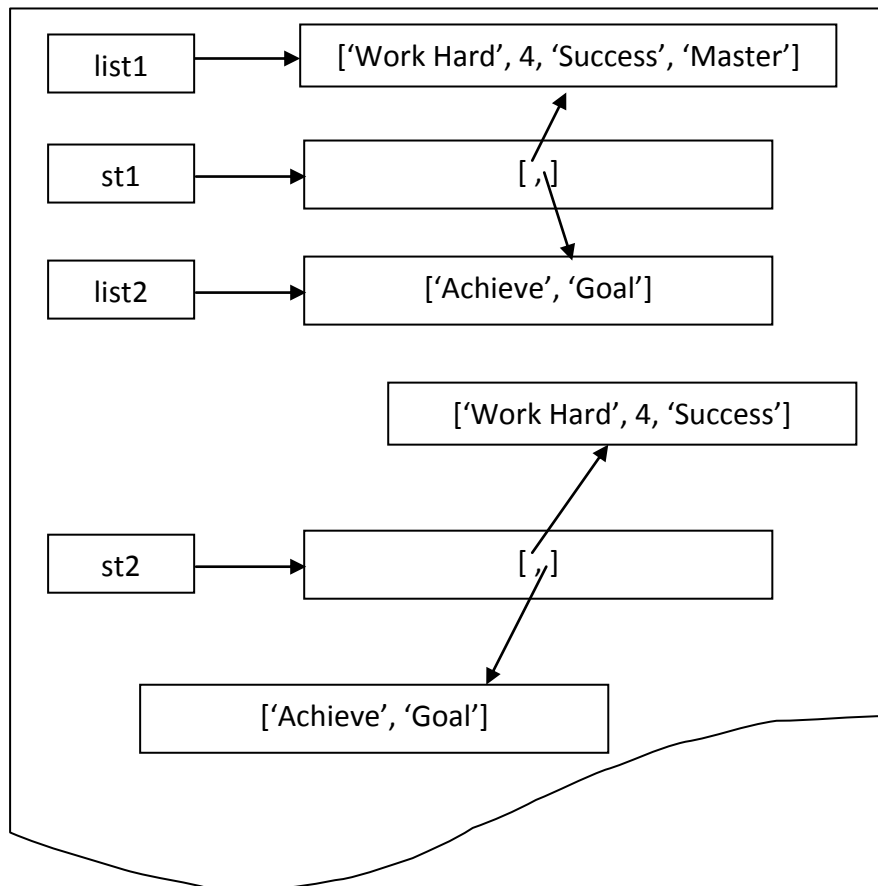
>>> print(id(st2))
42036600
```

- The elements of `st1` are not copies of the lists to which `list1` and `list2` are bound, but are rather the lists themselves.
- The elements of `st2` are lists that contain the same elements as the lists in `st`, but they are not the same lists. We can see this by running the code:

```
>>> print(id(st1[0]), ' ', id(st2[0]))
42036160      43535656

>>> print(id(st1[1]), ' ', id(st2[1]))
42036240      42035560
```

- The elements of `st2` are lists that contain the same elements as the lists in `st`, but they are not the same lists.
- Consider the Code: `list1.append('Master')`
- The *append* method has a *side effect*. Rather than create a new list, it mutates the existing list `list1` by adding a new element, the string 'Master', to the end of it.
- After `append` is executed, the state of the computation looks like:



### Example of Mutability

- Output after appending an item to the list:

```

>>> print(st1)
[['Work Hard', 4, 'Success', 'Master'], ['Achieve', 'Goal']]
>>> print(st2)
[['Work Hard', 4, 'Success'], ['Achieve', 'Goal']]
  
```

- What we have here is something called *aliasing*.
- There are two distinct paths to the same list object. One path is through the variable `list1` and the other through the first element of the list object to which `st1` is bound. One can mutate (alter / change) the object via either path, and the effect of the mutation will be visible through both paths. This can be convenient, but it can also be treacherous (untrustworthy). Unintentional aliasing leads to programming errors that are often enormously hard to track down.

Methods associated with lists	
List1.append(e)	Adds the object e to the end of List1.
List1.count(e)	Returns the number of times that e occurs in List1.
List1.insert(i, e)	Inserts the object e into List1 at index i.
List1.extend(List2)	Adds the items in list List2 to the end of List1.
List1.remove(e)	Deletes the first occurrence of e from List1.
List1.index(e)	Returns the index of the first occurrence of e in List1. (Raises an exception if e is not in List1).
List1.pop(i)	Removes and returns the item at index i in List1. If i is omitted, it defaults to -1, to remove and return the last element of List1.
List1.sort()	Sorts the elements of List1 in ascending order.
List1.reverse()	Reverses the order of the elements in List1.

**:: Try Followings::**

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

```

>>> l1.remove(12)
>>> print(l1)
[1, 2, 3, 4, 5, 6]
>>> l1.pop(0)
1
>>> print(l1)
[2, 3, 4, 5, 6]
>>> l1.index(2)
0
>>> l1.index(4)
2
>>> l1.count(3)
1
>>> print(l1)
[2, 3, 4, 5, 6]
>>> l1.count(5)
1
>>> print(l2)
[6, 5, 4]
>>> l1.reverse()
>>> print(l1)
[6, 5, 4, 3, 2]

```

- *List Comprehension* provides a concise way to apply an operation to the values in a sequence.
- It creates a new list in which each element is the result of applying a given operation to a value from a sequence.
- The *for* clause in a list comprehension (check below example) can be followed by one or more *if* and *for* statements that are applied to the values produced by the *for* clause.

