

Lists

- ❖ List is an ordered sequence of items. Values in the list are called elements / items.
- ❖ It can be written as a list of comma-separated items (values) between **square brackets[]**.
- ❖ Items in the lists can be of different data types.

Operations on list:

1. Indexing
2. Slicing
3. Concatenation
4. Repetitions
5. Updating
6. Membership
7. Comparison

operations	examples	description
create a list	<code>>>> a=[2,3,4,5,6,7,8,9,10]</code> <code>>>> print(a)</code> <code>[2, 3, 4, 5, 6, 7, 8, 9, 10]</code>	in this way we can create a list at compile time
Indexing	<code>>>> print(a[0])</code> 2 <code>>>> print(a[8])</code> 10 <code>>>> print(a[-1])</code> 10	Accessing the item in the position 0 Accessing the item in the position 8 Accessing a last element using negative indexing.
Slicing	<code>>>> print(a[0:3])</code> [2, 3, 4] <code>>>> print(a[0:])</code> [2, 3, 4, 5, 6, 7, 8, 9, 10]	Printing a part of the list.
Concatenation	<code>>>>b=[20,30]</code> <code>>>> print(a+b)</code> [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30]	Adding and printing the items of two lists.
Repetition	<code>>>> print(b*3)</code> [20, 30, 20, 30, 20, 30]	Create a multiple copies of the same list.

Updating	<pre>>>> print(a[2]) 4 >>> a[2]=100 >>> print(a) [2, 3, 100, 5, 6, 7, 8, 9, 10]</pre>	Updating the list using index value.
Membership	<pre>>>> a=[2,3,4,5,6,7,8,9,10] >>> 5 in a True >>> 100 in a False >>> 2 not in a False</pre>	Returns True if element is present in list. Otherwise returns false.
Comparison	<pre>>>> a=[2,3,4,5,6,7,8,9,10] >>>b=[2,3,4] >>> a==b False >>> a!=b True</pre>	Returns True if all elements in both elements are same. Otherwise returns false

List slices:

- ❖ List slicing is an operation that extracts a subset of elements from a list and packages them as another list.

Syntax:

Listname[start:stop]
Listname[start:stop:steps]

- ❖ default start value is 0
- ❖ default stop value is n-1
- ❖ [:] this will print the entire list
- ❖ [2:2] this will create an empty slice

slices	example	description
a[0:3]	<pre>>>> a=[9,8,7,6,5,4] >>> a[0:3] [9, 8, 7]</pre>	Printing a part of a list from 0 to 2.
a[:4]	<pre>>>> a[:4] [9, 8, 7, 6]</pre>	Default start value is 0. so prints from 0 to 3
a[1:]	<pre>>>> a[1:] [8, 7, 6, 5, 4]</pre>	default stop value will be n-1. so prints from 1 to 5
a[:]	<pre>>>> a[:] [9, 8, 7, 6, 5, 4]</pre>	Prints the entire list.

<code>a[2:2]</code>	<code>>>> a[2:2] []</code>	print an empty slice
<code>a[0:6:2]</code>	<code>>>> a[0:6:2] [9, 7, 5]</code>	Slicing list values with step size 2.
<code>a[::-1]</code>	<code>>>> a[::-1] [4, 5, 6, 7, 8, 9]</code>	Returns reverse of given list values

List methods:

- ❖ Methods used in lists are used to manipulate the data quickly.
- ❖ These methods work only on lists.
- ❖ They do not work on the other sequence types that are not mutable, that is, the values they contain cannot be changed, added, or deleted.

syntax:

list name.method name(element/index/list)

	syntax	example	description
1	<code>a.append(element)</code>	<code>>>> a=[1,2,3,4,5] >>> a.append(6) >>> print(a) [1, 2, 3, 4, 5, 6]</code>	Add an element to the end of the list
2	<code>a.insert(index,element)</code>	<code>>>> a.insert(0,0) >>> print(a) [0, 1, 2, 3, 4, 5, 6]</code>	Insert an item at the defined index
3	<code>a.extend(b)</code>	<code>>>> b=[7,8,9] >>> a.extend(b) >>> print(a) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]</code>	Add all elements of a list to the another list
4	<code>a.index(element)</code>	<code>>>> a.index(8) 8</code>	Returns the index of the first matched item
5	<code>a.sort()</code>	<code>>>> a.sort() >>> print(a) [0, 1, 2, 3, 4, 5, 6, 7, 8]</code>	Sort items in a list in ascending order
6	<code>a.reverse()</code>	<code>>>> a.reverse() >>> print(a) [8, 7, 6, 5, 4, 3, 2, 1, 0]</code>	Reverse the order of items in the list

7	a.pop()	>>> a.pop() 0	Removes and returns an element at the last element
8	a.pop(index)	>>> a.pop(0) 8	Remove the particular element and return it.
9	a.remove(element)	>>> a.remove(1) >>> print(a) [7, 6, 5, 4, 3, 2]	Removes an item from the list
10	a.count(element)	>>> a.count(6) 1	Returns the count of number of items passed as an argument
11	a.copy()	>>> b=a.copy() >>> print(b) [7, 6, 5, 4, 3, 2]	Returns a shallow copy of the list
12	len(list)	>>> len(a) 6	return the length of the length
13	min(list)	>>> min(a) 2	return the minimum element in a list
14	max(list)	>>> max(a) 7	return the maximum element in a list.
15	a.clear()	>>> a.clear() >>> print(a) []	Removes all items from the list.
16	del(a)	>>> del(a) >>> print(a) Error: name 'a' is not defined	delete the entire list.

List loops:

1. For loop
2. While loop
3. Infinite loop

List using For Loop:

- ❖ The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects.
- ❖ Iterating over a sequence is called traversal.
- ❖ Loop continues until we reach the last item in the sequence.
- ❖ The body of for loop is separated from the rest of the code **using indentation**.

Syntax: for val in sequence:

Accessing element	output
a=[10,20,30,40,50] for i in a: print(i)	1 2 3 4 5
Accessing index	output
a=[10,20,30,40,50] for i in range(0,len(a),1): print(i)	0 1 2 3 4
Accessing element using range:	output
a=[10,20,30,40,50] for i in range(0,len(a),1): print(a[i])	10 20 30 40 50

List using While loop

- ❖ The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
- ❖ When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Syntax: while (condition): body of while

Sum of elements in list	Output:
a=[1,2,3,4,5] i=0 sum=0 while i<len(a): sum=sum+a[i] i=i+1 print(sum)	15

Infinite Loop

A loop becomes infinite loop if the condition given never becomes false. It keeps on running. Such loops are called infinite loop.

Example	Output:
<pre>a=1 while (a==1): n=int(input("enter the number")) print("you entered:", n)</pre>	<pre>Enter the number 10 you entered:10 Enter the number 12 you entered:12 Enter the number 16 you entered:16</pre>

Mutability:

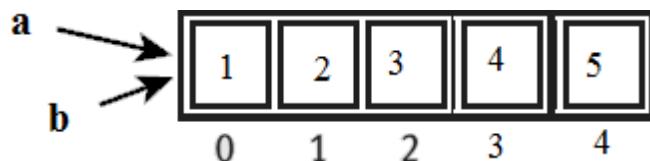
- ❖ Lists are mutable. (can be changed)
- ❖ Mutability is the ability for certain types of data to be changed without entirely recreating it.
- ❖ An item can be changed in a list by accessing it directly as part of the assignment statement.
- ❖ Using the indexing operator (square brackets[]) on the left side of an assignment, one of the list items can be updated.

Example	description
<pre>>>> a=[1,2,3,4,5] >>> a[0]=100 >>> print(a) [100, 2, 3, 4, 5]</pre>	changing single element
<pre>>>> a=[1,2,3,4,5] >>> a[0:3]=[100,100,100] >>> print(a) [100, 100, 100, 4, 5]</pre>	changing multiple element
<pre>>>> a=[1,2,3,4,5] >>> a[0:3]=[] >>> print(a) [4, 5]</pre>	The elements from a list can also be removed by assigning the empty list to them.
<pre>>>> a=[1,2,3,4,5] >>> a[0:0]=[20,30,45] >>> print(a) [20,30,45,1, 2, 3, 4, 5]</pre>	The elements can be inserted into a list by squeezing them into an empty slice at the desired location.

Aliasing(copying):

- ❖ Creating a copy of a list is called aliasing. When you create a copy both list will be having same memory location. changes in one list will affect another list.
- ❖ **Alaisng refers to having different names for same list values.**

Example	Output:
a=[1, 2, 3 ,4 ,5] b=a print (b) a is b a[0]=100 print(a) print(b)	[1, 2, 3, 4, 5] True [100,2,3,4,5] [100,2,3,4,5]



- ❖ In this a single list object is created and modified using the subscript operator.
- ❖ When the first element of the list named “a” is replaced, the first element of the list named “b” is also replaced.
- ❖ This type of change is what is known as a **side effect**. This happens because after the assignment **b=a**, the variables **a** and **b** refer to the exact same list object.
- ❖ They are **aliases** for the same object. This phenomenon is known as **aliasing**.
- ❖ To prevent aliasing, a new object can be created and the contents of the original can be copied which is called **cloning**.

Clonning:

- ❖ To avoid the disadvantages of copying we are using cloning. creating a copy of a same list of elements with two different memory locations is called cloning.
- ❖ Changes in one list will not affect locations of aother list.
- ❖ Cloning is a process of making a copy of the list without modifying the original list.

1. Slicing
2. list()method
3. copy() method

clonning using Slicing

```
>>>a=[1,2,3,4,5]
>>>b=a[:]
>>>print(b)
[1,2,3,4,5]
>>>a is b
False
```

clonning using List() method

```
>>>a=[1,2,3,4,5]
>>>b=list
>>>print(b)
[1,2,3,4,5]
>>>a is b
false
>>>a[0]=100
>>>print(a)
>>>a=[100,2,3,4,5]
>>>print(b)
>>>b=[1,2,3,4,5]
```

clonning using copy() method

```
a=[1,2,3,4,5]
>>>b=a.copy()
>>> print(b)
[1, 2, 3, 4, 5]
>>> a is b
False
```

List as parameters:

- ❖ In python, arguments are passed by reference.
- ❖ If any changes are done in the parameter which refers within the function, then the changes also reflects back in the calling function.
- ❖ When a list to a function is passed, the function gets a reference to the list.
- ❖ Passing a list as an argument actually passes a reference to the list, not a copy of the list.
- ❖ Since lists are mutable, changes made to the elements referenced by the parameter change the same list that the argument is referencing.

Example 1`:

```
def remove(a):
    a.remove(1)
a=[1,2,3,4,5]
remove(a)
print(a)
```

Output

```
[2,3,4,5]
```

Example 2:	Output
<pre>def inside(a): for i in range(0,len(a),1): a[i]=a[i]+10 print("inside",a) a=[1,2,3,4,5] inside(a) print("outside",a)</pre>	inside [11, 12, 13, 14, 15] outside [11, 12, 13, 14, 15]
Example 3	output
<pre>def insert(a): a.insert(0,30) a=[1,2,3,4,5] insert(a) print(a)</pre>	[30, 1, 2, 3, 4, 5]

Tuple:

- ❖ A tuple is same as list, except that the set of elements is enclosed in parentheses instead of square brackets.
- ❖ A tuple is an immutable list. i.e. once a tuple has been created, you can't add elements to a tuple or remove elements from the tuple.
- ❖ **But tuple can be converted into list and list can be converted in to tuple.**

methods	example	Description
list()	<pre>>>> a=(1,2,3,4,5) >>> a=list(a) >>> print(a) [1, 2, 3, 4, 5]</pre>	it convert the given tuple into list.
tuple()	<pre>>>> a=[1,2,3,4,5] >>> a=tuple(a) >>> print(a) (1, 2, 3, 4, 5)</pre>	it convert the given list into tuple.

Benefit of Tuple:

- ❖ Tuples are faster than lists.
- ❖ If the user wants to protect the data from accidental changes, tuple can be used.
- ❖ Tuples can be used as keys in dictionaries, while lists can't.

Operations on Tuples:

1. Indexing
2. Slicing
3. Concatenation
4. Repetitions
5. Membership
6. Comparison

Operations	examples	Description
Creating a tuple	>>>a=(20,40,60,"apple","ball")	Creating the tuple with elements of different data types.
Indexing	>>>print(a[0]) 20 >>> a[2] 60	Accessing the item in the position 0 Accessing the item in the position 2
Slicing	>>>print(a[1:3]) (40,60)	Displaying items from 1st till 2nd.
Concatenation	>>> b=(2,4) >>>print(a+b) >>>(20,40,60,"apple","ball",2,4)	Adding tuple elements at the end of another tuple elements
Repetition	>>>print(b*2) >>>(2,4,2,4)	repeating the tuple in n no of times
Membership	>>> a=(2,3,4,5,6,7,8,9,10) >>> 5 in a True >>> 100 in a False >>> 2 not in a False	Returns True if element is present in tuple. Otherwise returns false.
Comparison	>>> a=(2,3,4,5,6,7,8,9,10) >>>b=(2,3,4) >>> a==b False >>> a!=b True	Returns True if all elements in both elements are same. Otherwise returns false

Tuple methods:

- ❖ Tuple is immutable so changes cannot be done on the elements of a tuple once it is assigned.

methods	example	Description
a.index(tuple)	>>> a=(1,2,3,4,5) >>> a.index(5) 4	Returns the index of the first matched item.
a.count(tuple)	>>>a=(1,2,3,4,5) >>> a.count(3) 1	Returns the count of the given element.
len(tuple)	>>> len(a) 5	return the length of the tuple

min(tuple)	>>> min(a) 1	return the minimum element in a tuple
max(tuple)	>>> max(a) 5	return the maximum element in a tuple
del(tuple)	>>> del(a)	Delete the entire tuple.

Tuple Assignment:

- ❖ Tuple assignment allows, variables on the left of an assignment operator and values of tuple on the right of the assignment operator.
- ❖ Multiple assignment works by creating a tuple of expressions from the right hand side, and a tuple of targets from the left, and then matching each expression to a target.
- ❖ Because multiple assignments use tuples to work, it is often termed tuple assignment.

Uses of Tuple assignment:

- ❖ It is often useful to swap the values of two variables.

Example:

Swapping using temporary variable:	Swapping using tuple assignment:
a=20 b=50 temp = a a = b b = temp print("value after swapping is",a,b)	a=20 b=50 (a,b)=(b,a) print("value after swapping is",a,b)

Multiple assignments:

Multiple values can be assigned to multiple variables using tuple assignment.

```
>>>(a,b,c)=(1,2,3)
>>>print(a)
1
>>>print(b)
2
>>>print(c)
3
```

Tuple as return value:

- ❖ A Tuple is a comma separated sequence of items.
- ❖ It is created with or without ().
- ❖ A function can return one value. if you want to return more than one value from a function. we can use tuple as return value.

Example1:	Output:
<pre>def div(a,b): r=a%b q=a//b return(r,q) a=eval(input("enter a value:")) b=eval(input("enter b value:")) r,q=div(a,b) print("reminder:",r) print("quotient:",q)</pre>	enter a value:4 enter b value:3 reminder: 1 quotient: 1
Example2:	Output:
<pre>def min_max(a): small=min(a) big=max(a) return(small,big) a=[1,2,3,4,6] small,big=min_max(a) print("smallest:",small) print("biggest:",big)</pre>	smallest: 1 biggest: 6

Tuple as argument:

- ❖ The parameter name that begins with * gathers argument into a tuple.

Example:	Output:
<pre>def printall(*args): print(args) printall(2,3,'a')</pre>	(2, 3, 'a')

Dictionaries:

- ❖ Dictionary is an unordered collection of elements. An element in dictionary has a key: value pair.
- ❖ All elements in dictionary are placed inside the curly braces i.e. { }
- ❖ Elements in Dictionaries are **accessed via keys** and not by their position.
- ❖ The values of a dictionary can be any data type.
- ❖ Keys must be immutable data type (numbers, strings, tuple)

Operations on dictionary:

1. Accessing an element
2. Update
3. Add element
4. Membership

Operations	Example	Description
Creating a dictionary	>>> a={"one":1,"two":2} >>> print(a) {'one': 1, 'two': 2}	Creating the dictionary with elements of different data types.
accessing an element	>>> a[1] 'one' >>> a[0] KeyError: 0	Accessing the elements by using keys.
Update	>>> a[1] = "ONE" >>> print(a) {'one': 'ONE', 'two': 2}	Assigning a new value to key. It replaces the old value by new value.
add element	>>> a[3] = "three" >>> print(a) {'one': 'ONE', 'two': 2, 'three': 'three'}	Add new element in to the dictionary with key.
membership	a={1: 'ONE', 2: 'two', 3: 'three'} >>> 1 in a True >>> 3 not in a False	Returns True if the key is present in dictionary. Otherwise returns false.

Methods in dictionary:

Method	Example	Description
a.copy()	a={1: 'ONE', 2: 'two', 3: 'three'} >>> b=a.copy() >>> print(b) {'one': 'ONE', 'two': 'two', 'three': 'three'}	It returns copy of the dictionary. here copy of dictionary 'a' get stored in to dictionary 'b'
a.items()	>>> a.items() dict_items([(1, 'ONE'), (2, 'two'), (3, 'three')])	Return a new view of the dictionary's items. It displays a list of dictionary's (key, value) tuple pairs.
a.keys()	>>> a.keys() dict_keys([1, 2, 3])	It displays list of keys in a dictionary
a.values()	>>> a.values() dict_values(['ONE', 'two', 'three'])	It displays list of values in dictionary
a.pop(key)	>>> a.pop(3) 'three' >>> print(a) {'one': 'ONE', 'two': 'two'}	Remove the element with <i>key</i> and return its value from the dictionary.

setdefault(key,value)	>>> a.setdefault(3,"three") 'three' >>> print(a) {1: 'ONE', 2: 'two', 3: 'three'} >>> a.setdefault(2) 'two'	If key is in the dictionary, return its value. If key is not present, insert key with a value of dictionary and return dictionary.
a.update(dictionary)	>>> b={4:"four"} >>> a.update(b) >>> print(a) {1: 'ONE', 2: 'two', 3: 'three', 4: 'four'}	It will add the dictionary with the existing dictionary
fromkeys()	>>> key={"apple", "ball"} >>> value="for kids" >>> d=dict.fromkeys(key,value) >>> print(d) {'apple': 'for kids', 'ball': 'for kids'}	It creates a dictionary from key and values.
len(a)	a={1: 'ONE', 2: 'two', 3: 'three'} >>> len(a) 3	It returns the length of the list.
clear()	a={1: 'ONE', 2: 'two', 3: 'three'} >>> a.clear() >>> print(a) >>> {}	Remove all elements form the dictionary.
del(a)	a={1: 'ONE', 2: 'two', 3: 'three'} >>> del(a)	It will delete the entire dictionary.

Difference between List, Tuples and dictionary:

List	Tuples	Dictionary
A list is mutable	A tuple is immutable	A dictionary is mutable
Lists are dynamic	Tuples are fixed size in nature	In values can be of any data type and can repeat, keys must be of immutable type
List are enclosed in brackets[] and their elements and size can be changed	Tuples are enclosed in parenthesis () and cannot be updated	Tuples are enclosed in curly braces { } and consist of key:value
Homogenous	Heterogeneous	Homogenous
Example: List = [10, 12, 15]	Example: Words = ("spam", "eggs") Or Words = "spam", "eggs"	Example: Dict = {"ram": 26, "abi": 24}
<u>Access:</u> print(list[0])	<u>Access:</u> print(words[0])	<u>Access:</u> print(dict["ram"])

Can contain duplicate elements	Can contain duplicate elements. Faster compared to lists	Cant contain duplicate keys, but can contain duplicate values
Slicing can be done	Slicing can be done	Slicing can't be done
<u>Usage:</u> <ul style="list-style-type: none"> ❖ List is used if a collection of data that doesn't need random access. ❖ List is used when data can be modified frequently 	<u>Usage:</u> <ul style="list-style-type: none"> ❖ Tuple can be used when data cannot be changed. ❖ A tuple is used in combination with a dictionary i.e. a tuple might represent a key. 	<u>Usage:</u> <ul style="list-style-type: none"> ❖ Dictionary is used when a logical association between key:value pair. ❖ When in need of fast lookup for data, based on a custom key. ❖ Dictionary is used when data is being constantly modified.

Advanced list processing:

List Comprehension:

- ❖ List comprehensions provide a concise way to apply operations on a list.
- ❖ It creates a new list in which each element is the result of applying a given operation in a list.
- ❖ It consists of brackets containing an expression followed by a “for” clause, then a list.
- ❖ The list comprehension always returns a result list.

Syntax

list=[expression for item in list if conditional]

List Comprehension	Output
<pre>>>>L=[x**2 for x in range(0,5)] >>>print(L)</pre>	[0, 1, 4, 9, 16]
<pre>>>>[x for x in range(1,10) if x%2==0]</pre>	[2, 4, 6, 8]
<pre>>>>[x for x in 'Python Programming' if x in ['a','e','i','o','u']]</pre>	['o', 'o', 'a', 'i']
<pre>>>>mixed=[1,2,"a",3,4.2] >>> [x**2 for x in mixed if type(x)==int]</pre>	[1, 4, 9]
<pre>>>>[x+3 for x in [1,2,3]]</pre>	[4, 5, 6]
<pre>>>> [x*x for x in range(5)]</pre>	[0, 1, 4, 9, 16]
<pre>>>> num=[-1,2,-3,4,-5,6,-7] >>> [x for x in num if x>=0]</pre>	[2, 4, 6]
<pre>>>> str=["this","is","an","example"] >>> element=[word[0] for word in str] >>> print(element)</pre>	['t', 'i', 'a', 'e']

Nested list:

List inside another list is called nested list.

Example:

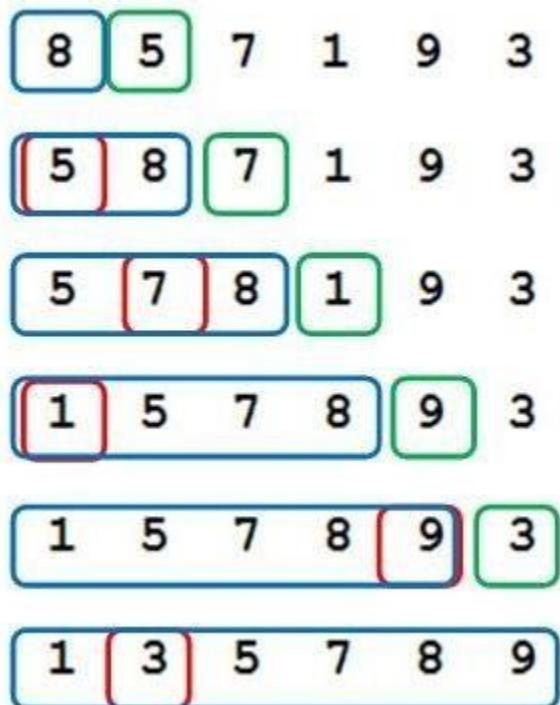
```
>>> a=[56,34,5,[34,57]]
>>> a[0]
56
>>> a[3]
[34, 57]
>>> a[3][0]
34
>>> a[3][1]
57
```

Programs on matrix:

Matrix addition	Output
<pre>a=[[1,1],[1,1]] b=[[2,2],[2,2]] c=[[0,0],[0,0]] for i in range(len(a)): for j in range(len(b)): c[i][j]=a[i][j]+b[i][j] for i in c: print(i)</pre>	[3, 3] [3, 3]
Matrix multiplication	Output
<pre>a=[[1,1],[1,1]] b=[[2,2],[2,2]] c=[[0,0],[0,0]] for i in range(len(a)): for j in range(len(b)): for k in range(len(b)): c[i][j]=a[i][j]+a[i][k]*b[k][j] for i in c: print(i)</pre>	[3, 3] [3, 3]
Matrix transpose	Output
<pre>a=[[1,3],[1,2]] c=[[0,0],[0,0]] for i in range(len(a)): for j in range(len(a)): c[i][j]=a[j][i] for i in c: print(i)</pre>	[1, 1] [3, 2]

Illustrative programs:

Selection sort	Output
<pre>a=input("Enter list:").split() a=list(map(eval,a)) for i in range(0,len(a)): smallest = min(a[i:]) sindex= a.index(smallest) a[i],a[sindex] = a[sindex],a[i] print (a)</pre>	<pre>Enter list:23 78 45 8 32 56 [8,23,32,45,56,78]</pre>
Insertion sort	output
<pre>a=input("enter a list:").split() a=list(map(int,a)) for i in a: j = a.index(i) while j>0: if a[j-1] > a[j]: a[j-1],a[j] = a[j],a[j-1] else: break j = j-1 print (a)</pre>	<pre>enter a list: 8 5 7 1 9 3 [1,3,5,7,8,9]</pre>



Merge sort

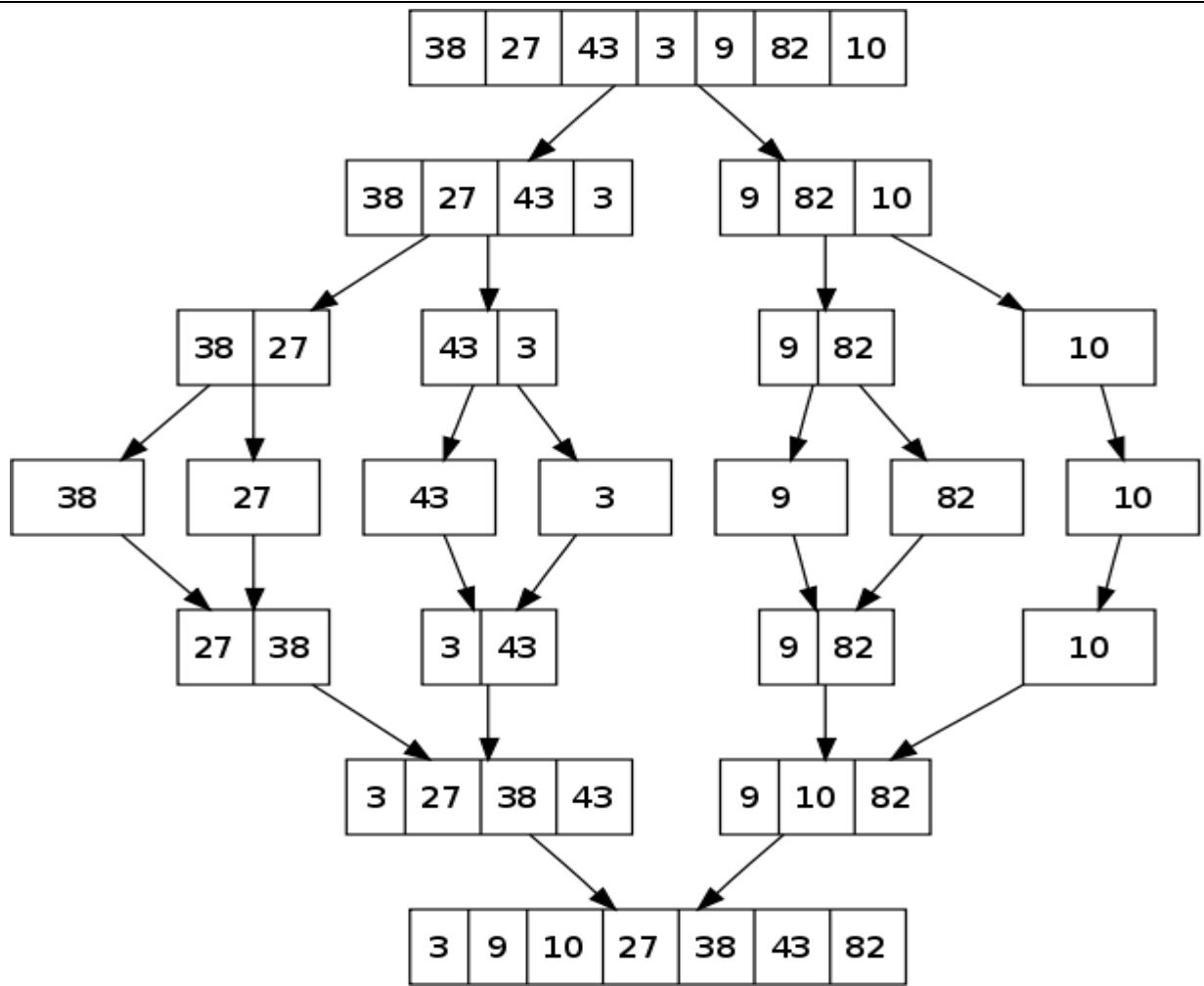
```
def merge(a,b):
    c = []
    while len(a) != 0 and len(b) != 0:
        if a[0] < b[0]:
            c.append(a[0])
            a.remove(a[0])
        else:
            c.append(b[0])
            b.remove(b[0])
    if len(a) == 0:
        c=c+b
    else:
        c=c+a
    return c
```

```
def divide(x):
    if len(x) == 0 or len(x) == 1:
        return x
    else:
        middle = len(x)//2
        a = divide(x[:middle])
        b = divide(x[middle:])
        return merge(a,b)
```

```
x=[38,27,43,3,9,82,10]
c=divide(x)
print(c)
```

output

[3,9,10,27,38,43,82]



Histogram	Output
<pre>def histogram(a): for i in a: sum = "" while(i>0): sum=sum+"#" i=i-1 print(sum) a=[4,5,7,8,12] histogram(a)</pre>	<pre>***** ***** ***** ***** *****</pre>
Calendar program	Output
<pre>import calendar y=int(input("enter year:")) m=int(input("enter month:")) print(calendar.month(y,m))</pre>	<pre>enter year:2017 enter month:11 November 2017 Mo Tu We Th Fr Sa Su 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30</pre>

Basic python programs:

Addition of two numbers	Output
<pre>a=eval(input("enter first no")) b=eval(input("enter second no")) c=a+b print("the sum is ",c)</pre>	enter first no 5 enter second no 6 the sum is 11
Area of rectangle	Output
<pre>l=eval(input("enter the length of rectangle")) b=eval(input("enter the breath of rectangle")) a=l*b print(a)</pre>	enter the length of rectangle 5 enter the breath of rectangle 6 30
Area & circumference of circle	output
<pre>r=eval(input("enter the radius of circle")) a=3.14*r*r c=2*3.14*r print("the area of circle",a) print("the circumference of circle",c)</pre>	enter the radius of circle4 the area of circle 50.24 the circumference of circle 25.12
Calculate simple interest	Output
<pre>p=eval(input("enter principle amount")) n=eval(input("enter no of years")) r=eval(input("enter rate of interest")) si=p*n*r/100 print("simple interest is",si)</pre>	enter principle amount 5000 enter no of years 4 enter rate of interest6 simple interest is 1200.0
Calculate engineering cutoff	Output
<pre>p=eval(input("enter physics marks")) c=eval(input("enter chemistry marks")) m=eval(input("enter maths marks")) cutoff=(p/4+c/4+m/2) print("cutoff =",cutoff)</pre>	enter physics marks 100 enter chemistry marks 99 enter maths marks 96 cutoff = 97.75
Check voting eligibility	output
<pre>age=eval(input("enter ur age")) If(age>=18): print("eligible for voting") else: print("not eligible for voting")</pre>	Enter ur age 19 Eligible for voting

Find greatest of three numbers	output
<pre>a=eval(input("enter the value of a")) b=eval(input("enter the value of b")) c=eval(input("enter the value of c")) if(a>b): if(a>c): print("the greatest no is",a) else: print("the greatest no is",c) else: if(b>c): print("the greatest no is",b) else: print("the greatest no is",c)</pre>	enter the value of a 9 enter the value of a 1 enter the value of a 8 the greatest no is 9

Programs on for loop

Print n natural numbers	Output
<pre>for i in range(1,5,1): print(i)</pre>	1 2 3 4
Print n odd numbers	Output
<pre>for i in range(1,10,2): print(i)</pre>	1 3 5 7 9
Print n even numbers	Output
<pre>for i in range(2,10,2): print(i)</pre>	2 4 6 8
Print squares of numbers	Output
<pre>for i in range(1,5,1): print(i*i)</pre>	1 4 9 16
Print squares of numbers	Output
<pre>for i in range(1,5,1): print(i*i*i)</pre>	1 8 27 64

Programs on while loop

Print n natural numbers	Output
i=1 while(i<=5): print(i) i=i+1	1 2 3 4 5
Print n odd numbers	Output
i=2 while(i<=10): print(i) i=i+2	2 4 6 8 10
Print n even numbers	Output
i=1 while(i<=10): print(i) i=i+2	1 3 5 7 9
Print n squares of numbers	Output
i=1 while(i<=5): print(i*i) i=i+1	1 4 9 16 25
Print n cubes numbers	Output
i=1 while(i<=3): print(i*i*i) i=i+1	1 8 27
find sum of n numbers	Output
i=1 sum=0 while(i<=10): sum=sum+i i=i+1 print(sum)	55

factorial of n numbers/product of n numbers	Output
<pre>i=1 product=1 while(i<=10): product=product*i i=i+1 print(product)</pre>	3628800
sum of n numbers	Output
<pre>def add(): a=eval(input("enter a value")) b=eval(input("enter b value")) c=a+b print("the sum is",c) add()</pre>	enter a value 6 enter b value 4 the sum is 10
area of rectangle using function	Output
<pre>def area(): l=eval(input("enter the length of rectangle")) b=eval(input("enter the breath of rectangle")) a=l*b print("the area of rectangle is",a) area()</pre>	enter the length of rectangle 20 enter the breath of rectangle 5 the area of rectangle is 100
swap two values of variables	Output
<pre>def swap(): a=eval(input("enter a value")) b=eval(input("enter b value")) c=a a=b b=c print("a=",a,"b=",b) swap()</pre>	enter a value3 enter b value5 a= 5 b= 3

check the no divisible by 5 or not	Output
<pre>def div(): n=eval(input("enter n value")) if(n%5==0): print("the number is divisible by 5") else: print("the number not divisible by 5") div()</pre>	enter n value10 the number is divisible by 5
find remainder and quotient of given no	Output
<pre>def reminder(): a=eval(input("enter a")) b=eval(input("enter b")) R=a%b print("the remainder is",R) def quotient(): a=eval(input("enter a")) b=eval(input("enter b")) Q=a/b print("the remainder is",Q) reminder() quotient()</pre>	enter a 6 enter b 3 the remainder is 0 enter a 8 enter b 4 the remainder is 2.0
convert the temperature	Output
<pre>def ctof(): c=eval(input("enter temperature in centigrade")) f=(1.8*c)+32 print("the temperature in Fahrenheit is",f) def ftoc(): f=eval(input("enter temp in Fahrenheit")) c=(f-32)/1.8 print("the temperature in centigrade is",c) ctof() ftoc()</pre>	enter temperature in centigrade 37 the temperature in Fahrenheit is 98.6 enter temp in Fahrenheit 100 the temperature in centigrade is 37.77

program for basic calculator	Output
<pre> def add(): a=eval(input("enter a value")) b=eval(input("enter b value")) c=a+b print("the sum is",c) def sub(): a=eval(input("enter a value")) b=eval(input("enter b value")) c=a-b print("the diff is",c) def mul(): a=eval(input("enter a value")) b=eval(input("enter b value")) c=a*b print("the mul is",c) def div(): a=eval(input("enter a value")) b=eval(input("enter b value")) c=a/b print("the div is",c) add() sub() mul() div() </pre>	<pre> enter a value 10 enter b value 10 the sum is 20 enter a value 10 enter b value 10 the diff is 0 enter a value 10 enter b value 10 the mul is 100 enter a value 10 enter b value 10 the div is 1 </pre>

1. INTRODUCTION TO PYTHON:

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language.

It was created by Guido van Rossum during 1985- 1990.

Python got its name from “Monty Python’s flying circus”. Python was released in the year 2000.

- ❖ **Python is interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.
- ❖ **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- ❖ **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- ❖ **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications.

1.1. Python Features:

- ❖ **Easy-to-learn:** Python is clearly defined and easily readable. The structure of the program is very simple. It uses few keywords.
- ❖ **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
- ❖ **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- ❖ **Interpreted:** Python is processed at runtime by the interpreter. So, there is no need to compile a program before executing it. You can simply run the program.
- ❖ **Extensible:** Programmers can embed python within their C,C++,Java script ,ActiveX, etc.
- ❖ **Free and Open Source:** Anyone can freely distribute it, read the source code, and edit it.
- ❖ **High Level Language:** When writing programs, programmers concentrate on solutions of the current problem, no need to worry about the low level details.
- ❖ **Scalable:** Python provides a better structure and support for large programs than shell scripting.

1.2. Applications:

- ❖ Bit Torrent file sharing
- ❖ Google search engine, Youtube
- ❖ Intel, Cisco, HP, IBM
- ❖ i-Robot
- ❖ NASA

- ❖ Facebook, Drop box

1.3. Python interpreter:

Interpreter: To execute a program in a high-level language by translating it one line at a time.

Compiler: To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.

Compiler	Interpreter
Compiler Takes Entire program as input	Interpreter Takes Single instruction as input
Intermediate Object Code is Generated	No Intermediate Object Code is Generated
Conditional Control Statements are Executes faster	Conditional Control Statements are Executes slower
Memory Requirement is More (Since Object Code is Generated)	Memory Requirement is Less
Program need not be compiled every time	Every time higher level program is converted into lower level program
Errors are displayed after entire program is checked	Errors are displayed for every instruction interpreted (if any)
Example : C Compiler	Example : PYTHON

1.4 MODES OF PYTHON INTERPRETER:

Python Interpreter is a program that reads and executes Python code. It uses 2 modes of Execution.

1. Interactive mode
2. Script mode

Interactive mode:

- ❖ Interactive Mode, as the name suggests, allows us to interact with OS.
- ❖ When we type Python statement, **interpreter displays the result(s) immediately.**

Advantages:

- ❖ Python, in interactive mode, is good enough to learn, experiment or explore.
- ❖ Working in interactive mode is convenient for beginners and for testing small pieces of code.

Drawback:

- ❖ We cannot save the statements and have to retype all the statements once again to re-run them.

In interactive mode, you type Python programs and the interpreter displays the result:

```
>>> 1 + 1
```

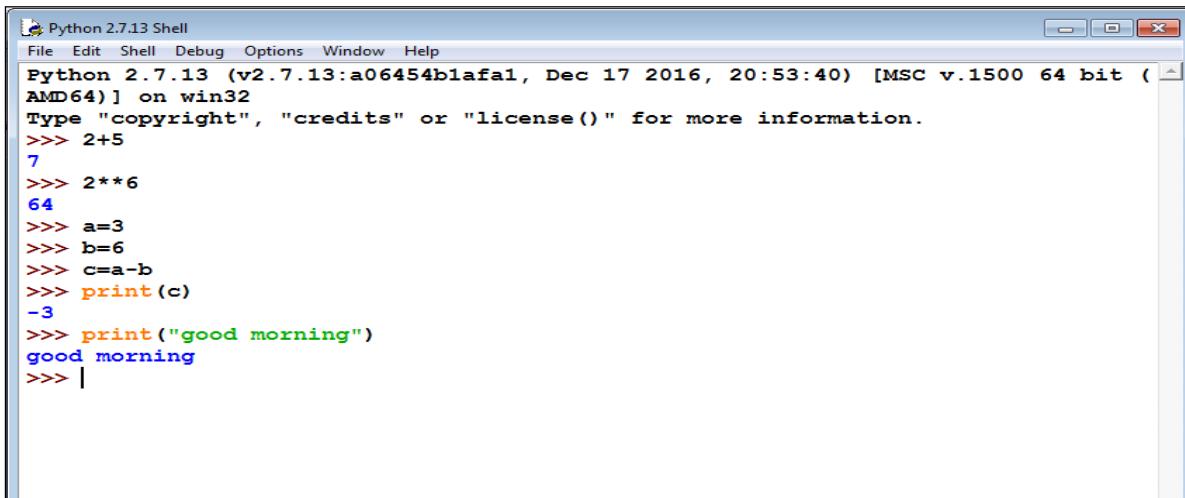
```
2
```

The chevron, >>>, is the prompt the interpreter uses to indicate that it is ready for you to enter code. If you type 1 + 1, the interpreter replies 2.

```
>>> print ('Hello, World!')
```

```
Hello, World!
```

This is an example of a print statement. It displays a result on the screen. In this case, the result is the words.



The screenshot shows the Python 2.7.13 Shell window. The code entered is:

```
Python 2.7.13 (v2.7.13:a06454b1afaf1, Dec 17 2016, 20:53:40) [MSC v.1500 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.

>>> 2+5
7
>>> 2**6
64
>>> a=3
>>> b=6
>>> c=a-b
>>> print(c)
-3
>>> print("good morning")
good morning
>>> |
```

Script mode:

- ❖ In script mode, we type python program in a file and then use interpreter to execute the content of the file.
- ❖ Scripts can be saved to disk for future use. **Python scripts have the extension .py**, meaning that the filename ends with .py
- ❖ Save the code with **filename.py** and run the interpreter in script mode to execute the script.

Example:

```
print(1)
x = 2
print(x)
```

Output:

```
>>>1
2
```

Interactive mode	Script mode
A way of using the Python interpreter by typing commands and expressions at the prompt.	A way of using the Python interpreter to read and execute statements in a script.
Cant save and edit the code	Can save and edit the code
If we want to experiment with the code, we can use interactive mode.	If we are very clear about the code, we can use script mode.
we cannot save the statements for further use and we have to retype all the statements to re-run them.	we can save the statements for further use and we no need to retype all the statements to re-run them.
We can see the results immediately.	We cant see the code immediately.

Integrated Development Learning Environment (IDLE):

- ❖ Is a **graphical user interface** which is completely written in Python.
- ❖ It is bundled with the default implementation of the python language and also comes with optional part of the Python packaging.

Features of IDLE:

- ❖ Multi-window text editor with syntax highlighting.

- ❖ Auto completion with **smart indentation**.
- ❖ **Python shell** to display output with syntax highlighting.

2. VALUES AND DATA TYPES

Value:

Value can be any letter ,number or string.

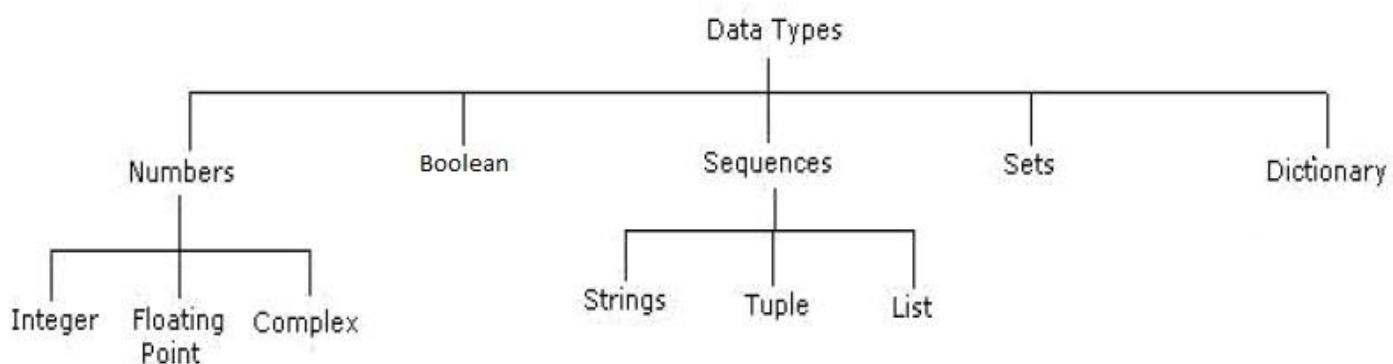
Eg, Values are 2, 42.0, and 'Hello, World!'. (These values belong to different datatypes.)

Data type:

Every value in Python has a data type.

It is a set of values, and the allowable operations on those values.

Python has four standard data types:



2.1 Numbers:

- ❖ Number data type stores **Numerical Values**.
- ❖ This data type is immutable [i.e. values/items cannot be changed].
- ❖ Python supports integers, floating point numbers and complex numbers. They are defined as,

Integers	Long	Float	Complex
<ul style="list-style-type: none"> - They are often called just integers or int. - They are positive or negative whole numbers with no decimal point. 	<ul style="list-style-type: none"> -They are long integers. -They can also be represented in octal and hexadecimal representation. 	<ul style="list-style-type: none"> -They are written with a decimal point dividing the integer and the fractional parts. 	<ul style="list-style-type: none"> -They are of the form a + bj, where a and b are floats and j represents the square root of -1 (which is an imaginary number). -The real part of the number is a, and the imaginary part is b.
Eg, 56	Eg, 5692431L	Eg, 56.778	Eg, square root of -1 is a complex number

2.2 Sequence:

- ❖ A sequence is an **ordered collection of items**, indexed by positive integers.
- ❖ It is a combination of **mutable** (value can be changed) and **immutable** (values cannot be changed) data types.

- ❖ There are three types of sequence data type available in Python, they are

1. **Strings**
2. **Lists**
3. **Tuples**

2.2.1 Strings:

- A String in Python consists of a series or sequence of characters - letters, numbers, and special characters.
 - Strings are marked by quotes:
 - single quotes (' ') Eg, 'This a string in single quotes'
 - double quotes (" ") Eg, ""This a string in double quotes""
 - triple quotes("""" """") Eg, This is a paragraph. It is made up of multiple lines and sentences.""""
 - Individual character in a string is accessed using a subscript (index).
 - Characters can be accessed using indexing and slicing operations
- Strings are immutable i.e. the contents of the string cannot be changed after it is created.

Indexing:

String A	H	E	L	L	O
Positive Index	0	1	2	3	4
Negative Index	-5	-4	-3	-2	-1

- Positive indexing helps in accessing the string from the beginning
- Negative subscript helps in accessing the string from the end.
- Subscript 0 or -ve n(where n is length of the string) displays the first element.
Example: A[0] or A[-5] will display "H"
- Subscript 1 or -ve (n-1) displays the second element.
Example: A[1] or A[-4] will display "E"

Operations on string:

- i. Indexing
- ii. Slicing
- iii. Concatenation
- iv. Repetitions
- v. Member ship

Creating a string	>>> s="good morning"	Creating the list with elements of different data types.
Indexing	>>> print(s[2]) o >>> print(s[6]) o	❖ Accessing the item in the position 0 ❖ Accessing the item in the position 2
Slicing(ending position -1)	>>> print(s[2:]) od morning	- Displaying items from 2 nd till last.

<u>Slice operator is used to extract part of a data type</u>	>>> print(s[:4]) Good	- Displaying items from 1 st position till 3 rd .
Concatenation	>>> print(s+"friends") good morningfriends	-Adding and printing the characters of two strings.
Repetition	>>> print(s*2) good morninggood morning	Creates new strings, concatenating multiple copies of the same string
in, not in (membership operator)	>>> s="good morning" >>>"m" in s True >>> "a" not in s True	Using membership operators to check a particular character is in string or not. Returns true if present.

2.2.2 Lists

- ❖ List is an ordered sequence of items. Values in the list are called elements / items.
- ❖ It can be written as a list of comma-separated items (values) between **square brackets[]**.
- ❖ Items in the lists can be of different data types.

Operations on list:

Indexing
 Slicing
 Concatenation
 Repetitions
 Updation, Insertion, Deletion

Creating a list	>>> list1=["python", 7.79, 101, "hello"] >>> list2=["god",6.78,9]	Creating the list with elements of different data types.
Indexing	>>> print(list1[0]) python >>> list1[2] 101	<ul style="list-style-type: none"> ❖ Accessing the item in the position 0 ❖ Accessing the item in the position 2
Slicing(ending position -1) <u>Slice operator is used to extract part of a string, or some part of a list</u> <u>Python</u>	>>> print(list1[1:3]) [7.79, 101] >>> print(list1[1:]) [7.79, 101, 'hello']	<ul style="list-style-type: none"> - Displaying items from 1st till 2nd. - Displaying items from 1st position till last.
Concatenation	>>> print(list1+list2) ['python', 7.79, 101, 'hello', 'god',	-Adding and printing the items of two lists.