```

>>> l = [x**2 for x in range(1,7)]
>>> print(l)
[1, 4, 9, 16, 25, 36]
>>> for x in range(1,7):
    print(x)
1
2
3
4
5
6
>>> mixed = [1,2,'a',3,4.0]
>>> print([x**2 for x in mixed if type(x) == int])
[1, 4, 9]

```

### **Functions as Objects:**

- In Python, functions are *first-class objects*. That means that they can be treated like objects of any other type (*for example: int or list*).
- In Python, functions may be stored in *data structures*, *passed as arguments* or *used in control structures*.
- A programming language is said to support first-class functions if it treats functions as first-class objects.
- Python supports the concept of *First Class* functions.
- Properties of first class functions:
  - A function is an instance of the Object type.
  - We can store the function in a variable.
  - We can pass the function as a parameter to another function.
  - We can return the function from a function.
  - We can store them in data structures such as hash tables, lists, ...
- Examples illustrating *First Class* functions in Python:

(1) *Functions are objects*: Python functions are first class objects. In the following example, we are assigning function to a variable. This assignment doesn't call the function. It takes the function object referenced by `fun1` and creates a second name pointing to it, `var1`.

```
>>> def fun1(text):  
        return text.upper()  
  
>>> print(fun1('Hello'))  
HELLO  
  
>>> var1 = fun1  
  
>>> print(var1('Hello'))  
HELLO
```

(2) *Functions can be passed as arguments to other functions*: Because functions are objects, we can pass them as arguments to other functions. Functions that can accept other functions as arguments are also called *higher-order functions*. In the following example, we have created *display* which takes a function as an argument.

```
>>> def fun1_upper(text):  
        return text.upper()  
  
>>> def fun1_lower(text):  
        return text.lower()  
  
>>> def display(func):  
        msg = func('Passing Func as Argu.')  
        print(msg)  
  
>>> display(fun1_upper)  
PASSING FUNC AS ARGU.  
  
>>> display(fun1_lower)  
passing func as argu.
```

(3) *Functions can return another function*: Because functions are objects we can return a function from another function. In the following example, the *fun\_add* function returns *add* function.

```

>>> def fun_add(x):
        def add(y):
            return x + y
        return add

>>> msg = fun_add(15)
>>> print(msg(10))
25

```

### **Strings, Tuples and Lists:**

- We have looked at three different sequence types: *str*, *tuple*, and *list*.
- They are similar in that objects of all of these types can be operated upon as described below:

Common Operations on Sequence Types	
seq[i]	Returns the ith element in the sequence.
len(seq)	Returns the length of sequence.
seq1 + seq2	Returns the concatenation of the two sequences.
n * seq	Returns a sequence that repeats seq n times
seq[start : end]	Returns a slice of the sequence.
e in seq	Is true if e is contained in the sequence and false otherwise.
e not in seq	Is true if e is not in the sequence and false otherwise.
for e in seq	Iterates over the elements of the sequence.

- Some of the similarities and differences of Sequence types are:

Type	Type of elements	Examples of literals	Mutable
str	characters	' ', 'a', 'abc'	No
tuple	any type	( ), (3, ), ('abc', 4)	No
list	any type	[ ], [3], ['abc', 4]	Yes

- Python programmers tend to use lists far more often than tuples. Since lists are mutable, they can be constructed incrementally during a computation. For example, the following code incrementally builds a list containing all of the even numbers in another list.