	6.78, 9]	
Repetition	<code>>>> list2*3 ['god', 6.78, 9, 'god', 6.78, 9, 'god', 6.78, 9]</code>	Creates new strings, concatenating multiple copies of the same string
Updating the list	<code>>>> list1[2]=45 >>> print(list1) ['python', 7.79, 45, 'hello']</code>	Updating the list using index value
Inserting an element	<code>>>> list1.insert(2,"program") >>> print(list1) ['python', 7.79, 'program', 45, 'hello']</code>	Inserting an element in 2 nd position
Removing an element	<code>>>> list1.remove(45) >>> print(list1) ['python', 7.79, 'program', 'hello']</code>	Removing an element by giving the element directly

2.2.4 Tuple:

- ❖ A tuple is same as list, except that the set of elements is **enclosed in parentheses** instead of square brackets.
- ❖ **A tuple is an immutable list.** i.e. once a tuple has been created, you can't add elements to a tuple or remove elements from the tuple.
- ❖ Benefit of Tuple:
- ❖ Tuples are faster than lists.
- ❖ If the user wants to protect the data from accidental changes, tuple can be used.
- ❖ Tuples can be used as keys in dictionaries, while lists can't.

Basic Operations:

Creating a tuple	<code>>>>t=("python", 7.79, 101, "hello")</code>	Creating the tuple with elements of different data types.
Indexing	<code>>>>print(t[0]) python >>> t[2] 101</code>	<ul style="list-style-type: none"> ❖ Accessing the item in the position 0 ❖ Accessing the item in the position 2
Slicing(ending position -1)	<code>>>>print(t[1:3]) (7.79, 101)</code>	❖ Displaying items from 1st till 2nd.
Concatenation	<code>>>> t+("ram", 67) ('python', 7.79, 101, 'hello', 'ram', 67)</code>	❖ Adding tuple elements at the end of another tuple elements
Repetition	<code>>>>print(t*2) ('python', 7.79, 101, 'hello', 'python', 7.79, 101, 'hello')</code>	❖ Creates new strings, concatenating multiple copies of the same string

Altering the tuple data type leads to error. Following error occurs when user tries to do.

```
>>> t[0] = "a"
Trace back (most recent call last):
  File "<stdin>", line 1, in <module>
Type Error: 'tuple' object does not support item assignment
```

2.3 Mapping

- This data type is unordered and mutable.
- Dictionaries fall under Mappings.

2.3.1 Dictionaries:

- ❖ Lists are ordered sets of objects, whereas **dictionaries are unordered sets**.
- ❖ Dictionary is created by using **curly brackets**. i.e. {}
- ❖ Dictionaries **are accessed via keys** and not via their position.
- ❖ A dictionary is an associative array (also known as hashes). Any key of the dictionary is associated (or mapped) to a value.
- ❖ The values of a dictionary can be any Python data type. So dictionaries are **unordered key-value-pairs**(The association of a key and a value is called a key-value pair)

Dictionaries don't support the sequence operation of the sequence data types like strings, tuples and lists.

Creating a dictionary	<pre>>>> food = {"ham": "yes", "egg": "yes", "rate": 450} >>> print(food) {'rate': 450, 'egg': 'yes', 'ham': 'yes'}</pre>	Creating the dictionary with elements of different data types.
Indexing	<pre>>>> print(food["rate"]) 450</pre>	Accessing the item with keys.
Slicing(ending position -1)	<pre>>>> print(t[1:3]) (7.79, 101)</pre>	Displaying items from 1st till 2nd.

If you try to access a key which doesn't exist, you will get an error message:

```
>>> words = {"house": "Haus", "cat": "Katze"}
>>> words["car"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'car'
```

Data type	Compile time	Run time
int	a=10	a=int(input("enter a"))
float	a=10.5	a=float(input("enter a"))
string	a="panimalar"	a=input("enter a string")
list	a=[20,30,40,50]	a=list(input("enter a list"))
tuple	a=(20,30,40,50)	a=tuple(input("enter a tuple"))

3. Variables,Keywords Expressions, Statements, Comments, Docstring ,Lines And Indentation, Quotation In Python, Tuple Assignment:

3.1VARIABLES:

- ❖ A variable allows us to store a value by assigning it to a name, which can be used later.
- ❖ Named memory locations to store values.
- ❖ Programmers generally choose names for their variables that are meaningful.
- ❖ It can be of any length. No space is allowed.
- ❖ We don't need to declare a variable before using it. In Python, we simply assign a value to a variable and it will exist.

Assigning value to variable:

Value should be given on the right side of assignment operator(=) and variable on left side.

```
>>>counter =45  
print(counter)
```

Assigning a single value to several variables simultaneously:

```
>>>a=b=c=100
```

Assigning multiple values to multiple variables:

```
>>>a,b,c=2,4,"ram"
```

3.2KEYWORDS:

- ❖ Keywords are the reserved words in Python.
- ❖ We cannot use a keyword as variable name, function name or any other identifier.
- ❖ They are used to define the syntax and structure of the Python language.
- ❖ Keywords are case sensitive.

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

3.3IDENTIFIERS:

Identifier is the name given to entities like class, functions, variables etc. in Python.

- ❖ Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).

- ❖ all are valid example.
- ❖ An identifier cannot start with a digit.
- ❖ Keywords cannot be used as identifiers.
- ❖ Cannot use special symbols like !, @, #, \$, % etc. in our identifier.
- ❖ Identifier can be of any length.

Example:

Names like myClass, var_1, and **this_is_a_long_variable**

Valid declarations	Invalid declarations
Num	Number 1
Num	num 1
Num1	addition of program
_NUM	1Num
NUM_temp2	Num.no
IF	if
Else	else

3.4 STATEMENTS AND EXPRESSIONS:

3.4.1 Statements:

- Instructions that a Python interpreter can execute are called **statements**.
- A statement is a unit of code like creating a variable or displaying a value.

```
>>> n = 17
>>> print(n)
```

Here, The first line is an assignment statement that gives a value to n.
The second line is a print statement that displays the value of n.

3.4.2 Expressions:

- An expression is a **combination of values, variables, and operators**.
- A value all by itself is considered an expression, and also a variable.
- So the following are all legal expressions:

```
>>> 42
42
>>> a=2
>>> a+3+2
7
>>> z=("hi"+"friend")
>>> print(z)
hifriend
```

3.5 INPUT AND OUTPUT

INPUT: Input is data entered by user (end user) in the program.

In python, **input () function** is available for input.

Syntax for input() is:

```
variable = input ("data")
```

Example:

```
>>> x=input("enter the name:")
enter the name: george
>>>y=int(input("enter the number"))
enter the number 3
```

#python accepts string as default data type. conversion is required for type.

OUTPUT: Output can be displayed to the user using Print statement .

Syntax:

```
print (expression/constant/variable)
```

Example:

```
>>> print ("Hello")
Hello
```

3.6 COMMENTS:

- ❖ A **hash sign (#)** is the beginning of a comment.
- ❖ Anything written after # in a line is ignored by interpreter.
Eg:percentage = (minute * 100) / 60 **# calculating percentage of an hour**
- ❖ Python **does not have multiple-line commenting feature**. You have to comment each line individually as follows :

Example:

```
# This is a comment.
# This is a comment, too.
# I said that already.
```

3.7 DOCSTRING:

- ❖ Docstring is short for documentation string.
- ❖ It is a string that occurs as the first statement in a module, function, class, or method definition. We must write what a function/class does in the docstring.
- ❖ **Triple quotes** are used while writing docstrings.

Syntax:

```
functionname_doc._
```

Example:

```
def double(num):
    """Function to double the value"""
    return 2*num
>>> print(double.__doc__)
Function to double the value
```

3.8 LINES AND INDENTATION:

- ❖ Most of the programming languages like C, C++, Java use braces { } to define a block of code. But, python uses indentation.
- ❖ Blocks of code are denoted by line indentation.
- ❖ It is a space given to the block of codes for class and function definitions or flow control.

Example:

```
a=3
b=1
if a>b:
    print("a is greater")
else:
    print("b is greater")
```

3.9 QUOTATION IN PYTHON:

Python accepts single ('), double ("") and triple ('' or "'''') quotes to denote string literals.
Anything that is represented using quotations are considered as string.

- ❖ single quotes (' ') Eg, 'This a string in single quotes'
- ❖ double quotes (" ") Eg, "This a string in double quotes"
- ❖ triple quotes("''' ''''") Eg, This is a paragraph. It is made up of multiple lines and sentences."'''"

3.10 TUPLE ASSIGNMENT

- ❖ An assignment to all of the elements in a tuple using a single assignment statement.
- ❖ Python has a very powerful **tuple assignment** feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.
- ❖ The left side is a tuple of variables; the right side is a tuple of values.
- ❖ Each value is assigned to its respective variable.
- ❖ All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.
- ❖ Naturally, the number of variables on the left and the number of values on the right have to be the same.

```
>>> (a, b, c, d) = (1, 2, 3)
```

```
ValueError: need more than 3 values to unpack
```

Example:

-It is useful to swap the values of two variables. With **conventional assignment statements**, we have to use a temporary variable. For example, to swap a and b:

Swap two numbers	Output:
<pre>a=2;b=3 print(a,b) temp = a a = b b = temp print(a,b)</pre>	<pre>(2, 3) (3, 2) >>></pre>

-Tuple assignment solves this problem neatly:

```
(a, b) = (b, a)
```

-One way to think of tuple assignment is as tuple packing/unpacking.

In tuple packing, the values on the left are ‘packed’ together in a tuple:

```
>>> b = ("George", 25, "20000") # tuple packing
```

-In tuple unpacking, **the values in a tuple on the right are ‘unpacked’ into the variables/names on the right:**

```
>>> b = ("George", 25, "20000") # tuple packing
>>> (name, age, salary) = b # tuple unpacking
>>> name
'George'
>>> age
25
>>> salary
'20000'
```

-The right side can be any kind of sequence (string,list,tuple)

Example:

-To split an email address in to user name and a domain

```
>>> mailid='god@abc.org'
>>> name, domain=mailid.split('@')
>>> print name
god
>>> print (domain)
abc.org
```

4. OPERATORS:

- ❖ Operators are the constructs which can manipulate the value of operands.
- ❖ Consider the **expression $4 + 5 = 9$** . Here, **4 and 5 are called operands and + is called operator**
- ❖ Types of Operators:

-Python language supports the following types of operators

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

4.1 Arithmetic operators:

They are used to perform **mathematical operations** like addition, subtraction, multiplication etc. **Assume, a=10 and b=5**

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	$a + b = 30$
- Subtraction	Subtracts right hand operand from left hand operand.	$a - b = -10$
*	Multiplies values on either side of the operator	$a * b = 200$
/ Division	Divides left hand operand by right hand operand	$b / a = 2$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$b \% a = 0$
** Exponent	Performs exponential (power) calculation on operators	$a^{**}b = 10 \text{ to the power } 20$
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed	$5//2=2$

Examples

```

a=10
b=5
print("a+b=",a+b)
print("a-b=",a-b)
print("a*b=",a*b)
print("a/b=",a/b)
print("a%b=",a%b)
print("a//b=",a//b)
print("a**b=",a**b)

```

Output:

```

a+b= 15
a-b= 5
a*b= 50
a/b= 2.0
a%b= 0
a//b= 2
a**b= 100000

```

4.2 Comparison (Relational) Operators:

- Comparison operators are used to compare values.
- It either returns True or False according to the condition. **Assume, a=10 and b=5**

Operator	Description	Example
==	If the values of two operands are equal, then the condition	$(a == b)$ is

	becomes true.	not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!=b) is true
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Example

```
a=10
b=5
print("a>b=>",a>b)
print("a>b=>",a<b)
print("a==b=>",a==b)
print("a!=b=>",a!=b)
print("a>=b=>",a<=b)
print("a>=b=>",a>=b)
```

Output:

```
a>b=> True
a>b=> False
a==b=> False
a!=b=> True
a>=b=> False
a>=b=> True
```

4.3 Assignment Operators:

-Assignment operators are used in Python to assign values to variables.

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a

<code>*= AND</code>	It multiplies right operand with the left operand and assign the result to left operand	<code>c *= a</code> is equivalent to <code>c = c * a</code>
<code>/= AND</code>	It divides left operand with the right operand and assign the result to left operand	<code>c /= a</code> is equivalent to <code>c = c / a</code> <code>/= a</code> is equivalent to <code>c = c / a</code>
<code>%= AND</code>	It takes modulus using two operands and assign the result to left operand	<code>c %= a</code> is equivalent to <code>c = c % a</code>
<code>**= AND</code>	Performs exponential (power) calculation on operators and assign value to the left operand	<code>c **= a</code> is equivalent to <code>c = c ** a</code>
<code>//= Floor Division</code>	It performs floor division on operators and assign value to the left operand	<code>c //= a</code> is equivalent to <code>c = c // a</code>

Example

```

a = 21
b = 10
c = 0
c = a + b
print("Line 1 - Value of c is ", c)
c += a
print("Line 2 - Value of c is ", c)
c *= a
print("Line 3 - Value of c is ", c)
c /= a
print("Line 4 - Value of c is ", c)
c = 2
c %= a
print("Line 5 - Value of c is ", c)
c **= a
print("Line 6 - Value of c is ", c)
c //= a
print("Line 7 - Value of c is ", c)

```

Output

```

Line 1 - Value of c is 31
Line 2 - Value of c is 52
Line 3 - Value of c is 1092
Line 4 - Value of c is 52.0
Line 5 - Value of c is 2
Line 6 - Value of c is 2097152
Line 7 - Value of c is 99864

```

4.4 Logical Operators:

-Logical operators are the and, or, not operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Example

```
a = True  
b = False  
print('a and b is',a and b)  
print('a or b is',a or b)  
print('not a is',not a)
```

Output

```
x and y is False  
x or y is True  
not x is False
```

4.5 Bitwise Operators:

- A **bitwise operation** operates on one or more **bit** patterns at the level of individual bits

Example: Let x = 10 (0000 1010 in binary) and
y = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
	Bitwise OR	x y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x>> 2 = 2 (0000 0010)
<<	Bitwise left shift	x<< 2 = 40 (0010 1000)

Example

```
a = 60      # 60 = 0011 1100  
b = 13      # 13 = 0000 1101  
c = 0  
c = a & b;  # 12 = 0000 1100
```

Output

```
Line 1 - Value of c is 12  
Line 2 - Value of c is 61  
Line 3 - Value of c is 49  
Line 4 - Value of c is -61
```

```
print "Line 1 - Value of c is ", c  
c = a | b;    # 61 = 0011 1101  
print "Line 2 - Value of c is ", c  
c = a ^ b;    # 49 = 0011 0001  
print "Line 3 - Value of c is ", c  
c = ~a;       # -61 = 1100 0011
```

```
Line 5 - Value of c is 240  
Line 6 - Value of c is 15
```



```

print "Line 4 - Value of c is ", c
c = a << 2;      # 240 = 1111 0000
print "Line 5 - Value of c is ", c
c = a >> 2;      # 15 = 0000 1111
print "Line 6 - Value of c is ", c

```

4.6 Membership Operators:

- ❖ Evaluates to find a value or a variable is in the specified sequence of string, list, tuple, dictionary or not.
- ❖ Let, **x=[5,3,6,4,1]**. To check particular item in list or not, **in** and **not in** operators are used.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Example:

```

x=[5,3,6,4,1]
>>> 5 in x
True
>>> 5 not in x
False

```

4.7 Identity Operators:

- ❖ They are used to check if two values (or variables) are located on the same part of the memory.

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

Example

```

x = 5
y = 5
x2 = 'Hello'
y2 = 'Hello'
print(x1 is not y1)
print(x2 is y2)

```

Output

```

False
True

```

5.OPERATOR PRECEDENCE:

When an expression contains **more than one operator**, the order of evaluation depends on the order of operations.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^	Bitwise exclusive 'OR' and regular 'OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //=-+= *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

-For mathematical operators, Python follows mathematical convention.

-The acronym **PEMDAS** (Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction) is a useful way to remember the rules:

- ❖ Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, $2 * (3-1)$ is 4, and $(1+1)**(5-2)$ is 8.
- ❖ You can also use parentheses to make an expression easier to read, as in $(\text{minute} * 100) / 60$, even if it doesn't change the result.
- ❖ Exponentiation has the next highest precedence, so $1 + 2**3$ is 9, not 27, and $2 * 3**2$ is 18, not 36.
- ❖ Multiplication and Division have higher precedence than Addition and Subtraction. So $2*3-1$ is 5, not 4, and $6+4/2$ is 8, not 5.
- ❖ Operators with the same precedence are evaluated from left to right (except exponentiation).

Example:

a=9-12/3+3*2-1 a=? a=9-4+3*2-1 a=9-4+6-1 a=5+6-1 a=11-1 a=10	A=2*3+4%5-3/2+6 A=6+4%5-3/2+6 A=6+4-3/2+6 A=6+4-1+6 A=10-1+6 A=9+6 A=15	find m=? m=-43 8&&0 -2 m=-43 0 -2 m=1 -2 m=1
a=2,b=12,c=1 d=a a c d=2<12>1 d=1>1 d=0	a=2,b=12,c=1 d=a a c-1 d=2<12>1-1 d=2<12>0 d=1>0 d=1	a=2*3+4%5-3//2+6 a=6+4-1+6 a=10-1+6 a=15

6.Functions, Function Definition And Use, Function call, Flow Of Execution, Function Prototypes, Parameters And Arguments, Return statement, Argumentstypes,Modules

6.1 FUNCTIONS:

- **Function is a sub program which consists of set of instructions used to perform a specific task. A large program is divided into basic building blocks called function.**

Need For Function:

- ❖ When the program is too complex and large they are divided into parts. Each part is separately coded and combined into single program. Each subprogram is called as function.
- ❖ Debugging, Testing and maintenance becomes easy when the program is divided into subprograms.
- ❖ Functions are used to avoid rewriting same code again and again in a program.
- ❖ Function provides code re-usability
- ❖ The length of the program is reduced.

Types of function:

Functions can be classified into two categories:

- i) user defined function
- ii) Built in function

i) Built in functions

- ❖ Built in functions are the functions that are already created and stored in python.
- ❖ These built in functions are always available for usage and accessed by a programmer. It cannot be modified.

Built in function	Description
-------------------	-------------

>>> max(3,4)	# returns largest element
4	
>>> min(3,4)	# returns smallest element
3	
>>> len("hello")	#returns length of an object
5	
>>> range(2,8,1)	#returns range of given values
[2, 3, 4, 5, 6, 7]	
>>> round(7.8)	#returns rounded integer of the given number
8.0	
>>> chr(5)	#returns a character (a string) from an integer
\x05'	
>>> float(5)	#returns float number from string or integer
5.0	
>>> int(5.0)	# returns integer from string or float
5	
>>> pow(3,5)	#returns power of given number
243	
>>> type(5.6)	#returns data type of object to which it belongs
<type 'float'>	
>>> t=tuple([4,6.0,7])	# to create tuple of items from list
(4, 6.0, 7)	
>>> print("good morning")	# displays the given object
Good morning	
>>> input("enter name: ")	# reads and returns the given string
enter name : George	

ii) User Defined Functions:

- ❖ User defined functions are the functions that programmers create for their requirement and use.
- ❖ These functions can then be combined to form module which can be used in other programs by importing them.
- ❖ Advantages of user defined functions:
 - Programmers working on large project can divide the workload by making different functions.
 - If repeated code occurs in a program, function can be used to include those codes and execute when needed by calling that function.

6.2 Function definition: (Sub program)

- ❖ def keyword is used to define a function.
- ❖ Give the function name after def keyword followed by parentheses in which arguments are given.
- ❖ End with colon (:)
- ❖ Inside the function add the program statements to be executed
- ❖ End with or without return statement

Syntax:

```
def fun_name(Parameter1,Parameter2...Parameter n):  
    statement1  
    statement2...  
    statement n  
    return[expression]
```

Example:

```
def my_add(a,b):  
    c=a+b  
    return c
```

6.3 Function Calling: (Main Function)

- Once we have defined a function, we can call it from another function, program or even the Python prompt.
- To call a function we **simply type the function name with appropriate arguments.**

Example:

```
x=5  
y=4  
my_add(x,y)
```

6.4 Flow of Execution:

- ❖ The order in which statements are executed is called the **flow of execution**
- ❖ Execution always begins at the first statement of the program.
- ❖ Statements are executed one at a time, in order, from top to bottom.
- ❖ Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- ❖ Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the **def** statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called.

6.5 Function Prototypes:

- i. Function without arguments and without return type
- ii. Function with arguments and without return type
- iii. Function without arguments and with return type
- iv. Function with arguments and with return type

i) Function without arguments and without return type

- In this type no argument is passed through the function call and no output is return to main function
- The sub function will read the input values perform the operation and print the result in the same block

ii) Function with arguments and without return type

- Arguments are passed through the function call but output is not return to the main function

iii) Function without arguments and with return type

- In this type no argument is passed through the function call but output is return to the main function.

iv) Function with arguments and with return type

- In this type arguments are passed through the function call and output is return to the main function

Without Return Type

Without argument	With argument
<pre>def add(): a=int(input("enter a")) b=int(input("enter b")) c=a+b print(c) add()</pre>	<pre>def add(a,b): c=a+b print(c) a=int(input("enter a")) b=int(input("enter b")) add(a,b)</pre>
OUTPUT: enter a 5 enter b 10 15	OUTPUT: enter a 5 enter b 10 15

With return type

Without argument	With argument
<pre>def add(): a=int(input("enter a")) b=int(input("enter b")) c=a+b return c c=add() print(c)</pre>	<pre>def add(a,b): c=a+b return c a=int(input("enter a")) b=int(input("enter b")) c=add(a,b) print(c)</pre>
OUTPUT: enter a 5 enter b 10 15	OUTPUT: enter a 5 enter b 10 15

6.6 Parameters And Arguments:

Parameters:

- Parameters are the value(s) provided in the parenthesis when we write function header.
- These are the values required by function to work.
- If there is more than one value required, all of them will be listed in parameter list separated by **comma**.
- Example: def my_add(a,b):

Arguments :

- Arguments are the value(s) provided in function call/invoke statement.
- List of arguments should be supplied in same way as parameters are listed.
- Bounding of parameters to arguments is done 1:1, and so there should be same number and type of arguments as mentioned in parameter list.
- Example: my_add(x,y)

6.7 RETURN STATEMENT:

- The **return statement is used to exit a function** and go back to the place from where it was called.
- If the return statement has no arguments, then it will not return any values. But exits from function.

Syntax:

```
return[expression]
```

Example:

```
def my_add(a,b):
    c=a+b
    return c
x=5
y=4
print(my_add(x,y))
```

Output:

```
9
```

6.8 ARGUMENTS TYPES:

1. Required Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable length Arguments

- ❖ **Required Arguments:** The number of arguments in the function call should match exactly with the function definition.

```
def my_details( name, age ):
    print("Name: ", name)
    print("Age ", age)
    return
my_details("george",56)
```

Output:

```
Name: george  
Age 56
```

❖ Keyword Arguments:

Python interpreter is able to use the keywords provided to match the values with parameters even though if they are arranged in out of order.

```
def my_details( name, age ):  
    print("Name: ", name)  
    print("Age ", age)  
    return  
my_details(age=56,name="george")
```

Output:

```
Name: george  
Age 56
```

❖ Default Arguments:

Assumes a default value if a value is not provided in the function call for that argument.

```
def my_details( name, age=40 ):  
    print("Name: ", name)  
    print("Age ", age)  
    return  
my_details(name="george")
```

Output:

```
Name: george  
Age 40
```

❖ Variable length Arguments

If we want to specify more arguments than specified while defining the function, variable length arguments are used. It is denoted by * symbol before parameter.

```
def my_details(*name ):  
    print(*name)  
my_details("rajan","rahul","micheal",  
ärjun")
```

Output:

```
rajan rahul micheal ärjun
```

6.9 MODULES:

- A module is a file containing Python definitions ,functions, statements and instructions.
- Standard library of Python is extended as modules.
- To use these modules in a program, programmer needs to import the module.

- Once we import a module, we can reference or use to any of its functions or variables in our code.
 - There is large number of standard modules also available in python.
 - Standard modules can be imported the same way as we import our user-defined modules.
 - Every module contains many function.
 - To access one of the function , you have to specify the name of the module and the name of the function separated by dot. This format is called dot notation.

Syntax:

```
import module_name
module_name.function_name(variable)
```

Importing Builtin Module:	Importing User Defined Module:
<pre>import math x=math.sqrt(25) print(x)</pre>	<pre>import cal x=cal.add(5,4) print(x)</pre>

There are **four ways to import a module** in our program, they are

Import: It is simplest and most common way to use modules in our code. Example: <pre>import math x=math.pi print("The value of pi is", x)</pre> <p>Output: The value of pi is 3.141592653589793</p>	from import : It is used to get a specific function in the code instead of complete file. Example: <pre>from math import pi x=pi print("The value of pi is", x)</pre> <p>Output: The value of pi is 3.141592653589793</p>
import with renaming: We can import a module by renaming the module as our wish. Example: <pre>import math as m x=m.pi print("The value of pi is", x)</pre> <p>Output: The value of pi is 3.141592653589793</p>	import all: We can import all names(definitions) form a module using * Example: <pre>from math import * x=pi print("The value of pi is", x)</pre> <p>Output: The value of pi is 3.141592653589793</p>

Built-in python modules are,

1.math – mathematical functions:
some of the functions in math module is,

- math.ceil(x) - Return the ceiling of x , the smallest integer greater

- than or equal to x
- + math.floor(x) - Return the floor of x, the largest integer less than or equal to x.
 - + math.factorial(x) -Return x factorial. math.gcd(x,y)- Return the greatest common divisor of the integers a and b
 - + math.sqrt(x)- Return the square root of x
 - + math.log(x)- return the natural logarithm of x
 - + math.log10(x) – returns the base-10 logarithms
 - + math.log2(x) - Return the base-2 logarithm of x.
 - + math.sin(x) – returns sin of x radians
 - + math.cos(x)- returns cosine of x radians
 - + math.tan(x)-returns tangent of x radians
 - + math.pi - The mathematical constant $\pi = 3.141592$
 - + math.e – returns The mathematical constant $e = 2.718281$

2 .random-Generate pseudo-random numbers

- + random.randrange(stop)
- + random.randrange(start, stop[, step])
- + random.uniform(a, b)
- Return a random floating point number

ILLUSTRATIVE PROGRAMS

<u>Program for SWAPPING(Exchanging)of values</u>	<u>Output</u>
<pre>a = int(input("Enter a value ")) b = int(input("Enter b value ")) c = a a = b b = c print("a=",a,"b=",b,)</pre>	Enter a value 5 Enter b value 8 a=8 b=5
<u>Program to find distance between two points</u>	<u>Output</u>
<pre>import math x1=int(input("enter x1")) y1=int(input("enter y1")) x2=int(input("enter x2")) y2=int(input("enter y2")) distance =math.sqrt((x2-x1)**2)+((y2-y1)**2) print(distance)</pre>	enter x1 7 enter y1 6 enter x2 5 enter y2 7 2.5

<u>Program to circulate n numbers</u>	<u>Output:</u>
a=list(input("enter the list"))	enter the list '1234'

<pre>print(a) for i in range(1,len(a),1): print(a[i:]+a[:i])</pre>	['1', '2', '3', '4'] ['2', '3', '4', '1'] ['3', '4', '1', '2'] ['4', '1', '2', '3']
--	--

BOOLEAN VALUES:

Boolean:

- ❖ Boolean data type have two values. They are 0 and 1.
- ❖ 0 represents False
- ❖ 1 represents True
- ❖ True and False are keyword.

Example:

```
>>> 3==5
False
>>> 6==6
True
>>> True+True
2
>>> False+True
1
>>> False*True
0
```

OPERATORS:

- ❖ Operators are the constructs which can manipulate the value of operands.
- ❖ Consider the expression $4 + 5 = 9$. Here, 4 and 5 are called operands and + is called operator.

Types of Operators:

1. Arithmetic Operators
2. Comparison (Relational) Operators
3. Assignment Operators
4. Logical Operators
5. Bitwise Operators
6. Membership Operators
7. Identity Operators

Arithmetic operators:

They are used to perform mathematical operations like addition, subtraction, multiplication etc.

Operator	Description	Example a=10,b=20
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a - b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b = 10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed	5//2=2

Comparison (Relational) Operators:

- ❖ Comparison operators are used to compare values.
- ❖ It either returns True or False according to the condition.

Operator	Description	Example a=10,b=20
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!=b) is true
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Assignment Operators:

Assignment operators are used in Python to assign values to variables.

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a

<code>/= Divide AND</code>	It divides left operand with the right operand and assign the result to left operand	$c /= a$ is equivalent to $c = c / a$ $c /= a$ is equivalent to $c = c / a$
<code>%= Modulus AND</code>	It takes modulus using two operands and assign the result to left operand	$c %= a$ is equivalent to $c = c \% a$
<code>**= Exponent AND</code>	Performs exponential (power) calculation on operators and assign value to the left operand	$c **= a$ is equivalent to $c = c ** a$
<code>//= Floor Division</code>	It performs floor division on operators and assign value to the left operand	$c //= a$ is equivalent to $c = c // a$

Logical Operators:

Logical operators are and, or, not operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Bitwise Operators:

Let $x = 10$ (0000 1010 in binary) and $y = 4$ (0000 0100 in binary)

Operator	Meaning	Example
<code>&</code>	Bitwise AND	$x \& y = 0$ (0000 0000)
<code> </code>	Bitwise OR	$x y = 14$ (0000 1110)
<code>-</code>	Bitwise NOT	$\sim x = -11$ (1111 0101)
<code>^</code>	Bitwise XOR	$x ^ y = 14$ (0000 1110)
<code>>></code>	Bitwise right shift	$x >> 2 = 2$ (0000 0010)
<code><<</code>	Bitwise left shift	$x << 2 = 40$ (0010 1000)

Membership Operators:

- ❖ Evaluates to find a value or a variable is in the specified sequence of string, list, tuple, dictionary or not.
- ❖ To check particular element is available in the list or not.
- ❖ Operators are in and not in.

Operator	Meaning	Example
<code>in</code>	True if value/variable is found in the sequence	5 in x
<code>not in</code>	True if value/variable is not found in the sequence	5 not in x

Example:

```
x=[5,3,6,4,1]
```

```
>>> 5 in x
```

```
True
```

```
>>> 5 not in x
```

```
False
```

Identity Operators:

They are used to check if two values (or variables) are located on the same part of the memory.

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

Example

```
x = 5
```

```
y = 5
```

```
a = 'Hello'
```

```
b = 'Hello'
```

```
print(x is not y) // False
```

```
print(a is b)//True
```

CONDITIONALS

- ❖ Conditional if
- ❖ Alternative if... else
- ❖ Chained if...elif...else
- ❖ Nested if....else

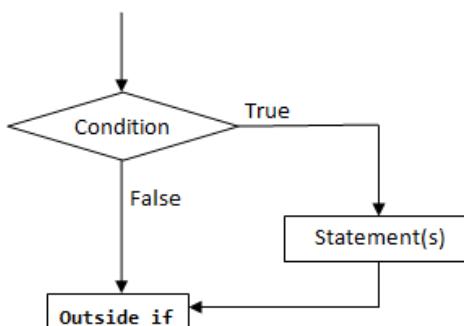
Conditional (if):

conditional (if) is used to test a condition, if the condition is true the statements inside if will be executed.

syntax:

```
if(condition 1):
    Statement 1
```

Flowchart:



Example:

1. Program to provide flat rs 500, if the purchase amount is greater than 2000.
2. Program to provide bonus mark if the category is sports.

Program to provide flat rs 500, if the purchase amount is greater than 2000.	output
<pre>purchase=eval(input("enter your purchase amount")) if(purchase>=2000): purchase=purchase-500 print("amount to pay",purchase)</pre>	enter your purchase amount 2500 amount to pay 2000
Program to provide bonus mark if the category is sports	output
<pre>m=eval(input("enter ur mark out of 100")) c=input("enter ur category G/S") if(c=="S"): m=m+5 print("mark is",m)</pre>	enter ur mark out of 100 85 enter ur category G/S S mark is 90

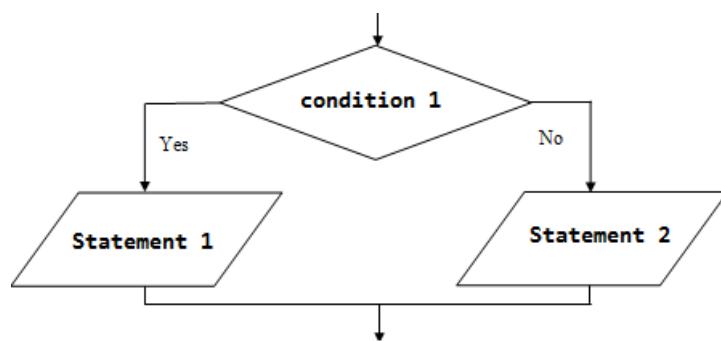
alternative (if-else)

In the alternative the condition must be true or false. In this **else** statement can be combined with **if** statement. The **else** statement contains the block of code that executes when the condition is false. If the condition is true statements inside the if get executed otherwise else part gets executed. The alternatives are called branches, because they are branches in the flow of execution.

syntax:

```
if(condition 1):
    Statement 1
else:
    Statement 2
```

Flowchart:



Examples:

1. odd or even number
2. positive or negative number
3. leap year or not

4. greatest of two numbers

5. eligibility for voting

Odd or even number	Output
n=eval(input("enter a number")) if(n%2==0): print("even number") else: print("odd number")	enter a number4 even number
positive or negative number	Output
n=eval(input("enter a number")) if(n>=0): print("positive number") else: print("negative number")	enter a number8 positive number
leap year or not	Output
y=eval(input("enter a yaer")) if(y%4==0): print("leap year") else: print("not leap year")	enter a yaer2000 leap year
greatest of two numbers	Output
a=eval(input("enter a value:")) b=eval(input("enter b value:")) if(a>b): print("greatest:",a) else: print("greatest:",b)	enter a value:4 enter b value:7 greatest: 7
eligibility for voting	Output
age=eval(input("enter ur age:")) if(age>=18): print("you are eligible for vote") else: print("you are eligible for vote")	enter ur age:78 you are eligible for vote

Chained conditionals(if-elif-else)

- The elif is short for else if.
- This is used to check more than one condition.
- If the condition1 is False, it checks the condition2 of the elif block. If all the conditions are False, then the else part is executed.
- Among the several if...elif...else part, only one part is executed according to the condition.

- The if block can have only one else block. But it can have multiple elif blocks.
- The way to express a computation like that is a chained conditional.

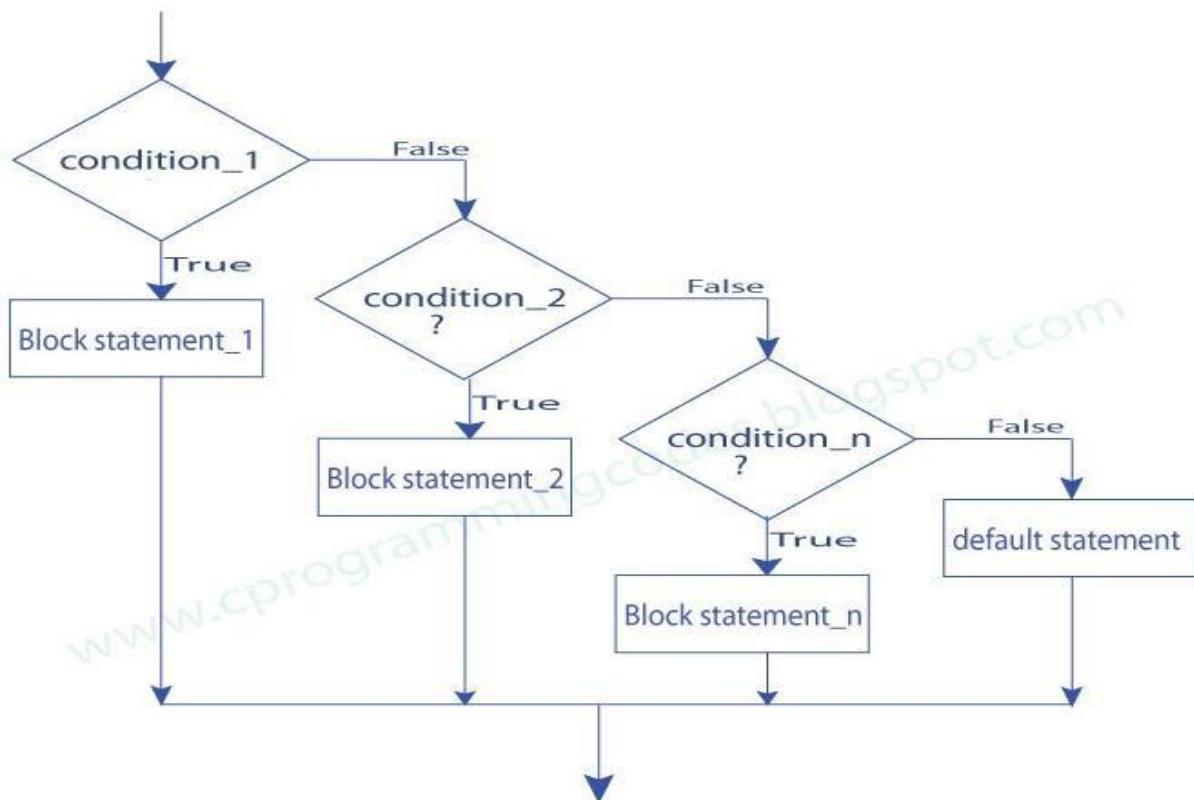
syntax:

```

if(condition 1):
    statement 1
elif(condition 2):
    statement 2
elif(condition 3):
    statement 3
else:
    default statement

```

Flowchart:



Example:

1. student mark system
2. traffic light system
3. compare two numbers
4. roots of quadratic equation

student mark system	Output
<pre>mark=eval(input("enter ur mark:")) if(mark>=90): print("grade:S") elif(mark>=80): print("grade:A") elif(mark>=70): print("grade:B") elif(mark>=50): print("grade:C") else: print("fail")</pre>	enter ur mark:78 grade:B
traffic light system	Output
<pre>colour=input("enter colour of light:") if(colour=="green"): print("GO") elif(colour=="yellow"): print("GET READY") else: print("STOP")</pre>	enter colour of light:green GO
compare two numbers	Output
<pre>x=eval(input("enter x value:")) y=eval(input("enter y value:")) if(x == y): print("x and y are equal") elif(x < y): print("x is less than y") else: print("x is greater than y")</pre>	enter x value:5 enter y value:7 x is less than y
Roots of quadratic equation	output
<pre>a=eval(input("enter a value:")) b=eval(input("enter b value:")) c=eval(input("enter c value:")) d=(b*b-4*a*c) if(d==0): print("same and real roots") elif(d>0): print("diffrent real roots") else: print("imaginagry roots")</pre>	enter a value:1 enter b value:0 enter c value:0 same and real roots

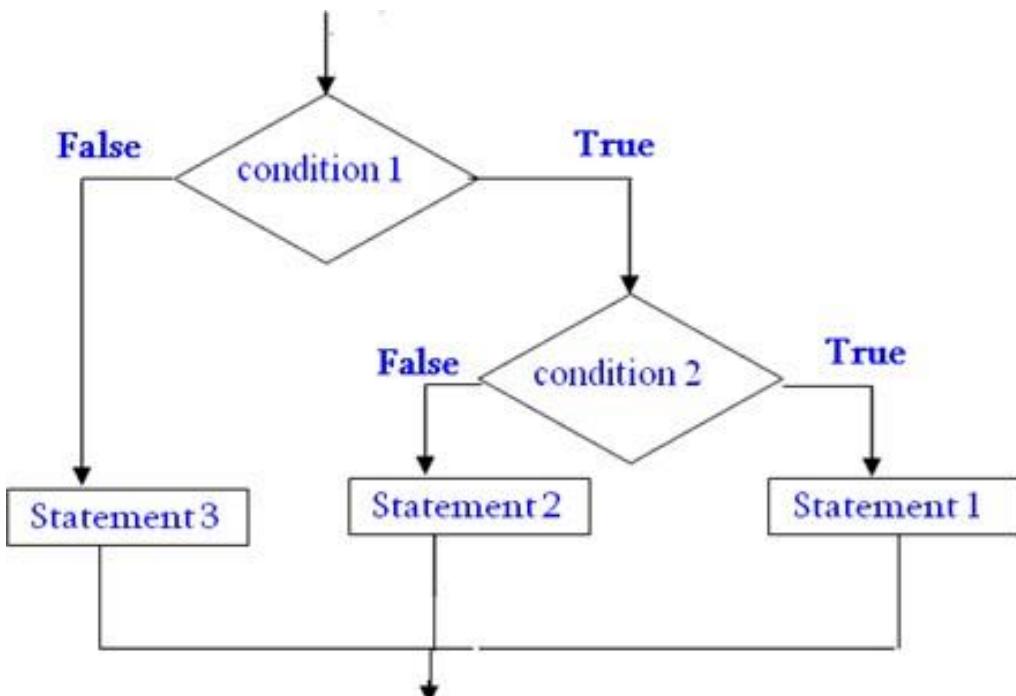
Nested conditionals

One conditional can also be nested within another. Any number of condition can be nested inside one another. In this, if the condition is true it checks another if condition1. If both the conditions are true statement1 get executed otherwise statement2 get execute. if the condition is false statement3 gets executed

Syntax:

```
if (condition):
    if(condition 1):
        statement 1
    else:
        statement 2
else:
    statement 3
```

Flowchart:



Example:

1. greatest of three numbers
2. positive negative or zero

greatest of three numbers	output
a=eval(input("enter the value of a")) b=eval(input("enter the value of b")) c=eval(input("enter the value of c")) if(a>b): if(a>c): print("the greatest no is",a) else: print("the greatest no is",c)	enter the value of a 9 enter the value of b 1 enter the value of c 8 the greatest no is 9

```

else:
    if(b>c):
        print("the greatest no is",b)
    else:
        print("the greatest no is",c)

```

positive negative or zero

```

n=eval(input("enter the value of n:"))
if(n==0):
    print("the number is zero")
else:
    if(n>0):
        print("the number is positive")
    else:
        print("the number is negative")

```

output

enter the value of n:-9
the number is negative

ITERATION/CONTROL STATEMENTS:

- ❖ state
- ❖ while
- ❖ for
- ❖ break
- ❖ continue
- ❖ pass

State:

Transition from one process to another process under specified condition with in a time is called state.

While loop:

- While loop statement in Python is used to repeatedly executes set of statement as long as a given condition is true.
- In while loop, test expression is checked first. The body of the loop is entered only if the test_expression is True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.
- In Python, the body of the while loop is determined through indentation.
- The statements inside the while starts with indentation and the first unindented line marks the end.

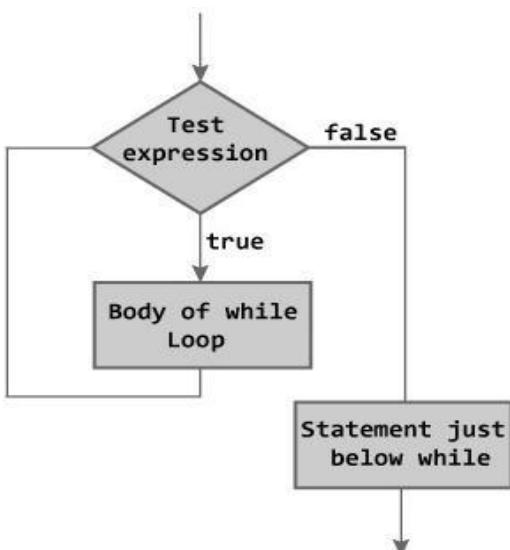
Syntax:

```

initial value
while(condition):
    body of while loop
    increment

```

Flowchart:



Examples:

1. program to find sum of n numbers:
2. program to find factorial of a number
3. program to find sum of digits of a number:
4. Program to Reverse the given number:
5. Program to find number is Armstrong number or not
6. Program to check the number is palindrome or not

Sum of n numbers:	output
<pre>n=eval(input("enter n")) i=1 sum=0 while(i<=n): sum=sum+i i=i+1 print(sum)</pre>	enter n 10 55
Factorial of a numbers:	output
<pre>n=eval(input("enter n")) i=1 fact=1 while(i<=n): fact=fact*i i=i+1 print(fact)</pre>	enter n 5 120
Sum of digits of a number:	output
<pre>n=eval(input("enter a number")) sum=0 while(n>0): a=n%10</pre>	enter a number 123 6

```

sum=sum+a
n=n//10
print(sum)

```

Reverse the given number:

```

n=eval(input("enter a number"))
sum=0
while(n>0):
    a=n%10
    sum=sum*10+a
    n=n//10
print(sum)

```

output

enter a number
123
321

Armstrong number or not

```

n=eval(input("enter a number"))
org=n
sum=0
while(n>0):
    a=n%10
    sum=sum+a*a*a
    n=n//10
if(sum==org):
    print("The given number is Armstrong
number")
else:
    print("The given number is not
Armstrong number")

```

output

enter a number153
The given number is Armstrong number

Palindrome or not

```

n=eval(input("enter a number"))
org=n
sum=0
while(n>0):
    a=n%10
    sum=sum*10+a
    n=n//10
if(sum==org):
    print("The given no is palindrome")
else:
    print("The given no is not palindrome")

```

output

enter a number121
The given no is palindrome

For loop:

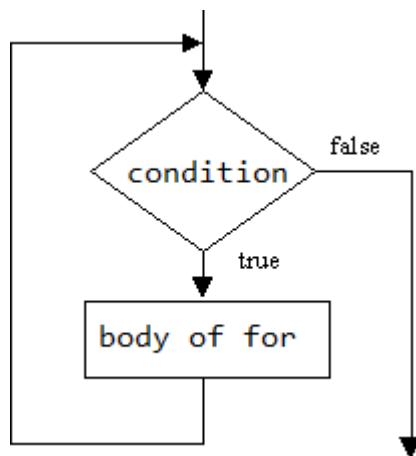
❖ for in range:

- ❖ We can generate a sequence of numbers using range() function.
range(10) will generate numbers from 0 to 9 (10 numbers).
- ❖ In range function have to define the start, stop and step size
as range(start,stop,step size). step size defaults to 1 if not provided.

syntax

```
for i in range(start,stop,steps):
    body of for loop
```

Flowchart:



For in sequence

- ❖ The for loop in Python is used to iterate over a sequence (list, tuple, string). Iterating over a sequence is called traversal. Loop continues until we reach the last element in the sequence.
- ❖ The body of for loop is separated from the rest of the code using indentation.