```
>>> evenElems = []
>>> for e in L:
    if e%2 == 0:
        evenElems.append(e)
```

- One advantage of tuples is that because they are immutable, aliasing is never a worry. Another advantage of their being immutable is that tuples, unlike lists, can be used as keys in dictionaries.
- Since strings can contain only characters, they are considerably less versatile than tuples or lists. On the other hand, when you are working with a string of characters there are many built-in methods that make life easy. Following contains short descriptions of a few of them. Keep in mind that since strings are immutable these all return values and have no side effect.

Some methods on strings	
s.count (s1)	Counts how many times the string <i>s1</i> occurs in <i>s</i> .
s.find (s1)	Returns the index of the first occurrence of the substring <i>s1</i> in <i>s</i> , and -1 if <i>s1</i> is not in <i>s</i> .
s.rfind (s1)	Same as find, but starts from the end of <i>s</i> . ('r' means reverse)
s.index (s1)	Same as find, but raises an exception if <i>s1</i> is not in <i>s</i> .
s.rindex (s1)	Same as index, but starts from the end of <i>s</i> .
s.lower()	Converts all uppercase letters in <i>s</i> to lowercase.
s.replace (old, new)	Replaces all occurrences of the string <i>old</i> in <i>s</i> with the string <i>new</i> .
s.rstrip()	Removes trailing white space from <i>s</i> .
s.split(d)	Splits <i>s</i> using <i>d</i> as a delimiter. Returns a list of substrings of <i>s</i> . If <i>d</i> is omitted, the substrings are separated by arbitrary strings of whitespace characters (space, tab, newline).

### **Dictionaries:**

- Objects of type *dict* (i.e. dictionary) are like lists except that "*indices*" need not be integers – they can be values of any immutable type.
- Since they are not ordered, we call them *keys* rather than indices.



- Dictionary is a set of key / value pairs.
- Literals of type *dict* are enclosed in *curly braces* { }, and each element is written as a key followed by a colon : followed by a value.
- The entries in dictionary are unordered and cannot be accessed with an index.
- When a *for* statement is used to iterate over a dictionary, the value assigned to the iteration variable is a key, not key / value pair.
- Dictionaries are one of the great things about Python. They greatly reduce the difficulty of writing a variety of programs.
- Like lists, dictionaries are mutable. So, we must be careful about side effects.
- Keys must be unique. Objects of any immutable type may be used as dictionary keys. For example, using a tuple of the form (flightnumber, day) to represent airline flights.

#### **#Creating Dictionary**

```
>>> dict = {'Rajkot':'GJ3', 'Morbi':'GJ36','AMD':'GJ1'}
```

#### **#Accessing values in Dictionary**

```
>>> print (dict['Rajkot'])
```

```
GJ3
```

```
>>> print (dict['Baroda']) #Generates Error
```

```
>>> dict['Rajkot'] = 'GJ007' #Updating Dictionary
```

```
>>> print(dict['Rajkot'])
```

```
GJ007
```

```
>>> dict['Baroda'] = 'GJ6' #Adding to Dictionary
```

#### **#Deleting from Dictionary (print(dict) after each statement) .**

```
>>> del dict['Baroda'] #Removes specific entry
```

```
>>> dict.clear() #Removes all entries
```

```
>>> del dict #Delete entire directory
```

### *For Information:*

⇒ Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user – defined objects. However, same is not true for the keys.

⇒ There are two important points to remember about 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.

```
>>> dict {'Rajkot' : 'Smart', 'Amd' :  
          'Good', 'Rajkot' : 'Great'}  
>>> print(dict)
```

- Keys must be immutable. Which means we can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

```
>>> dict [['Rajkot'] : 'Smart'] #Error
```

Some common operations on dictionary	
len (d)	Returns the number of items in <i>d</i> .
d.keys ()	Returns a list containing the keys in <i>d</i> .
d.values ()	Returns a list containing the values in <i>d</i> .
k in d	Returns true if key <i>k</i> is in <i>d</i> .
d[k]	Returns the item in <i>d</i> with key <i>k</i> .
d.get(k, v)	Returns <i>d[k]</i> if <i>k</i> is in <i>d</i> , and <i>v</i> otherwise.
d[k] = v	Associates the value <i>v</i> with the key <i>k</i> in <i>d</i> . If there is already a value associated with <i>k</i> , that value is replaced.
del d[k]	Removes the key <i>k</i> from <i>d</i>
for k in d	Iterates over the keys in <i>d</i> .

**Example:**

```
>>> EtoF = {'bread':'pain', 'wine':'vin', 'with':'avec',
            'I':'Je', 'eat':'mange', 'drink':'bois',
            'John':'Jean', 'friends':'amis', 'and': 'et',
            'of':'du', 'red':'rouge'}
>>> FtoE = {'pain':'bread', 'vin':'wine', 'avec':'with',
            'Je':'I', 'mange':'eat', 'bois':'drink',
            'Jean':'John', 'amis':'friends', 'et':'and',
            'du':'of', 'rouge':'red'}
>>> dicts={'English to French':EtoF, 'French to English':FtoE}
>>> def translateWord(word, dictionary):
    if word in dictionary.keys():
        return dictionary[word]
    elif word != '':
        return "'" + word + "'"
    return word

>>> def translate(phrase, dicts, direction):
    UCLetters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    LCLetters = 'abcdefghijklmnopqrstuvwxyz'
    letters = UCLetters + LCLetters
    dictionary = dicts[direction]
    translation = ''
    word = ''
    for c in phrase:
        if c in letters:
            word = word + c
        else:
            translation = translation\
                + translateWord(word, dictionary) + c
            word = ''
    return translation + ' ' + translateWord(word, dictionary)

>>> print (translate('I drink good red wine, and eat bread.',
                    dicts, 'English to French'))
Je bois "good" rouge vin, et mange pain.

>>> print (translate('Je bois du vin rouge.', dicts, 'French to
                    English'))
I drink of wine red.
```