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

Sequence can be a list, strings or tuples

s.no	sequences	example	output
1.	For loop in string	for i in "Ramu": print(i)	R A M U

2.	For loop in list	for i in [2,3,5,6,9]: print(i)	2 3 5 6 9
3.	For loop in tuple	for i in (2,3,1): print(i)	2 3 1

Examples:

1. print nos divisible by 5 not by 10:
2. Program to print fibonacci series.
3. Program to find factors of a given number
4. check the given number is perfect number or not
5. check the no is prime or not
6. Print first n prime numbers
7. Program to print prime numbers in range

print nos divisible by 5 not by 10	output
<pre>n=eval(input("enter a")) for i in range(1,n,1): if(i%5==0 and i%10!=0): print(i)</pre>	enter a:30 5 15 25
Fibonacci series	output
<pre>a=0 b=1 n=eval(input("Enter the number of terms: ")) print("Fibonacci Series: ") print(a,b) for i in range(1,n,1): c=a+b print(c) a=b b=c</pre>	Enter the number of terms: 6 Fibonacci Series: 0 1 1 2 3 5 8

find factors of a number	Output
<pre>n=eval(input("enter a number:")) for i in range(1,n+1,1): if(n%i==0): print(i)</pre>	enter a number:10 1 2 5 10

check the no is prime or not	output
<pre>n=eval(input("enter a number")) for i in range(2,n): if(n%i==0): print("The num is not a prime") break else: print("The num is a prime number.")</pre>	enter a no:7 The num is a prime number.
check a number is perfect number or not	Output
<pre>n=eval(input("enter a number:")) sum=0 for i in range(1,n,1): if(n%i==0): sum=sum+i if(sum==n): print("the number is perfect number") else: print("the number is not perfect number")</pre>	enter a number:6 the number is perfect number
Program to print first n prime numbers	Output
<pre>number=int(input("enter no of prime numbers to be displayed:")) count=1 n=2 while(count<=number): for i in range(2,n): if(n%i==0): break else: print(n) count=count+1 n=n+1</pre>	enter no of prime numbers to be displayed:5 2 3 5 7 11
Program to print prime numbers in range	output:
<pre>lower=eval(input("enter a lower range")) upper=eval(input("enter a upper range")) for n in range(lower,upper + 1): if n > 1: for i in range(2,n): if (n % i) == 0: break else: print(n)</pre>	enter a lower range50 enter a upper range100 53 59 61 67 71 73 79 83 89 97

Loop Control Structures

BREAK

- ❖ Break statements can alter the flow of a loop.
- ❖ It terminates the current
- ❖ loop and executes the remaining statement outside the loop.
- ❖ If the loop has else statement, that will also gets terminated and come out of the loop completely.

Syntax:

break

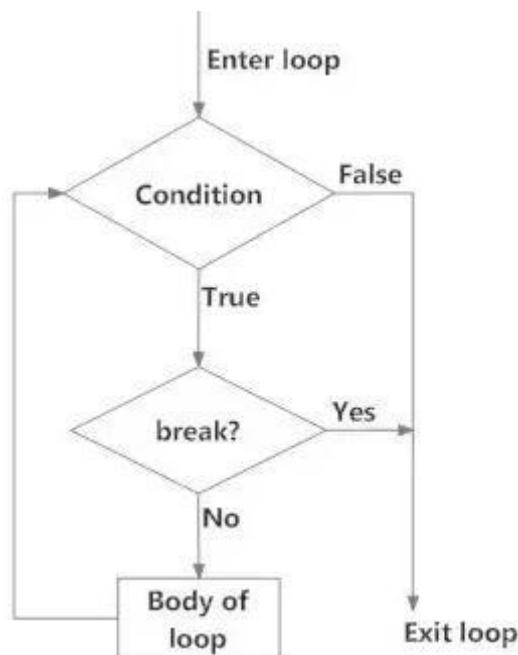
```
while (test Expression):
    // codes
    if (condition for break):
```

break

 // codes



Flowchart



example

```
for i in "welcome":
    if(i=="c"):
        break
    print(i)
```

Output

w
e
l

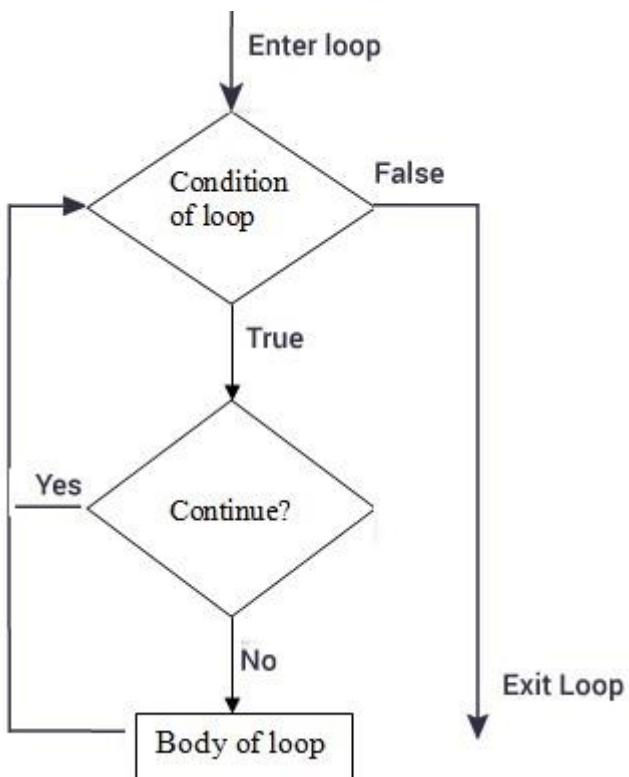
CONTINUE

It terminates the current iteration and transfer the control to the next iteration in the loop.

Syntax: Continue

```
→ while (test Expression):
    // codes
    if (condition for continue):
        continue
    // codes
```

Flowchart



Example:

```
for i in "welcome":
    if(i=="c"):
        continue
    print(i)
```

Output

```
w
e
l
o
m
e
```

PASS

- ❖ It is used when a statement is required syntactically but you don't want any code to execute.
- ❖ It is a null statement, nothing happens when it is executed.

Syntax:

```
pass
break
```

Example	Output
<pre>for i in "welcome": if (i == "c"): pass print(i)</pre>	w e l c o m e

Difference between break and continue

<u>break</u>	<u>continue</u>
It terminates the current loop and executes the remaining statement outside the loop.	It terminates the current iteration and transfer the control to the next iteration in the loop.
syntax: break	syntax: continue
<pre>for i in "welcome": if(i=="c"): break print(i)</pre>	<pre>for i in "welcome": if(i=="c"): continue print(i)</pre>
w e l	w e l o m e

else statement in loops:

else in for loop:

- ❖ If else statement is used in for loop, the else statement is executed when the loop has reached the limit.
- ❖ The statements inside for loop and statements inside else will also execute.

example	output
<pre>for i in range(1,6): print(i) else: print("the number greater than 6")</pre>	1 2 3 4 5 the number greater than 6

else in while loop:

- ❖ If else statement is used within while loop , the else part will be executed when the condition become false.
- ❖ The statements inside for loop and statements inside else will also execute.

Program	output
<pre>i=1 while(i<=5): print(i) i=i+1 else: print("the number greater than 5")</pre>	1 2 3 4 5 the number greater than 5

Fruitful Function

- ❖ Fruitful function
 - ❖ Void function
 - ❖ Return values
 - ❖ Parameters
 - ❖ Local and global scope
 - ❖ Function composition
 - ❖ Recursion

Fruitful function:

A function that returns a value is called fruitful function.

Example:

Root=sqrt(25)

Example:

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

Void Function

A function that perform action but don't return any value.

Example:

print("Hello")

Example:

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

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

Return values:

return keywords are used to return the values from the function.

example:

return a – return 1 variable
return a,b- return 2 variables
return a,b,c- return 3 variables
return a+b- return expression
return 8- return value

PARAMETERS / ARGUMENTS:

- ❖ Parameters are the variables which used in the function definition. Parameters are inputs to functions. Parameter receives the input from the function call.
- ❖ It is possible to define more than one parameter in the function definition.

Types of parameters/Arguments:

1. Required/Positional parameters
2. Keyword parameters
3. Default parameters
4. Variable length parameters

Required/ Positional Parameter:

The number of parameter in the function definition should match exactly with number of arguments in the function call.

Example	Output:
def student(name, roll): print(name,roll) student("George",98)	George 98

Keyword parameter:

When we call a function with some values, these values get assigned to the parameter according to their position. When we call functions in keyword parameter, the order of the arguments can be changed.

Example	Output:
def student(name,roll,mark): print(name,roll,mark) student(90,102,"bala")	90 102 bala

Default parameter:

Python allows function parameter to have default values; if the function is called without the argument, the argument gets its default value in function definition.

Example	Output:
def student(name, age=17): print (name, age)	Kumar 17
student("kumar"):	Ajay 17
student("ajay"):	

Variable length parameter

- ❖ Sometimes, we do not know in advance the number of arguments that will be passed into a function.
- ❖ Python allows us to handle this kind of situation through function calls with number of arguments.
- ❖ In the function definition we use an asterisk (*) before the parameter name to denote this is variable length of parameter.

Example	Output:
def student(name,*mark): print(name,mark) student ("bala",102,90)	bala (102 ,90)

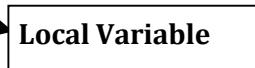
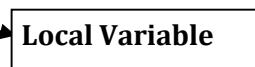
Local and Global Scope

Global Scope

- ❖ The scope of a variable refers to the places that you can see or access a variable.
- ❖ A variable with global scope can be used anywhere in the program.
- ❖ It can be created by defining a variable outside the function.

Example	output
a=50 def add(): b=20 c=a+b print(c) def sub(): b=30 c=a-b print(c) print(a)	70
	20
	50

Local Scope A variable with local scope can be used only within the function .

Example	output
<pre>def add(): b=20 c=a+b print(c)</pre> 	70
<pre>def sub(): b=30 c=a-b print(c) print(a) print(b)</pre> 	20 error error

Function Composition:

- ❖ Function Composition is the ability to call one function from within another function
- ❖ It is a way of combining functions such that the result of each function is passed as the argument of the next function.
- ❖ In other words the output of one function is given as the input of another function is known as function composition.

Example:	Output:
math.sqrt(math.log(10))	
<pre>def add(a,b): c=a+b return c def mul(c,d): e=c*d return e c=add(10,20) e=mul(c,30) print(e)</pre>	900
find sum and average using function composition	output
<pre>def sum(a,b): sum=a+b return sum def avg(sum): avg=sum/2 return avg a=eval(input("enter a:")) b=eval(input("enter b:")) sum=sum(a,b) avg=avg(sum)</pre>	enter a:4 enter b:8 the avg is 6.0

```
print("the avg is",avg)
```

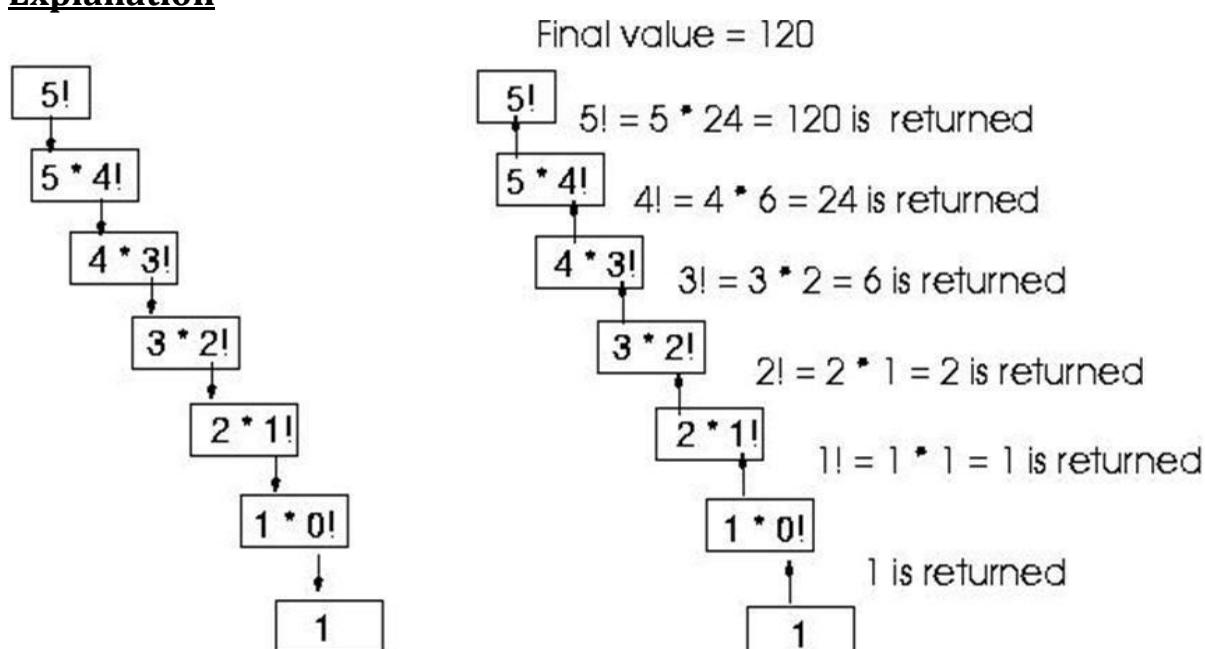
Recursion

A function calling itself till it reaches the base value - stop point of function call.

Example: factorial of a given number using recursion

Factorial of n	Output
<pre>def fact(n): if(n==1): return 1 else: return n*fact(n-1) n=eval(input("enter no. to find fact:")) fact=fact(n) print("Fact is",fact)</pre>	enter no. to find fact:5 Fact is 120

Explanation



Examples:

- sum of n numbers using recursion
- exponential of a number using recursion

Sum of n numbers	Output
<pre>def sum(n): if(n==1): return 1 else: return n*sum(n-1) n=eval(input("enter no. to find sum:")) sum=sum(n) print("Fact is",sum)</pre>	enter no. to find sum:10 Fact is 55

Strings:

- ❖ Strings
- ❖ String slices
- ❖ Immutability
- ❖ String functions and methods
- ❖ String module

Strings:

- ❖ String is defined as sequence of characters represented in quotation marks (either single quotes (') or double quotes (").
 - ❖ An individual character in a string is accessed using a index.
 - ❖ The index should always be an integer (positive or negative).
 - ❖ A index starts from 0 to n-1.
 - ❖ Strings are immutable i.e. the contents of the string cannot be changed after it is created.
 - ❖ Python will get the input at run time by default as a string.
 - ❖ Python does not support character data type. A string of size 1 can be treated as characters.
1. single quotes (' ')
 2. double quotes (" ")
 3. triple quotes("''' "''")

Operations on string:

1. Indexing
2. Slicing
3. Concatenation
4. Repetitions
5. Member ship

String A	H	E	L	L	O
Positive Index	0	1	2	3	4
Negative Index	-5	-4	-3	-2	-1

indexing

```
>>>a="HELLO"
>>>print(a[0])
>>>H
>>>print(a[-1])
>>>O
```

- ❖ Positive indexing helps in accessing the string from the beginning
- ❖ Negative subscript helps in accessing the string from the end.

Slicing:	Print[0:4] – HELL Print[:3] – HEL Print[0:]- HELLO	The Slice[start : stop] operator extracts sub string from the strings. A segment of a string is called a slice.
Concatenation	a="save" b="earth" >>>print(a+b) saveearth	The + operator joins the text on both sides of the operator.
Repetitions:	a="panimalar " >>>print(3*a) panimalarpanimalar panimalar	The * operator repeats the string on the left hand side times the value on right hand side.
Membership:	>>> s="good morning" >>>"m" in s True >>> "a" not in s True	Using membership operators to check a particular character is in string or not. Returns true if present

String slices:

- ❖ A part of a string is called string slices.
- ❖ **The process of extracting a sub string from a string is called slicing.**

Slicing: a="HELLO"	Print[0:4] – HELL Print[:3] – HEL Print[0:]- HELLO	The Slice[n : m] operator extracts sub string from the strings. A segment of a string is called a slice.
----------------------------------	--	---

Immutability:

- ❖ Python strings are “immutable” as they cannot be changed after they are created.
- ❖ Therefore [] operator cannot be used on the left side of an assignment.

operations	Example	output
element assignment	a="PYTHON" a[0]='x'	TypeError: 'str' object does not support element assignment
element deletion	a="PYTHON" del a[0]	TypeError: 'str' object doesn't support element deletion
delete a string	a="PYTHON" del a	NameError: name 'my_string' is not defined

	print(a)	
--	----------	--

string built in functions and methods:

A **method** is a function that “belongs to” an object.

Syntax to access the method

Stringname.method()

a="happy birthday"

here, a is the string name.

	syntax	example	description
1	a.capitalize()	>>> a.capitalize() ' Happy birthday'	capitalize only the first letter in a string
2	a.upper()	>>> a.upper() 'HAPPY BIRTHDAY'	change string to upper case
3	a.lower()	>>> a.lower() ' happy birthday'	change string to lower case
4	a.title()	>>> a.title() ' Happy Birthday '	change string to title case i.e. first characters of all the words are capitalized.
5	a.swapcase()	>>> a.swapcase() 'HAPPY BIRTHDAY'	change lowercase characters to uppercase and vice versa
6	a.split()	>>> a.split() ['happy', 'birthday']	returns a list of words separated by space
7	a.center(width,"fillchar")	>>>a.center(19,"*") '***happy birthday***'	pads the string with the specified “fillchar” till the length is equal to “width”
8	a.count(substring)	>>>a.count('happy') 1	returns the number of occurrences of substring
9	a.replace(old,new)	>>>a.replace('happy', 'wishyou happy') 'wishyou happy birthday'	replace all old substrings with new substrings
10	a.join(b)	>>> b="happy" >>> a="-" >>> a.join(b) 'h-a-p-p-y'	returns a string concatenated with the elements of an iterable. (Here “a” is the iterable)
11	a.isupper()	>>> a.isupper() False	checks whether all the case-based characters (letters) of the string are uppercase.
12	a.islower()	>>> a.islower() True	checks whether all the case-based characters (letters) of the string are lowercase.
13	a.isalpha()	>>> a.isalpha() False	checks whether the string consists of alphabetic characters only.

14	a.isalnum()	>>> a.isalnum() False	checks whether the string consists of alphanumeric characters.
15	a.isdigit()	>>> a.isdigit() False	checks whether the string consists of digits only.
16	a.isspace()	>>> a.isspace() False	checks whether the string consists of whitespace only.
17	a.istitle()	>>> a.istitle() False	checks whether string is title cased.
18	a.startswith(substring)	>>> a.startswith("h") True	checks whether string starts with substring
19	a.endswith(substring)	>>> a.endswith("y") True	checks whether the string ends with the substring
20	a.find(substring)	>>> a.find("happy") 0	returns index of substring, if it is found. Otherwise -1 is returned.
21	len(a)	>>> len(a) >>> 14	Return the length of the string
22	min(a)	>>> min(a) >>> ''	Return the minimum character in the string
23	max(a)	>>> max(a) >>> 'y'	Return the maximum character in the string

String modules:

- ❖ A module is a file containing Python definitions, functions, statements.
- ❖ Standard library of Python is extended as modules.
- ❖ To use these modules in a program, programmer needs to import the module.
- ❖ Once we import a module, we can reference or use to any of its functions or variables in our code.
- ❖ There is large number of standard modules also available in python.
- ❖ Standard modules can be imported the same way as we import our user-defined modules.

Syntax:

```
import module_name
```

Example	output
<pre>import string print(string.punctuation) print(string.digits) print(string.printable) print(string.capwords("happy birthday")) print(string.hexdigits) print(string.octdigits)</pre>	<pre>!"#\$%&'()*+,-./;:<=>?@[\\]^_`{ }~ 0123456789 0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ KLMNOPQRSTUVWXYZ!"#\$%&'()*+,-./;:<=>?@[\\]^_`{ }~ Happy Birthday 0123456789abcdefABCDEF 01234567</pre>

Escape sequences in string

Escape Sequence	Description	example
\n	new line	>>> print("hai \nhello") hai hello
\\"	prints Backslash (\)	>>> print("hai\\hello") hai\hello
'	prints Single quote (')	>>> print("") '
"	prints Double quote (")	>>> print("") "
\t	prints tab sapace	>>> print("hai\thello") hai hello
\a	ASCII Bell (BEL)	>>> print("\a")

List as array:

Array:

Array is a collection of similar elements. Elements in the array can be accessed by index. Index starts with 0. Array can be handled in python by module named array.

To create array have to import array module in the program.

Syntax :

import array

Syntax to create array:

Array_name = module_name.function_name('datatype',[elements])

example:

a=array.array('i',[1,2,3,4])

a- array name

array- module name

i- integer datatype

Example

Program to find sum of array elements	Output
<pre>import array sum=0 a=array.array('i',[1,2,3,4]) for i in a: sum=sum+i print(sum)</pre>	10

Convert list into array:

fromlist() function is used to append list to array. Here the list is act like a array.

Syntax:

arrayname.fromlist(list_name)

Example

program to convert list into array	Output
<pre>import array sum=0 l=[6,7,8,9,5] a=array.array('i',[]) a.fromlist(l) for i in a: sum=sum+i print(sum)</pre>	35

Methods in array

a=[2,3,4,5]

	Syntax	example	Description
1	array(data type, value list)	array('i',[2,3,4,5])	This function is used to create an array with data type and value list specified in its arguments.
2	append()	>>>a.append(6) [2,3,4,5,6]	This method is used to add the at the end of the array.
3	insert(index,element)	>>>a.insert(2,10) [2,3,10,5,6]	This method is used to add the value at the position specified in its argument.
4	pop(index)	>>>a.pop(1) [2,10,5,6]	This function removes the element at the position mentioned in its argument, and returns it.
5	index(element)	>>>a.index(2) 0	This function returns the index of value
6	reverse()	>>>a.reverse() [6,5,10,2]	This function reverses the array.
7	count()	a.count()	This is used to count number of

	4	elements in an array
--	---	----------------------

ILLUSTRATIVE PROGRAMS:

Square root using newtons method:	Output:
<pre>def newtonsqrt(n): root=n/2 for i in range(10): root=(root+n/root)/2 print(root) n=eval(input("enter number to find Sqrt: ")) newtonsqrt(n)</pre>	enter number to find Sqrt: 9 3.0
GCD of two numbers	output
<pre>n1=int(input("Enter a number1:")) n2=int(input("Enter a number2:")) for i in range(1,n1+1): if(n1%i==0 and n2%i==0): gcd=i print(gcd)</pre>	Enter a number1:8 Enter a number2:24 8
Exponent of number	Output:
<pre>def power(base,exp): if(exp==1): return(base) else: return(base*power(base,exp-1)) base=int(input("Enter base: ")) exp=int(input("Enter exponential value: ")) result=power(base,exp) print("Result:",result)</pre>	Enter base: 2 Enter exponential value:3 Result: 8
sum of array elements:	output:
<pre>a=[2,3,4,5,6,7,8] sum=0 for i in a: sum=sum+i print("the sum is",sum)</pre>	the sum is 35
Linear search	output
<pre>a=[20,30,40,50,60,70,89] print(a) search=eval(input("enter a element to search:")) for i in range(0,len(a),1): if(search==a[i]): print("element found at",i+1) break else: print("not found")</pre>	[20, 30, 40, 50, 60, 70, 89] enter a element to search:30 element found at 2

Binary search	output
<pre>a=[20, 30, 40, 50, 60, 70, 89] print(a) search=eval(input("enter a element to search:")) start=0 stop=len(a)-1 while(start<=stop): mid=(start+stop)//2 if(search==a[mid]): print("elemrnt found at",mid+1) break elif(search<a[mid]): stop=mid-1 else: start=mid+1 else: print("not found")</pre>	[20, 30, 40, 50, 60, 70, 89] enter a element to search:30 element found at 2

Modules

A group of functions, variables and classes saved to a file, which is nothing but module.

Every Python file (.py) acts as a module.

Eg: ACYTechmath.py

```
1) x=888
2)
3) def add(a,b):
4)     print("The Sum:",a+b)
5)
6) def product(a,b):
7)     print("The Product:",a*b)
```

ACYTechmath module contains one variable and 2 functions.

If we want to use members of module in our program then we should import that module.

import modulename

We can access members by using module name.

modulename.variable
modulename.function()

test.py:

```
1) import ACYTechmath
2) print(prabhamath.x)
3) ACYTechmath.add(10,20)
4) prabhamath.product(10,20)
5)
6) Output
7) 888
8) The Sum: 30
9) The Product: 200
```

Note:

whenever we are using a module in our program, for that module compiled file will be generated and stored in the hard disk permanently.

Renaming a module at the time of import (module aliasing):

Eg:

```
import ACYTechmath as m
```

here ACYTechmath is original module name and m is alias name.

We can access members by using alias name m

test.py:

```
1) import ACYTechmath as m  
2) print(m.x)  
3) m.add(10,20)  
4) m.product(10,20)
```

from ... import:

We can import particular members of module by using from ... import .

The main advantage of this is we can access members directly without using module name.

Eg:

```
from ACYTechmath import  
x,addprint(x)  
add(10,20)  
product(10,20)==> NameError: name 'product' is not defined
```

We can import all members of a module as follows

```
from ACYTechmath import *
```

test.py:

```
1) from ACYTechmath import *  
2) print(x)  
3) add(10,20)  
4) product(10,20)
```

Various possibilties of import:

```
import modulename  
import module1,module2,module3  
import module1 as m  
import module1 as m1,module2 as m2,module3  
from module import member  
from module import member1,member2,memebr3  
from module import memeber1 as x  
from module import *
```

member aliasing:

```
from ACYTechmath import x as y,add as  
sumprint(y)  
sum(10,20)
```

Once we defined as alias name,we should use alias name only and we should not use original name

Eg:

```
from ACYTechmath import x as y  
print(x)==>NameError: name 'x' is not defined
```

Reloading a Module:

By default module will be loaded only once eventhough we are importing multiple multiple times.

Demo Program for module reloading:

```
1) import time  
2) from imp import reload  
3) import module1  
4) time.sleep(30)  
5) reload(module1)  
6) time.sleep(30)  
7) reload(module1)  
8) print("This is test file")
```

Note: In the above program, everytime updated version of module1 will be available to our program

module1.py:

```
print("This is from module1")
```

test.py

```
1) import module1
2) import module1
3) import module1
4) import module1
5) print("This is test module")
6)
7) Output
8) This is from module1
9) This is test module
```

In the above program test module will be loaded only once even though we are importing multiple times.

The problem in this approach is after loading a module if it is updated outside then updated version of module1 is not available to our program.

We can solve this problem by reloading module explicitly based on our requirement.
We can reload by using reload() function of imp module.

```
import imp
imp.reload(module1)
```

test.py:

```
1) import module1
2) import module1
3) from imp import reload
4) reload(module1)
5) reload(module1)
6) reload(module1)
7) print("This is test module")
```

In the above program module1 will be loaded 4 times in that 1 time by default and 3 times explicitly. In this case output is

```
1) This is from module1
2) This is from module1
3) This is from module1
4) This is from module1
5) This is test module
```

The main advantage of explicit module reloading is we can ensure that updated version is always available to our program.

Finding members of module by using dir() function:

Python provides inbuilt function dir() to list out all members of current module or a specified module.

dir() ==>To list out all members of current module

dir(moduleName)==>To list out all members of specified module

Eg 1: test.py

```
1) x=10
2) y=20
3) def f1():
4)     print("Hello")
5) print(dir()) # To print all members of current module
6)
7) Output
8) ['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
   '__package__', '__spec__', 'f1', 'x', 'y']
```

Eg 2: To display members of particular module:

ACYTechmath.py:

```
1) x=888
2)
3) def add(a,b):
4)     print("The Sum:",a+b)
5)
6) def product(a,b):
7)     print("The Product:",a*b)
```

test.py:

```
1) import ACYTechmath
2) print(dir(ACYTechmath))
3)
4) Output
5) ['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
   '__package__', '__spec__', 'add', 'product', 'x']
```

Note: For every module at the time of execution Python interpreter will add some special properties automatically for internal use.

Eg: __builtins__, __cached__,'__doc__', __file__, __loader__, __name__, __package__, __spec__

Based on our requirement we can access these properties also in our program.

Eg: test.py:

```
1) print(__builtins__)
2) print(__cached__)
3) print(__doc__)
4) print(__file__)
5) print(__loader__)
6) print(__name__)
7) print(__package__)
8) print(__spec__)
9)
10) Output
11) <module 'builtins' (built-in)>
12) None
13) None
```

test.py

```
1) <_frozen_importlib_external.SourceFileLoader object at 0x00572170>
2) main_____
3) None
4) None
```

The Special variable __name__:

For every Python program , a special variable __name__ will be added internally. This variable stores information regarding whether the program is executed as an individual program or as a module.

If the program executed as an individual program then the value of this variable is __main____

If the program executed as a module from some other program then the value of this variable is the name of module where it is defined.

Hence by using this __name__ variable we can identify whether the program executed directly or as a module.

Demo program:

module1.py:

```
1) def f1():
2)     if __name__=='__main__':
3)         print("The code executed as a program")
4)     else:
5)         print("The code executed as a module from some other program")
6) f1()
```

test.py:

```
1) import module1
2) module1.f1()
3)
4) D:\Python_classes>py module1.py
5) The code executed as a program
6)
7) D:\Python_classes>py test.py
8) The code executed as a module from some other program
9) The code executed as a module from some other program
```

Working with math module:

Python provides inbuilt module math.

This module defines several functions which can be used for mathematical operations.

The main important functions are

1. sqrt(x)
2. ceil(x)
3. floor(x)
4. fabs(x)
5. log(x)
6. sin(x)
7. tan(x)

....

Eg:

```
1) from math import *
2) print(sqrt(4))
3) print(ceil(10.1))
4) print(floor(10.1))
5) print(fabs(-10.6))
6) print(fabs(10.6))
```

- 7)
- 8) Output
- 9) 2.0
- 10) 11
- 11) 10
- 12) 10.6
- 13) 10.6

Note:

We can find help for any module by using `help()` function

Eg:

```
import math  
help(math)
```

Working with random module:

This module defines several functions to generate random numbers.

We can use these functions while developing games,in cryptography and to generate random numbers on fly for authentication.

1. random() function:

This function always generate some float value between 0 and 1 (not inclusive)

$$0 < x < 1$$

Eg:

- 1) `from random import *`
- 2) `for i in range(10):`
- 3) `print(random())`
- 4)
- 5) Output
- 6) `0.4572685609302056`
- 7) `0.6584325233197768`
- 8) `0.15444034016553587`
- 9) `0.18351427005232201`
- 10) `0.1330257265904884`
- 11) `0.9291139798071045`
- 12) `0.6586741197891783`
- 13) `0.8901649834019002`
- 14) `0.25540891083913053`
- 15) `0.7290504335962871`

2. randint() function:

To generate random integer between two given numbers(inclusive)

Eg:

```
1) from random import *
2) for i in range(10):
3)     print(randint(1,100)) # generate random int value between 1 and 100(inclusive)
4)
5) Output
6) 51
7) 44
8) 39
9) 70
10) 49
11) 74
12) 52
13) 10
14) 40
15) 8
```

3. uniform():

It returns random float values between 2 given numbers(not inclusive)

Eg:

```
1) from random import *
2) for i in range(10):
3)     print(uniform(1,10))
4)
5) Output
6) 9.787695398230332
7) 6.81102218793548
8) 8.068672144377329
9) 8.567976357239834
10) 6.363511674803802
11) 2.176137584071641
12) 4.822867939432386
13) 6.0801725149678445
14) 7.508457735544763
15) 1.9982221862917555
```

random() ==>in between 0 and 1 (not inclusive)

randint(x,y) ==>in between x and y (inclusive)

uniform(x,y) ==> in between x and y (not inclusive)

4. randrange([start],stop,[step])

returns a random number from range

start \leq x < stop

start argument is optional and default value is 0

step argument is optional and default value is 1

randrange(10)-->generates a number from 0 to 9

randrange(1,11)-->generates a number from 1 to 10

randrange(1,11,2)-->generates a number from 1,3,5,7,9

Eg 1:

- 1) `from random import *`
- 2) `for i in range(10):`
- 3) `print(randrange(10))`
- 4)
- 5) Output
- 6) 9
- 7) 4
- 8) 0
- 9) 2
- 10) 9
- 11) 4
- 12) 8
- 13) 9
- 14) 5
- 15) 9

Eg 2:

- 1) `from random import *`
- 2) `for i in range(10):`
- 3) `print(randrange(1,11))`
- 4)
- 5) Output
- 6) 2
- 7) 2
- 8) 8
- 9) 10
- 10) 3
- 11) 5
- 12) 9
- 13) 1
- 14) 6
- 15) 3

Eg 3:

```
1) from random import *
2) for i in range(10):
3)     print(randrange(1,11,2))
4)
5) Output
6) 1
7) 3
8) 9
9) 5
10) 7
11) 1
12) 1
13) 1
14) 7
15) 3
```

5. choice() function:

It wont return random number.

It will return a random object from the given list or tuple.

Eg:

```
1) from random import *
2) list=["Sunny","Bunny","Chinny","Vinny","pinny"]
3) for i in range(10):
4)     print(choice(list))
```

Output

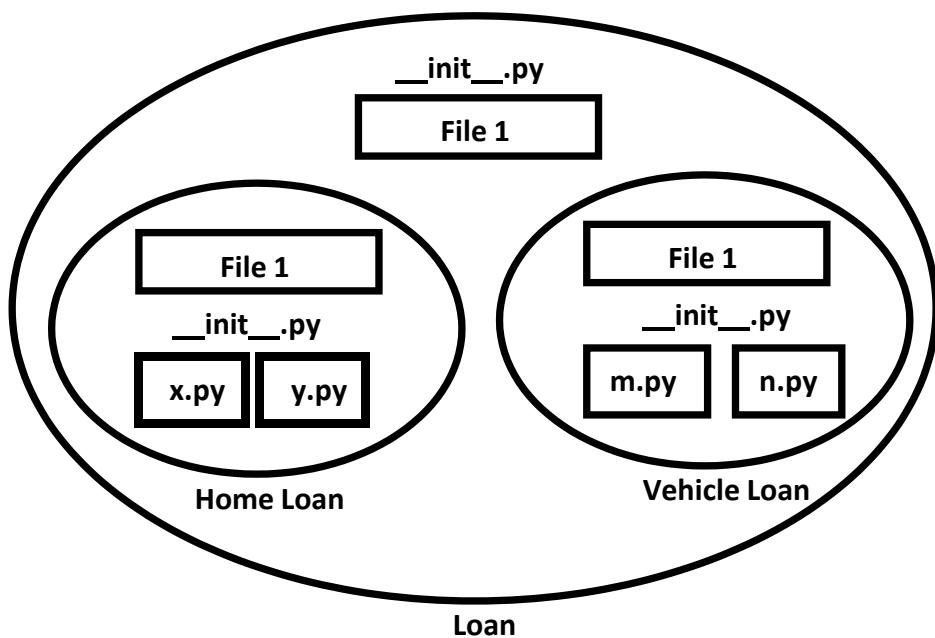
Bunny
pinny
Bunny
Sunny
Bunny
pinny
pinny
Vinny
Bunny
Sunny

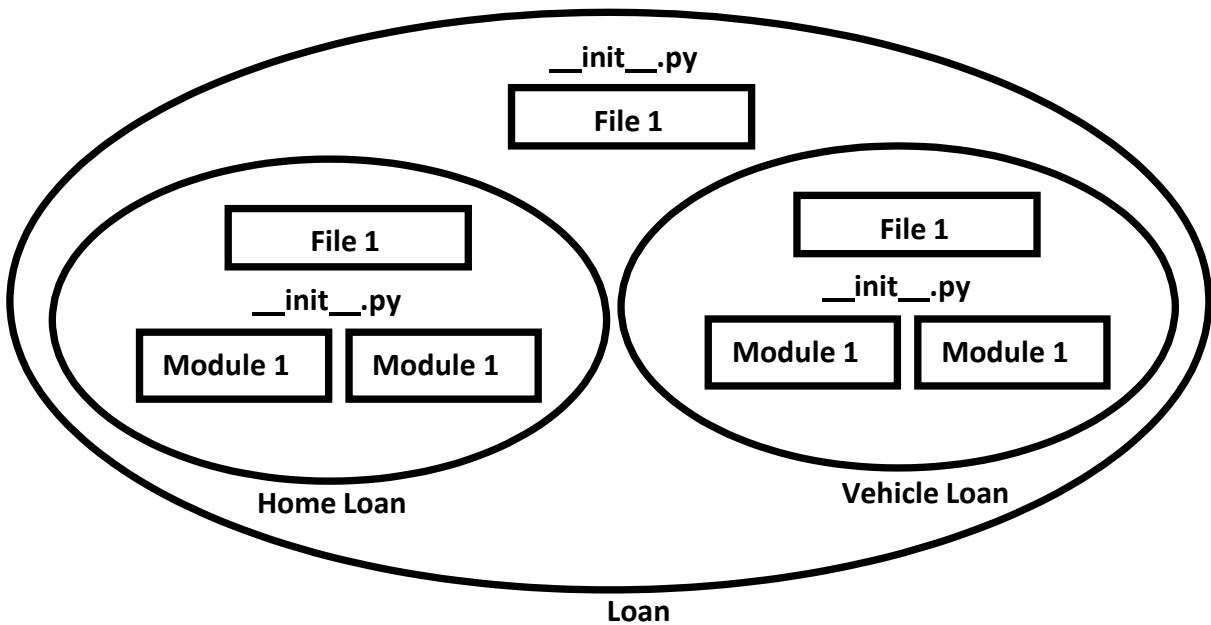
Packages

It is an encapsulation mechanism to group related modules into a single unit.
package is nothing but folder or directory which represents collection of Python modules.

Any folder or directory contains `__init__.py` file, is considered as a Python package. This file can be empty.

A package can contain sub packages also.





The main advantages of package statement are

1. We can resolve naming conflicts
2. We can identify our components uniquely
3. It improves modularity of the application

Eg 1:

```
D:\Python_classes>
| -test.py
|-pack1
| -module1.py
| -__init__.py
```

__init__.py:

empty file

module1.py:

```
def f1():
    print("Hello this is from module1 present in pack1")
```

test.py (version-1):

```
import pack1.module1
pack1.module1.f1()
```

test.py (version-2):

```
from pack1.module1 import f1
f1()
```

Eg 2:

```
D:\Python_classes>
  |-test.py
  |-com
    |-module1.py
    |-__init__.py
      |
      |-module2.py
      |-__init__.py
```

__init__.py:

empty file

module1.py:

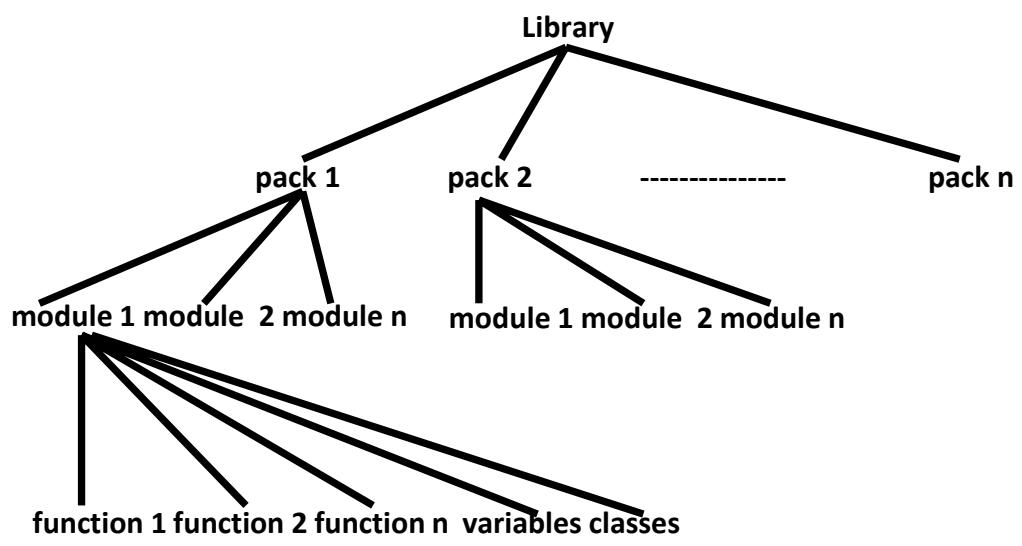
```
def f1():
    print("Hello this is from module1 present in com")
```

module2.py:

```
def f2():
    print("Hello this is from module2 present in com.")test.py:
```

1. **from com.module1 import f1**
2. **from com..module2 import f2**
3. **f1()**
4. **f2()**
- 5.
6. **Output**
7. **D:\Python_classes>py test.py**
8. **Hello this is from module1 present in com**
9. **Hello this is from module2 present in com.**

Note: Summary diagram of library,packages,modules which contains functions,classes and variables.



File Handling

As the part of programming requirement, we have to store our data permanently for future purpose. For this requirement we should go for files.

Files are very common permanent storage areas to store our data.

Types of Files:

There are 2 types of files

1. Text Files:

Usually we can use text files to store character data
eg: abc.txt

2. Binary Files:

Usually we can use binary files to store binary data like images,video files, audio files etc...

Opening a File:

Before performing any operation (like read or write) on the file,first we have to open that file.For this we should use Python's inbuilt function open()

But at the time of open, we have to specify mode,which represents the purpose of opening file.

```
f = open(filename, mode)
```

The allowed modes in Python are

1. r → open an existing file for read operation. The file pointer is positioned at the beginning of the file.If the specified file does not exist then we will get FileNotFoundError.This is default mode.

2. w → open an existing file for write operation. If the file already contains some data then it will be overridden. If the specified file is not already available then this mode will create that file.

3. a → open an existing file for append operation. It won't override existing data. If the specified file is not already available then this mode will create a new file.

4. r+ → To read and write data into the file. The previous data in the file will not be deleted. The file pointer is placed at the beginning of the file.

5. w+ → To write and read data. It will override existing data.

6. a+ → To append and read data from the file. It won't override existing data.

7. x → To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.

Note: All the above modes are applicable for text files. If the above modes suffixed with 'b' then these represent for binary files.

Eg: rb,wb,ab,r+b,w+b,a+b,xb

```
f = open("abc.txt","w")
```

We are opening abc.txt file for writing data.

Closing a File:

After completing our operations on the file, it is highly recommended to close the file. For this we have to use close() function.

```
f.close()
```

Various properties of File Object:

Once we open a file and we get file object, we can get various details related to that file by using its properties.

name → Name of opened file

mode → Mode in which the file is opened

closed → Returns boolean value indicates that file is closed or not

readable() → Returns boolean value indicates that whether file is readable or not

writable() → Returns boolean value indicates that whether file is writable or not.

Eg:

```
1) f=open("abc.txt",'w')
2) print("File Name: ",f.name)
3) print("File Mode: ",f.mode)
4) print("Is File Readable: ",f.readable())
5) print("Is File Writable: ",f.writable())
6) print("Is File Closed : ",f.closed)
7) f.close()
8) print("Is File Closed : ",f.closed)
9)
10)
11) Output
12) D:\Python_classes>py test.py
13) File Name: abc.txt
14) File Mode: w
15) Is File Readable: False
16) Is File Writable: True
17) Is File Closed : False
18) Is File Closed : True
```

Writing data to text files:

We can write character data to the text files by using the following 2 methods.

`write(str)`
`writelines(list of lines)`

Eg:

```
1) f=open("abcd.txt",'w')
2) f.write("ACYTech\n")
3) f.write("Software\n")
4) f.write("Solutions\n")
5) print("Data written to the file successfully")
6) f.close()
```

abcd.txt:

ACYTech
Software
Solutions

Note: In the above program, data present in the file will be overridden everytime if we run the program. Instead of overriding if we want append operation then we should open the file as follows.

```
f = open("abcd.txt","a")
```

Eg 2:

```
1) f=open("abcd.txt",'w')
2) list=["sunny\n","bunny\n","vinny\n","chinny"]
3) f.writelines(list)
4) print("List of lines written to the file successfully")
5) f.close()
```

abcd.txt:

sunny
bunny
vinny
chinny

Note: while writing data by using write() methods, compulsory we have to provide line separator(\n), otherwise total data should be written to a single line.

Reading Character Data from text files:

We can read character data from text file by using the following read methods.

read() → To read total data from the file
read(n) → To read 'n' characters from the file
readline() → To read only one line
readlines() → To read all lines into a list

Eg 1: To read total data from the file

```
1) f=open("abc.txt",'r')
2) data=f.read()
3) print(data)
4) f.close()
5)
6) Output
7) sunny
8) bunny
9) chinny
10) vinny
```

Eg 2: To read only first 10 characters:

```
1) f=open("abc.txt",'r')
2) data=f.read(10)
3) print(data)
4) f.close()
5)
```

- 6) Output
- 7) sunny
- 8) bunn

Eg 3: To read data line by line:

```
1) f=open("abc.txt",'r')
2) line1=f.readline()
3) print(line1,end="")
4) line2=f.readline()
5) print(line2,end="")
6) line3=f.readline()
7) print(line3,end="")
8) f.close()
9)
10) Output
11) sunny
12) bunny
13) chinny
```

Eg 4: To read all lines into list:

```
1) f=open("abc.txt",'r')
2) lines=f.readlines()
3) for line in lines:
4)     print(line,end="")
5) f.close()
6)
7) Output
8) sunny
9) bunny
10) chinny
11) vinny
```

Eg 5:

```
1) f=open("abc.txt","r")
2) print(f.read(3))
3) print(f.readline())
4) print(f.read(4))
5) print("Remaining data")
6) print(f.read())
7)
8) Output
9) sun
10) ny
11)
12) bunn
13) Remaining data
```

- 14) y
- 15) chinny
- 16) vinny

The with statement:

The with statement can be used while opening a file. We can use this to group file operation statements within a block.

The advantage of with statement is it will take care closing of file, after completing all operations automatically even in the case of exceptions also, and we are not required to close explicitly.

Eg:

- 1) with open("abc.txt","w") as f:
- 2) f.write("ACYTech\n")
- 3) f.write("Software\n")
- 4) f.write("Solutions\n")
- 5) print("Is File Closed: ",f.closed)
- 6) print("Is File Closed: ",f.closed)
- 7)
- 8) Output
- 9) Is File Closed: False
- 10) Is File Closed: True

The seek() and tell() methods:

tell():

The position(index) of first character in files is zero just like string index.

Eg:

- 1) f=open("abc.txt","r")
- 2) print(f.tell())
- 3) print(f.read(2))
- 4) print(f.tell())
- 5) print(f.read(3))
- 6) print(f.tell())

abc.txt:

sunny
bunny
chinny
vinny

Output:

```
0
su
2
nny
5
```

seek():

We can use seek() method to move cursor(file pointer) to specified location.
[Can you please seek the cursor to a particular location]

```
f.seek(offset, fromwhere)
```

offset represents the number of positions

The allowed values for second attribute(from where) are

0--- >From beginning of file(default value)

1>From current position

2>From end of the file

Note: Python 2 supports all 3 values but Python 3 supports only zero.

Eg:

```
1) data="All Students are STUPIDS"
2) f=open("abc.txt","w")
3) f.write(data)
4) with open("abc.txt","r+") as f:
5)     text=f.read()
6)     print(text)
7)     print("The Current Cursor Position: ",f.tell())
8)     f.seek(17)
9)     print("The Current Cursor Position: ",f.tell())
10)    f.write("GEMS!!!")
11)    f.seek(0)
12)    text=f.read()
13)    print("Data After Modification:")
14)    print(text)
15)
16) Output
17)
18) All Students are STUPIDS
19) The Current Cursor Position: 24
20) The Current Cursor Position: 17
21) Data After Modification:
```

| 22) All Students are GEMS!!!

How to check a particular file exists or not?

We can use os library to get information about files in our computer.

os module has path sub module, which contains isFile() function to check whether a particular file exists or not?

os.path.isfile(fname)

Q. Write a program to check whether the given file exists or not. If it is available then print its content?

```
1) import os,sys
2) fname=input("Enter File Name: ")
3) if os.path.isfile(fname):
4)     print("File exists:",fname)
5)     f=open(fname,"r")
6) else:
7)     print("File does not exist:",fname)
8)     sys.exit(0)
9) print("The content of file is:")
10) data=f.read()
11) print(data)
12)
13) Output
14) D:\Python_classes>py test.py
15) Enter File Name: ACYTech.txt
16) File does not exist: prabha.txt
17)
18) D:\Python_classes>py test.py
19) Enter File Name: abc.txt
20) File exists: abc.txt
21) The content of file is:
22) All Students are GEMS!!!
23) All Students are GEMS!!!
24) All Students are GEMS!!!
25) All Students are GEMS!!!
26) All Students are GEMS!!!
27) All Students are GEMS!!!
```

Note:

sys.exit(0) ==> To exit system without executing rest of the program.

argument represents status code . 0 means normal termination and it is the default value.

Q. Program to print the number of lines,words and characters present in the given file?

```
1) import os,sys
2) fname=input("Enter File Name: ")
3) if os.path.isfile(fname):
4)     print("File exists:",fname)
5)     f=open(fname,"r")
6) else:
7)     print("File does not exist:",fname)
8)     sys.exit(0)
9) lcount=wcount=ccount=0
10) for line in f:
11)     lcount=lcount+1
12)     ccount=ccount+len(line)
13)     words=line.split()
14)     wcount=wcount+len(words)
15) print("The number of Lines:",lcount)
16) print("The number of Words:",wcount)
17) print("The number of Characters:",ccount)
18)
19) Output
20) D:\Python_classes>py test.py
21) Enter File Name: ACYTech.txt
22) File does not exist:
ACYTech.txt23)
24) D:\Python_classes>py test.py
25) Enter File Name: abc.txt
26) File exists: abc.txt
27) The number of Lines: 6
28) The number of Words: 24
29) The number of Characters: 149
```

abc.txt:

All Students are GEMS!!!
All Students are GEMS!!!

Handling Binary Data:

It is very common requirement to read or write binary data like images,video files,audio files etc.

Q. Program to read image file and write to a new image file?

```
1) f1=open("rossum.jpg","rb")
2) f2=open("newpic.jpg","wb")
3) bytes=f1.read()
4) f2.write(bytes)
5) print("New Image is available with the name: newpic.jpg")
```

Handling csv files:

CSV==>Comma seperated values

As the part of programming,it is very common requirement to write and read data wrt csv files. Python provides csv module to handle csv files.

Writing data to csv file:

```
1) import csv
2) with open("emp.csv","w",newline="") as f:
3)     w=csv.writer(f) # returns csv writer object
4)     w.writerow(["ENO","ENAME","ESAL","EADDR"])
5)     n=int(input("Enter Number of Employees:"))
6)     for i in range(n):
7)         eno=input("Enter Employee No:")
8)         ename=input("Enter Employee Name:")
9)         esal=input("Enter Employee Salary:")
10)        eaddr=input("Enter Employee Address:")
11)        w.writerow([eno,ename,esal,eaddr])
12)    print("Total Employees data written to csv file successfully")
```

Note: Observe the difference with newline attribute and without

with open("emp.csv","w",newline="") as f:
with open("emp.csv","w") as f:

Note: If we are not using newline attribute then in the csv file blank lines will be included between data. To prevent these blank lines, newline attribute is required in Python-3,but in Python-2 just we can specify mode as 'wb' and we are not required to use newline attribute.

Reading Data from csv file:

```
1) import csv
2) f=open("emp.csv",'r')
3) r=csv.reader(f) #returns csv reader object
4) data=list(r)
5) #print(data)
6) for line in data:
7)     for word in line:
8)         print(word,"\t",end="")
9)     print()
10)
11) Output
12) D:\Python_classes>py test.py
13) ENO ENAME ESAL EADDR
14) 100 ACYTech 1000    Hyd
15) 200 Sachin 2000 Mumbai
16) 300 Dhoni 3000 Ranchi
```

Zipping and Unzipping Files:

It is very common requirement to zip and unzip files.

The main advantages are:

1. To improve memory utilization
2. We can reduce transport time
3. We can improve performance.

To perform zip and unzip operations, Python contains one in-built module zip file.

This module contains a class : ZipFile

To create Zip file:

We have to create ZipFile class object with name of the zip file, mode and constant ZIP_DEFLATED. This constant represents we are creating zip file.

```
f = ZipFile("files.zip","w","ZIP_DEFLATED")
```

Once we create ZipFile object, we can add files by using write() method.

```
f.write(filename)
```

Eg:

```
1) from zipfile import *
2) f=ZipFile("files.zip",'w',ZIP_DEFLATED)
3) f.write("file1.txt")
4) f.write("file2.txt")
5) f.write("file3.txt")
6) f.close()
7) print("files.zip file created successfully")
```

To perform unzip operation:

We have to create ZipFile object as follows

```
f = ZipFile("files.zip","r",ZIP_STORED)
```

ZIP_STORED represents unzip operation. This is default value and hence we are not required to specify.

Once we created ZipFile object for unzip operation, we can get all file names present in that zip file by using namelist() method.

```
names = f.namelist()
```

Eg:

```
1) from zipfile import *
2) f=ZipFile("files.zip",'r',ZIP_STORED)
3) names=f.namelist()
4) for name in names:
5)     print( "File Name: ",name)
6)     print("The Content of this file is:")
7)     f1=open(name,'r')
8)     print(f1.read())
9)     print()
```

Working with Directories:

It is very common requirement to perform operations for directories like

1. To know current working directory
 2. To create a new directory
 3. To remove an existing directory
 4. To rename a directory
 5. To list contents of the directory
- etc...

To perform these operations, Python provides inbuilt module os, which contains several functions to perform directory related operations.

Q1. To Know Current Working Directory:

```
import os  
cwd=os.getcwd()  
print("Current Working Directory:", cwd)
```

Q2. To create a sub directory in the current working directory:

```
import os  
os.mkdir("mysub")  
print("mysub directory created in cwd")
```

Q3. To create a sub directory in mysub directory:

```
        cwd  
        |-mysub  
            |-mysub2  
  
import os  
os.mkdir("mysub/mysub2")  
print("mysub2 created inside mysub")
```

Note: Assume mysub already present in cwd.

Q4. To create multiple directories like sub1 in that sub2 in that sub3:

```
import os  
os.makedirs("sub1/sub2/sub3")  
print("sub1 and in that sub2 and in that sub3 directories created")
```

Q5. To remove a directory:

```
import os  
os.rmdir("mysub/mysub2")  
print("mysub2 directory deleted")
```

Q6. To remove multiple directories in the path:

```
import os  
os.removedirs("sub1/sub2/sub3")  
print("All 3 directories sub1,sub2 and sub3 removed")
```

Q7. To rename a directory:

```
import os  
os.rename("mysub","newdir")  
print("mysub directory renamed to newdir")
```

Q8. To know contents of directory:

os module provides listdir() to list out the contents of the specified directory. It won't display the contents of sub directory.

Eg:

```
1) import os  
2) print(os.listdir("."))  
3)  
4) Output  
5) D:\Python_classes>py test.py  
6) ['abc.py', 'abc.txt', 'abcd.txt', 'com', 'demo.py', 'prabhamath.py', 'emp.csv', '  
7) file1.txt', 'file2.txt', 'file3.txt', 'files.zip', 'log.txt', 'module1.py', 'myl  
8) og.txt', 'newdir', 'newpic.jpg', 'pack1', 'rossum.jpg', 'test.py', '__pycache__'  
9) ]
```

The above program display contents of current working directory but not contents of sub directories.

If we want the contents of a directory including sub directories then we should go for walk() function.

Q9. To know contents of directory including sub directories:

We have to use walk() function

[Can you please walk in the directory so that we can aware all contents of that directory]

```
os.walk(path,topdown=True,onerror=None,followlinks=False)
```

It returns an Iterator object whose contents can be displayed by using for loop

path-->Directory path. cwd means .

topdown=True --->Travel from top to bottom

onerror=None --->on error detected which function has to execute.

followlinks=True -->To visit directories pointed by symbolic links

Eg: To display all contents of Current working directory including sub directories:

```
1) import os
2) for dirname,dirnames,filenames in os.walk('.'):
3)     print("Current Directory Path:",dirname)
4)     print("Directories:",dirnames)
5)     print("Files:",filenames)
6)     print()
7)
8)
9) Output
10) Current Directory Path: .
11) Directories: ['com', 'newdir', 'pack1', '__pycache__']
12) Files: ['abc.txt', 'abcd.txt', 'demo.py', 'ACYTechmath.py', 'emp.csv', 'file1.txt'
13) , 'file2.txt', 'file3.txt', 'files.zip', 'log.txt', 'module1.py', 'mylog.txt', '
14) newpic.jpg', 'rossum.jpg', 'test.py']
15)
16) Current Directory Path: .\com
17) Directories: ['', '__pycache__']
18) Files: ['module1.py', '__init__.py']
19)
20) ...
```

Note: To display contents of particular directory,we have to provide that directory name as argument to walk() function.

`os.walk("directoryname")`

Q. What is the difference between listdir() and walk() functions?

In the case of listdir(), we will get contents of specified directory but not sub directory contents. But in the case of walk() function we will get contents of specified directory and its sub directories also.

Running Other programs from Python program:

`os` module contains `system()` function to run programs and commands.
It is exactly same as `system()` function in C language.

`os.system("command string")`

The argument is any command which is executing from DOS.

Eg:

```
import os
os.system("dir *.py")
os.system("py abc.py")
```

How to get information about a File:

We can get statistics of a file like size, last accessed time, last modified time etc by using stat() function of os module.

```
stats = os.stat("abc.txt")
```

The statistics of a file includes the following parameters:

st_mode==>Protection Bits
st_ino==>Inode number
st_dev==>device
st_nlink==>no of hard links
st_uid==>userid of owner
st_gid==>group id of owner
st_size==>size of file in bytes
st_atime==>Time of most recent access
st_mtime==>Time of Most recent modification
st_ctime==> Time of Most recent meta data change

Note:

st_atime, st_mtime and st_ctime returns the time as number of milli seconds since Jan 1st 1970 ,12:00AM. By using datetime module fromtimestamp() function, we can get exact date and time.

Q. To print all statistics of file abc.txt:

```
1) import os
2) stats=os.stat("abc.txt")
3) print(stats)
4)
5) Output
6) os.stat_result(st_mode=33206, st_ino=844424930132788, st_dev=2657980798, st_nlin
7) k=1, st_uid=0, st_gid=0, st_size=22410, st_atime=1505451446, st_mtime=1505538999
8) , st_ctime=1505451446)
```

Q. To print specified properties:

```
1) import os
2) from datetime import *
3) stats=os.stat("abc.txt")
4) print("File Size in Bytes:",stats.st_size)
5) print("File Last Accessed Time:",datetime.fromtimestamp(stats.st_atime))
6) print("File Last Modified Time:",datetime.fromtimestamp(stats.st_mtime))
7)
```

- 8) Output
- 9) File Size in Bytes: 22410
- 10) File Last Accessed Time: 2017-09-15 10:27:26.599490
- 11) File Last Modified Time: 2017-09-16 10:46:39.245394

Pickling and Unpickling of Objects:

Sometimes we have to write total state of object to the file and we have to read total object from the file.

The process of writing state of object to the file is called pickling and the process of reading state of an object from the file is called unpickling.

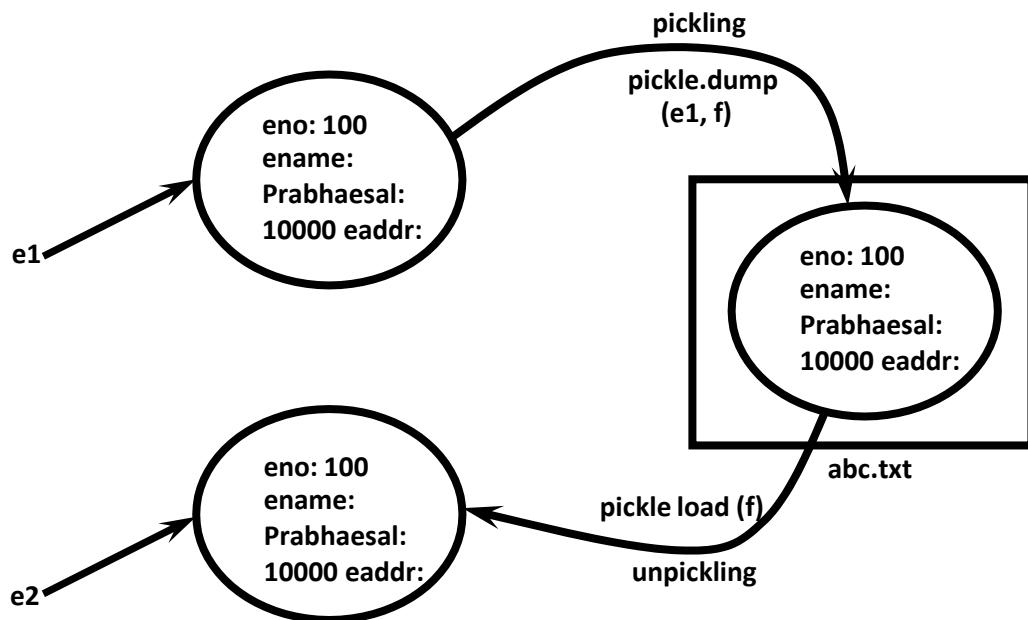
We can implement pickling and unpickling by using pickle module of Python.

pickle module contains dump() function to perform pickling.

```
pickle.dump(object,file)
```

pickle module contains load() function to perform unpickling

```
obj=pickle.load(file)
```



Writing and Reading State of object by using pickle Module:

```
1) import pickle
2) class Employee:
3)     def __init__(self,eno,ename,esal,eaddr):
4)         self.eno=eno;
5)         self.ename=ename;
6)         self.esal=esal;
7)         self.eaddr=eaddr;
8)     def display(self):
9)         print(self.eno,"\\t",self.ename,"\\t",self.esal,"\\t",self.eaddr)
10)    with open("emp.dat","wb") as f:
11)        e=Employee(100,"ACYTech",1000,"Hyd")
12)        pickle.dump(e,f)
13)        print("Pickling of Employee Object completed...")
14)
15)    with open("emp.dat","rb") as f:
16)        obj=pickle.load(f)
17)        print("Printing Employee Information after unpickling")
18)        obj.display()
```

Writing Multiple Employee Objects to the file:

emp.py:

```
1) class Employee:
2)     def __init__(self,eno,ename,esal,eaddr):
3)         self.eno=eno;
4)         self.ename=ename;
5)         self.esal=esal;
6)         self.eaddr=eaddr;
7)     def display(self):
8)
9)     print(self.eno,"\\t",self.ename,"\\t",self.esal,"\\t",self.eaddr)
```

pick.py:

```
1) import emp,pickle
2) f=open("emp.dat","wb")
3) n=int(input("Enter The number of Employees:"))
4) for i in range(n):
5)     eno=int(input("Enter Employee Number:"))
6)     ename=input("Enter Employee Name:")
7)     esal=float(input("Enter Employee Salary:"))
8)     eaddr=input("Enter Employee Address:")
9)     e=emp.Employee(eno,ename,esal,eaddr)
10)    pickle.dump(e,f)
11)    print("Employee Objects pickled successfully")
```

unpick.py:

```
1) import emp,pickle
2) f=open("emp.dat","rb")
3) print("Employee Details:")
4) while True:
5)     try:
6)         obj=pickle.load(f)
7)         obj.display()
8)     except EOFError:
9)         print("All employees Completed")
10)        break
11) f.close()
```

Exception Handling

In any programming language there are 2 types of errors are possible.

1. Syntax Errors
2. Runtime Errors

1. Syntax Errors:

The errors which occurs because of invalid syntax are called syntax errors.

Eg 1:

```
x=10  
if x==10  
    print("Hello")
```

SyntaxError: invalid syntax

Eg 2:

```
print "Hello"
```

SyntaxError: Missing parentheses in call to 'print'

Note:

Programmer is responsible to correct these syntax errors. Once all syntax errors are corrected then only program execution will be started.

2. Runtime Errors:

Also known as exceptions.

While executing the program if something goes wrong because of end user input or programming logic or memory problems etc then we will get Runtime Errors.

Eg: `print(10/0) ==>ZeroDivisionError: division by zero`

```
print(10/"ten") ==>TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

```
x=int(input("Enter Number:"))  
print(x)
```

```
D:\Python_classes>py test.py
```

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

Note: Exception Handling concept applicable for Runtime Errors but not for syntax errors

What is Exception:

An unwanted and unexpected event that disturbs normal flow of program is called exception.

Eg:

ZeroDivisionError
TypeError
ValueError
FileNotFoundException
EOFError
SleepingError
TyrePuncturedError

It is highly recommended to handle exceptions. The main objective of exception handling is Graceful Termination of the program(i.e we should not block our resources and we should not miss anything)

Exception handling does not mean repairing exception. We have to define alternative way to continue rest of the program normally.

Eg:

For example our programming requirement is reading data from remote file locating at London. At runtime if london file is not available then the program should not be terminated abnormally. We have to provide local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

try:

```
    read data from remote file locating at london
except FileNotFoundException:
    use local file and continue rest of the program normally
```

- Q. What is an Exception?**
- Q. What is the purpose of Exception Handling?**
- Q. What is the meaning of Exception Handling?**

Default Exception Handing in Python:

Every exception in Python is an object. For every exception type the corresponding classes are available.

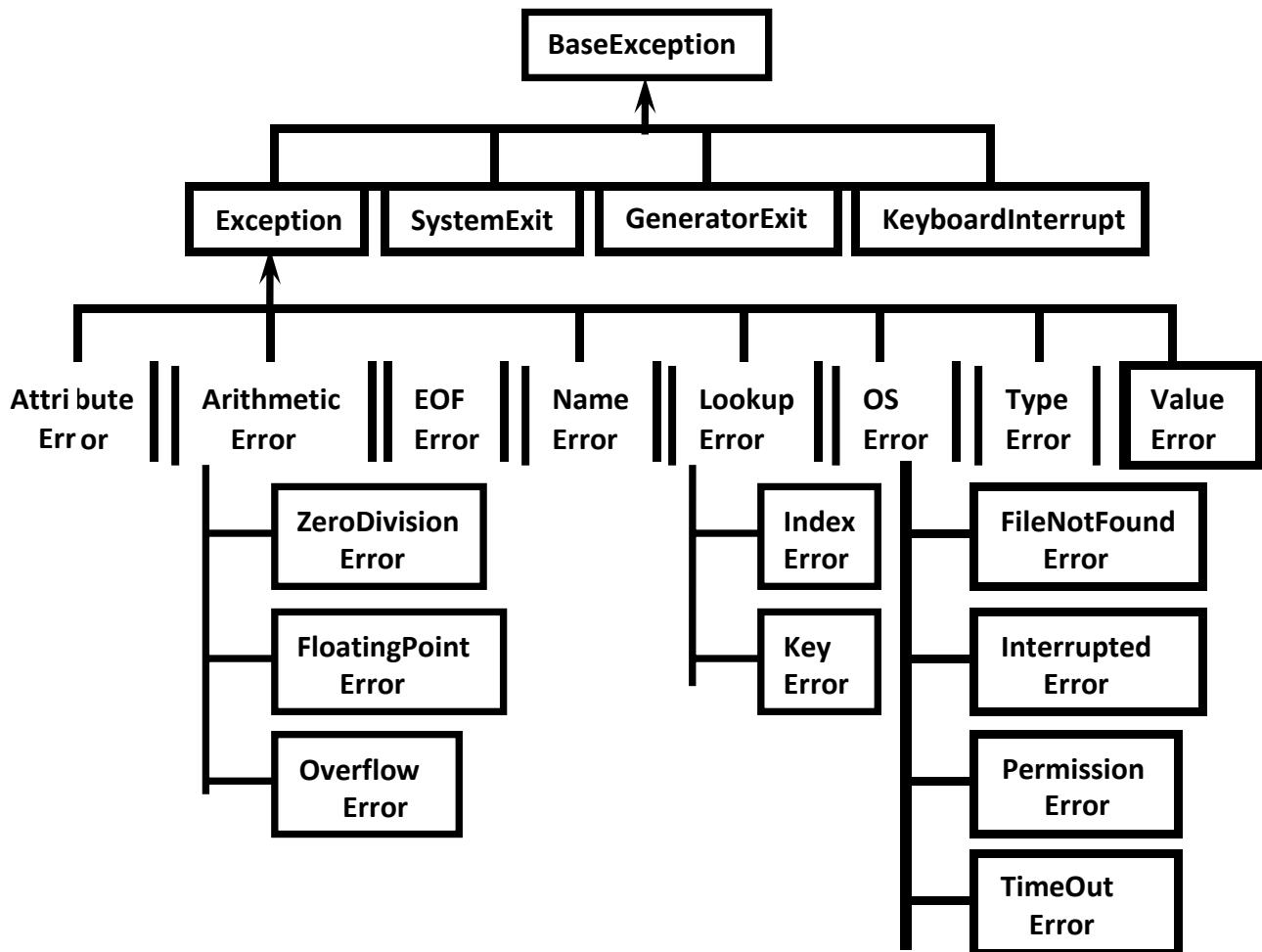
Whenever an exception occurs PVM will create the corresponding exception object and will check for handling code. If handling code is not available then Python interpreter terminates the program abnormally and prints corresponding exception information to the console.

The rest of the program won't be executed.

Eg:

```
1) print("Hello")
2) print(10/0)
3) print("Hi")
4)
5) D:\Python_classes>py test.py
6) Hello
7) Traceback (most recent call last):
8)   File "test.py", line 2, in <module>
9)     print(10/0)
10)    ZeroDivisionError: division by zero
```

Python's Exception Hierarchy



Every Exception in Python is a class.

All exception classes are child classes of `BaseException`. i.e every exception class extends `BaseException` either directly or indirectly. Hence `BaseException` acts as root for Python Exception Hierarchy.

Most of the times being a programmer we have to concentrate `Exception` and its child classes.

Customized Exception Handling by using try-except:

It is highly recommended to handle exceptions.

The code which may raise exception is called risky code and we have to take risky code inside `try` block. The corresponding handling code we have to take inside `except` block.

```
try:
    Risky Code
except XXX:
    Handling code/Alternative Code
```

without try-except:

1. `print("stmt-1")`
2. `print(10/0)`
3. `print("stmt-3")`
- 4.
5. Output
6. `stmt-1`
7. `ZeroDivisionError: division by zero`

Abnormal termination/Non-Graceful Termination

with try-except:

1. `print("stmt-1")`
2. `try:`
3. `print(10/0)`
4. `except ZeroDivisionError:`
5. `print(10/2)`
6. `print("stmt-3")`
- 7.
8. Output
9. `stmt-1`
10. `5.0`
11. `stmt-3`

Normal termination/Graceful Termination

Control Flow in try-except:

```
try:
    stmt-1
    stmt-2
    stmt-3
except XXX:
    stmt-4
stmt-5
```

case-1: If there is no exception
1,2,3,5 and Normal Termination

case-2: If an exception raised at stmt-2 and corresponding except block matched

1,4,5 Normal Termination

case-3: If an exception raised at stmt-2 and corresponding except block not matched

1, Abnormal Termination

case-4: If an exception raised at stmt-4 or at stmt-5 then it is always abnormal termination.

Conclusions:

1. within the try block if anywhere exception raised then rest of the try block wont be executed eventhough we handled that exception. Hence we have to take only risky code inside try block and length of the try block should be as less as possible.

2. In addition to try block,there may be a chance of raising exceptions inside except and finally blocks also.

3. If any statement which is not part of try block raises an exception then it is always abnormal termination.

How to print exception information:

try:

1. `print(10/0)`
2. `except ZeroDivisionError as msg:`
3. `print("exception raised and its description is:",msg)`
- 4.
5. Output exception raised and its description is: division by zero

try with multiple except blocks:

The way of handling exception is varied from exception to exception.Hence for every exception type a seperate except block we have to provide. i.e try with multiple except blocks is possible and recommended to use.

Eg:

try:

except ZeroDivisionError:
 perform alternative
 arithmetic operations

```
except FileNotFoundError:  
    use local file instead of remote file
```

If try with multiple except blocks available then based on raised exception the corresponding except block will be executed.

Eg:

```
1) try:  
2)     x=int(input("Enter First Number: "))  
3)     y=int(input("Enter Second Number: "))  
4)     print(x/y)  
5) except ZeroDivisionError :  
6)     print("Can't Divide with Zero")  
7) except ValueError:  
8)     print("please provide int value only")  
9)  
10) D:\Python_classes>py test.py  
11) Enter First Number: 10  
12) Enter Second Number: 2  
13) 5.0  
14)  
15) D:\Python_classes>py test.py  
16) Enter First Number: 10  
17) Enter Second Number: 0  
18) Can't Divide with Zero  
19)  
20) D:\Python_classes>py test.py  
21) Enter First Number: 10  
22) Enter Second Number: ten  
23) please provide int value only
```

If try with multiple except blocks available then the order of these except blocks is important .Python interpreter will always consider from top to bottom until matched except block identified.

Eg:

```
1) try:  
2)     x=int(input("Enter First Number: "))  
3)     y=int(input("Enter Second Number: "))  
4)     print(x/y)  
5) except ArithmeticError :  
6)     print("ArithmetricError")  
7) except ZeroDivisionError:  
8)     print("ZeroDivisionError")  
9)  
10) D:\Python_classes>py test.py
```

- 11) Enter First Number: 10
- 12) Enter Second Number: 0
- 13) ArithmeticError

Single except block that can handle multiple exceptions:

We can write a single except block that can handle multiple different types of exceptions.

```
except (Exception1,Exception2,exception3,..): or  
except (Exception1,Exception2,exception3,..) as msg :
```

Parenthesis are mandatory and this group of exceptions internally considered as tuple.

Eg:

- 1) try:
- 2) x=int(input("Enter First Number: "))
- 3) y=int(input("Enter Second Number: "))
- 4) print(x/y)
- 5) except (ZeroDivisionError,ValueError) as msg:
- 6) print("Plz Provide valid numbers only and problem is: ",msg)
- 7)
- 8) D:\Python_classes>py test.py
- 9) Enter First Number: 10
- 10) Enter Second Number: 0
- 11) Plz Provide valid numbers only and problem is: division by zero
- 12)
- 13) D:\Python_classes>py test.py
- 14) Enter First Number: 10
- 15) Enter Second Number: ten
- 16) Plz Provide valid numbers only and problem is: invalid literal for int() with b
- 17) ase 10: 'ten'

Default except block:

We can use default except block to handle any type of exceptions.

In default except block generally we can print normal error messages.

Syntax:

```
except:  
    statements
```

Eg:

- 1) try:
- 2) x=int(input("Enter First Number: "))
- 3) y=int(input("Enter Second Number: "))
- 4) print(x/y)

```
5) except ZeroDivisionError:  
6)     print("ZeroDivisionError:Can't divide with zero")  
7) except:  
8)     print("Default Except:Plz provide valid input only")  
9)  
10) D:\Python_classes>py test.py  
11) Enter First Number: 10  
12) Enter Second Number: 0  
13) ZeroDivisionError:Can't divide with zero  
14)  
15) D:\Python_classes>py test.py  
16) Enter First Number: 10  
17) Enter Second Number: ten  
18) Default Except:Plz provide valid input only
```

*****Note:** If try with multiple except blocks available then default except block should be last, otherwise we will get SyntaxError.

Eg:

```
1) try:  
2)     print(10/0)  
3) except:  
4)     print("Default Except")  
5) except ZeroDivisionError:  
6)     print("ZeroDivisionError")  
7)  
8) SyntaxError: default 'except:' must be last
```

Note:

The following are various possible combinations of except blocks

1. except ZeroDivisionError:
1. except ZeroDivisionError as msg:
3. except (ZeroDivisionError,ValueError) :
4. except (ZeroDivisionError,ValueError) as msg:
5. except :

finally block:

1. It is not recommended to maintain clean up code(Resource Deallocating Code or Resource Releasing code) inside try block because there is no guarantee for the execution of every statement inside try block always.
2. It is not recommended to maintain clean up code inside except block, because if there is no exception then except block won't be executed.

Hence we required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. Such type of best place is nothing but finally block.

Hence the main purpose of finally block is to maintain clean up code.

try:

 Risky Code

except:

 Handling Code

finally:

 Cleanup code

The speciality of finally block is it will be executed always whether exception raised or not raised and whether exception handled or not handled.

Case-1: If there is no exception

```
1) try:  
2)     print("try")  
3) except:  
4)     print("except")  
5) finally:  
6)     print("finally")  
7)  
8) Output  
9) try  
10) finally
```

Case-2: If there is an exception raised but handled:

```
1) try:  
2)     print("try")  
3)     print(10/0)  
4) except ZeroDivisionError:  
5)     print("except")  
6) finally:  
7)     print("finally")  
8)  
9) Output  
10) try  
11) except  
12) finally
```

Case-3: If there is an exception raised but not handled:

```
1) try:  
2)   print("try")  
3)   print(10/0)  
4) except NameError:  
5)   print("except")  
6) finally:  
7)   print("finally")  
8)  
9) Output  
10) try  
11) finally  
12) ZeroDivisionError: division by zero(Abnormal Termination)
```

*** **Note:** There is only one situation where finally block won't be executed ie whenever we are using os._exit(0) function.

Whenever we are using os._exit(0) function then Python Virtual Machine itself will be shutdown. In this particular case finally won't be executed.

```
1) imports  
2) try:  
3)   print("try")  
4)   os._exit(0)  
5) except NameError:  
6)   print("except")  
7) finally:  
8)   print("finally")  
9)  
10) Output  
11) try
```

Note:

os._exit(0)

where 0 represents status code and it indicates normal termination

There are multiple status codes are possible.

Control flow in try-except-finally:

```
try:  
  stmt-1  
  stmt-2  
  stmt-3  
except:  
  stmt-4
```

```
finally:  
    stmt-5  
stmt6
```

Case-1: If there is no exception

1,2,3,5,6 Normal Termination

Case-2: If an exception raised at stmt2 and the corresponding except block matched

1,4,5,6 Normal Termination

Case-3: If an exception raised at stmt2 but the corresponding except block not matched

1,5 Abnormal Termination

Case-4: If an exception raised at stmt4 then it is always abnormal termination but before that finally block will be executed.

Case-5: If an exception raised at stmt-5 or at stmt-6 then it is always abnormal termination

Nested try-except-finally blocks:

We can take try-except-finally blocks inside try or except or finally blocks.i.e nesting of try-except-finally is possible.

try:

```
-----  
-----  
-----
```

try:

```
-----  
-----  
-----
```

except:

```
-----  
-----  
-----
```

except:

```
-----  
-----  
-----
```

General Risky code we have to take inside outer try block and too much risky code we have to take inside inner try block. Inside Inner try block if an exception raised then inner

except block is responsible to handle. If it is unable to handle then outer except block is responsible to handle.

Eg:

```
1) try:  
2)   print("outer try block")  
3)   try:  
4)     print("Inner try block")  
5)     print(10/0)  
6)   except ZeroDivisionError:  
7)     print("Inner except block")  
8)   finally:  
9)     print("Inner finally block")  
10) except:  
11)   print("outer except block")  
12) finally:  
13)   print("outer finally block")  
14)  
15) Output  
16) outer try block  
17) Inner try block  
18) Inner except block  
19) Inner finally block  
20) outer finally block
```

Control flow in nested try-except-finally:

try:

```
    stmt-1  
    stmt-2  
    stmt-3  
    try:
```

```
        stmt-4  
        stmt-5  
        stmt-6
```

except X:

```
        stmt-7
```

finally:

```
        stmt-8
```

stmt-9

except Y:

```
    stmt-10
```

finally:

```
    stmt-11
```

stmt-12

case-1: If there is no exception

1,2,3,4,5,6,8,9,11,12 Normal Termination

case-2: If an exception raised at stmt-2 and the corresponding except block matched

1,10,11,12 Normal Termination

case-3: If an exception raised at stmt-2 and the corresponding except block not matched

1,11,Abnormal Termination

case-4: If an exception raised at stmt-5 and inner except block matched

1,2,3,4,7,8,9,11,12 Normal Termination

case-5: If an exception raised at stmt-5 and inner except block not matched but outer except block matched

1,2,3,4,8,10,11,12,Normal Termination

case-6:If an exception raised at stmt-5 and both inner and outer except blocks are not matched

1,2,3,4,8,11,Abnormal Termination

case-7: If an exception raised at stmt-7 and corresponding except block matched

1,2,3,....,8,10,11,12,Normal Termination

case-8: If an exception raised at stmt-7 and corresponding except block not matched

1,2,3,....,8,11,Abnormal Termination

case-9: If an exception raised at stmt-8 and corresponding except block matched

1,2,3,....,10,11,12 Normal Termination

case-10: If an exception raised at stmt-8 and corresponding except block not matched

1,2,3,....,11,Abnormal Termination

case-11: If an exception raised at stmt-9 and corresponding except block matched

1,2,3,....,8,10,11,12,Normal Termination

case-12: If an exception raised at stmt-9 and corresponding except block not matched

1,2,3,....,8,11,Abnormal Termination

case-13: If an exception raised at stmt-10 then it is always abnormal termination but

before abnormal termination finally block(stmt-11) will be executed.

Case-14: If an exception raised at stmt-11 or stmt-12 then it is always abnormal termination.

Note: If the control entered into try block then compulsory finally block will be executed. If the control not entered into try block then finally block won't be executed.

else block with try-except-finally:

We can use else block with try-except-finally blocks.

else block will be executed if and only if there are no exceptions inside try block.

try:

Risky Code

except:

will be executed if exception inside try

else:

will be executed if there is no exception inside try

finally:

will be executed whether exception raised or not raised and handled or not

handled

Eg:

try:

```
print("try")
print(10/0)-->1
```

except:

print("except")

else:

print("else")

finally:

print("finally")

If we comment line-1 then else block will be executed b'z there is no exception inside try.
In this case the output is:

```
try
else
finally
```

If we are not commenting line-1 then else block won't be executed b'z there is exception inside try block. In this case output is:

```
try  
except  
finally
```

Various possible combinations of try-except-else-finally:

1. Whenever we are writing try block, compulsory we should write except or finally block.i.e without except or finally block we cannot write try block.
2. Whenever we are writing except block, compulsory we should write try block. i.e except without try is always invalid.
3. Whenever we are writing finally block, compulsory we should write try block. i.e finally without try is always invalid.
4. We can write multiple except blocks for the same try, but we cannot write multiple finally blocks for the same try
5. Whenever we are writing else block compulsory except block should be there. i.e without except we cannot write else block.
6. In try-except-else-finally order is important.
7. We can define try-except-else-finally inside try, except, else and finally blocks. i.e nesting of try-except-else-finally is always possible.

1	try: print("try")	✗
2	except: print("Hello")	✗
3	else: print("Hello")	✗
4	finally: print("Hello")	✗
5	try: print("try") except: print("except")	✓
6	try: print("try") finally: print("finally")	✓
7	try: print("try")	✓

	<pre>except: print("except")</pre>	
	<pre>else: print("else")</pre>	
8	<pre>try: print("try") else: print("else")</pre>	✗
9	<pre>try: print("try") else: print("else") finally: print("finally")</pre>	✗
10	<pre>try: print("try") except XXX: print("except-1") except YYY: print("except-2")</pre>	✓
11	<pre>try: print("try") except : print("except-1") else: print("else") else: print("else")</pre>	✗
12	<pre>try: print("try") except : print("except-1") finally: print("finally") finally: print("finally")</pre>	✗
13	<pre>try: print("try") print("Hello") except: print("except")</pre>	✗
14	<pre>try: print("try") except:</pre>	✗

	<pre> print("except") print("Hello") except: print("except") </pre>	
15	<pre> try: print("try") except: print("except") print("Hello") finally: print("finally") </pre>	✗
16	<pre> try: print("try") except: print("except") print("Hello") else: print("else") </pre>	✗
17	<pre> try: print("try") except: print("except") try: print("try") except: print("except") </pre>	✓
18	<pre> try: print("try") except: print("except") try: print("try") finally: print("finally") </pre>	✓
19	<pre> try: print("try") except: print("except") if 10>20: print("if") else: print("else") </pre>	✗
20	<pre> try: print("try") </pre>	✓

	<pre> try: print("inner try") except: print("inner except block") finally: print("inner finally block") except: print("except") </pre>	
21	<pre> try: print("try") except: print("except") try: print("inner try") except: print("inner except block") finally: print("inner finally block") </pre>	✓
22	<pre> try: print("try") except: print("except") finally: try: print("inner try") except: print("inner except block") finally: print("inner finally block") </pre>	✓
23	<pre> try: print("try") except: print("except") try: print("try") else: print("else") </pre>	✗
24	<pre> try: print("try") try: print("inner try") except: print("except") </pre>	✗

25	<pre> try: print("try") else: print("else") except: print("except") finally: print("finally") </pre>	X
----	--	---

Types of Exceptions:

In Python there are 2 types of exceptions are possible.

1. Predefined Exceptions
2. User Defined Exceptions

1. Predefined Exceptions:

Also known as in-built exceptions

The exceptions which are raised automatically by Python virtual machine whenever a particular event occurs, are called pre defined exceptions.

Eg 1: Whenever we are trying to perform Division by zero, automatically Python will raise ZeroDivisionError.

```
print(10/0)
```

Eg 2: Whenever we are trying to convert input value to int type and if input value is not int value then Python will raise ValueError automatically

```
x=int("ten")====>ValueError
```

2. User Defined Exceptions:

Also known as Customized Exceptions or Programmatic Exceptions

Some time we have to define and raise exceptions explicitly to indicate that something goes wrong ,such type of exceptions are called User Defined Exceptions or Customized Exceptions

Programmer is responsible to define these exceptions and Python not having any idea about these. Hence we have to raise explicitly based on our requirement by using "raise" keyword.

Eg:

InSufficientFundsException
InvalidInputException
TooYoungException
TooOldException

How to Define and Raise Customized Exceptions:

Every exception in Python is a class that extends Exception class either directly or indirectly.

Syntax:

```
class classname(predefined exception class name):  
    def __init__(self,arg):  
        self.msg=arg
```

Eg:

```
1) class TooYoungException(Exception):  
2)     def __init__(self,arg):  
3)         self.msg=arg
```

TooYoungException is our class name which is the child class of Exception

We can raise exception by using raise keyword as follows

```
raise TooYoungException("message")
```

Eg:

```
1) class TooYoungException(Exception):  
2)     def __init__(self,arg):  
3)         self.msg=arg  
4)  
5) class TooOldException(Exception):  
6)     def __init__(self,arg):  
7)         self.msg=arg  
8)  
9) age=int(input("Enter Age:"))  
10) if age>60:  
11)     raise TooYoungException("Plz wait some more time you will get best match soon!!!")  
12) elif age<18:  
13)     raise TooOldException("Your age already crossed marriage age...no chance of getting ma  
rriage")  
14) else:  
15)     print("You will get match details soon by email!!!!")  
16)  
17) D:\Python_classes>py test.py
```

```
18) Enter Age:90
19) __main__.TooYoungException: Plz wait some more time you will get best match soon!!!
20)
21) D:\Python_classes>py test.py
22) Enter Age:12
23) __main__.TooOldException: Your age already crossed marriage age...no chance of g
24) etting marriage
25)
26) D:\Python_classes>py test.py
27) Enter Age:27
28) You will get match details soon by email!!!
```

Note:

raise keyword is best suitable for customized exceptions but not for pre defined exceptions

Python's Object Oriented Programming (OOPs)

What is Class:

- ⚽ In Python every thing is an object. To create objects we required some Model or Plan or Blueprint, which is nothing but class.
- ⚽ We can write a class to represent properties (attributes) and actions (behaviour) of object.
- ⚽ Properties can be represented by variables
- ⚽ Actions can be represented by Methods.
- ⚽ Hence class contains both variables and methods.

How to Define a class?

We can define a class by using class keyword.

Syntax:

```
class className:  
    """ documentation string ""  
    variables: instance variables, static and local variables  
    methods: instance methods, static methods, class methods
```

Documentation string represents description of the class. Within the class doc string is always optional. We can get doc string by using the following 2 ways.

1. `print(classname.__doc__)`
2. `help(classname)`

Example:

```
1) class Student:  
2)     """ This is student class with required data ""  
3)     print(Student.__doc__)  
4) help(Student)
```

Within the Python class we can represent data by using variables.

There are 3 types of variables are allowed.

1. Instance Variables (Object Level Variables)
2. Static Variables (Class Level Variables)
3. Local variables (Method Level Variables)

Within the Python class, we can represent operations by using methods. The following are various types of allowed methods

1. Instance Methods
2. Class Methods
3. Static Methods

Example for class:

```
1) class Student:  
2)     """Developed by prabha for python demo""  
3)     def __init__(self):  
4)         self.name='prabh'  
5)         self.age=40  
6)         self.marks=80  
7)  
8)     def talk(self):  
9)         print("Hello I am :",self.name)  
10)        print("My Age is:",self.age)  
11)        print("My Marks are:",self.marks)
```

What is Object:

Physical existence of a class is nothing but object. We can create any number of objects for a class.

Syntax to create object: referencevariable = classname()

Example: s = Student()

What is Reference Variable:

The variable which can be used to refer object is called reference variable.
By using reference variable, we can access properties and methods of object.

Program: Write a Python program to create a Student class and Creates an object to it. Call the method talk() to display student details

```
1) class Student:  
2)  
3)     def __init__(self,name,rollno,marks):  
4)         self.name=name  
5)         self.rollno=rollno  
6)         self.marks=marks  
7)  
8)     def talk(self):  
9)         print("Hello My Name is:",self.name)  
10)        print("My Rollno is:",self.rollno)  
11)        print("My Marks are:",self.marks)  
12)
```

```
13) s1=Student("ACYTech",101,80)
14) s1.talk()
```

Output:

```
D:\ACYTechclasses>py
test.pyHello My Name is:
ACYTech My Rollno is: 101
My Marks are: 80
```

Self variable:

self is the default variable which is always pointing to current object (like this keyword in Java)

By using self we can access instance variables and instance methods of object.

Note:

1. self should be first parameter inside constructor

```
def __init__(self):
```

2. self should be first parameter inside instance methods

```
def talk(self):
```

Constructor Concept:

- 👉 Constructor is a special method in python.
- 👉 The name of the constructor should be `__init__(self)`
- 👉 Constructor will be executed automatically at the time of object creation.
- 👉 The main purpose of constructor is to declare and initialize instance variables.
- 👉 Per object constructor will be executed only once.
- 👉 Constructor can take atleast one argument(atleast self)

- 👉 Constructor is optional and if we are not providing any constructor then python will provide default constructor.

Example:

```
1) def __init__(self,name,rollno,marks):
2)     self.name=name
3)     self.rollno=rollno
4)     self.marks=marks
```

Program to demonistrate constructor will execute only once per object:

```
1) class Test:
2)
3)     def __init__(self):
4)         print("Constructor execution...")
5)
```

```
6) def m1(self):
7)     print("Method execution...")
8)
9) t1=Test()
10) t2=Test()
11) t3=Test()
12) t1.m1()
```

Output

Constructor execution...
Constructor execution...
Constructor execution...
Method execution...

Program:

```
1) class Student:
2)
3)     """ This is student class with required data"""
4)     def __init__(self,x,y,z):
5)         self.name=x
6)         self.rollno=y
7)         self.marks=z
8)
9)     def display(self):
10)        print("Student Name:{}\nRollno:{} \nMarks:{}".format(self.name,self.rollno,self.marks))
11)
12) s1=Student("Prabha",101,80)
13) s1.display()
14) s2=Student("Sunny",102,100)
15) s2.display()
```

Output

Student
Name:ACYTech
Rollno:101
Marks:80
Student Name:Sunny
Rollno:102
Marks:100

Differences between Methods and Constructors:

Method	Constructor
1. Name of method can be any name	1. Constructor name should be always <code>__init__</code>
2. Method will be executed if we call that method	2. Constructor will be executed automatically at the time of object creation.
3. Per object, method can be called any number of times.	3. Per object, Constructor will be executed only once
4. Inside method we can write business logic	4. Inside Constructor we have to declare and initialize instance variables

Types of Variables:

Inside Python class 3 types of variables are allowed.

1. Instance Variables (Object Level Variables)
2. Static Variables (Class Level Variables)
3. Local variables (Method Level Variables)

1. Instance Variables:

If the value of a variable is varied from object to object, then such type of variables are called instance variables.

For every object a separate copy of instance variables will be created.

Where we can declare Instance variables:

1. Inside Constructor by using `self` variable
2. Inside Instance Method by using `self` variable
3. Outside of the class by using object reference variable

1. Inside Constructor by using self variable:

We can declare instance variables inside a constructor by using `self` keyword. Once we creates object, automatically these variables will be added to the object.

Example:

```
1) class Employee:  
2)  
3)     def __init__(self):  
4)         self.eno=100  
5)         self.ename='ACYTech'  
6)         self.esal=10000  
7)  
8) e=Employee()
```

```
9) print(e.__dict__)
```

Output: {'eno': 100, 'ename': 'ACYTech', 'esal': 10000}

2. Inside Instance Method by using self variable:

We can also declare instance variables inside instance method by using self variable. If any instance variable declared inside instance method, that instance variable will be added once we call that method.

Example:

```
1) class Test:  
2)  
3)     def __init__(self):  
4)         self.a=10  
5)         self.b=20  
6)  
7)     def m1(self):  
8)         self.c=30  
9)  
10)    t=Test()  
11)    t.m1()  
12)    print(t.__dict__)
```

Output

{'a': 10, 'b': 20, 'c': 30}

3. Outside of the class by using object reference variable:

We can also add instance variables outside of a class to a particular object.

```
1) class Test:  
2)  
3)     def __init__(self):  
4)         self.a=10  
5)         self.b=20  
6)  
7)     def m1(self):  
8)         self.c=30  
9)  
10)    t=Test()  
11)    t.m1()  
12)    t.d=40  
13)    print(t.__dict__)
```

Output {'a': 10, 'b': 20, 'c': 30, 'd': 40}

How to access Instance variables:

We can access instance variables with in the class by using self variable and outside of the class by using object reference.

```
1) class Test:  
2)  
3)     def __init__(self):  
4)         self.a=10  
5)         self.b=20  
6)  
7)     def display(self):  
8)         print(self.a)  
9)         print(self.b)  
10)  
11) t=Test()  
12) t.display()  
13) print(t.a,t.b)
```

Output

```
10  
20  
10 20
```

How to delete instance variable from the object:

1. Within a class we can delete instance variable as follows

```
del self.variableName
```

2. From outside of class we can delete instance variables as follows

```
del objectreference.variableName
```

Example:

```
1) class Test:  
2)     def __init__(self):  
3)         self.a=10  
4)         self.b=20  
5)         self.c=30  
6)         self.d=40  
7)     def m1(self):  
8)         del self.d  
9)  
10)    t=Test()  
11)    print(t.__dict__)
```

```
12) t.m1()
13) print(t.__dict__)
14) del t.c
15) print(t.__dict__)
```

Output

```
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30}
{'a': 10, 'b': 20}
```

Note: The instance variables which are deleted from one object, will not be deleted from other objects.

Example:

```
1) class Test:
2)     def __init__(self):
3)         self.a=10
4)         self.b=20
5)         self.c=30
6)         self.d=40
7)
8)
9) t1=Test()
10) t2=Test()
11) del t1.a
12) print(t1.__dict__)
13) print(t2.__dict__)
```

Output

```
{'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```

If we change the values of instance variables of one object then those changes won't be reflected to the remaining objects, because for every object we are separate copy of instance variables are available.

Example:

```
1) class Test:
2)     def __init__(self):
3)         self.a=10
4)         self.b=20
5)
6) t1=Test()
7) t1.a=888
8) t1.b=999
9) t2=Test()
10) print('t1:',t1.a,t1.b)
```

```
|11) print('t2:',t2.a,t2.b)
```

Output

t1: 888 999
t2: 10 20

1. Static variables:

If the value of a variable is not varied from object to object, such type of variables we have to declare with in the class directly but outside of methods. Such type of variables are called Static variables.

For total class only one copy of static variable will be created and shared by all objects of that class.

We can access static variables either by class name or by object reference. But recommended to use class name.

Instance Variable vs Static Variable:

Note: In the case of instance variables for every object a separate copy will be created, but in the case of static variables for total class only one copy will be created and shared by every object of that class.

```
1) class Test:  
2)     x=10  
3)     def __init__(self):  
4)         self.y=20  
5)  
6) t1=Test()  
7) t2=Test()  
8) print('t1:',t1.x,t1.y)  
9) print('t2:',t2.x,t2.y)  
10) Test.x=888  
11) t1.y=999  
12) print('t1:',t1.x,t1.y)  
13) print('t2:',t2.x,t2.y)
```

Output

t1: 10 20
t2: 10 20
t1: 888 999
t2: 888 20

Various places to declare static variables:

1. In general we can declare within the class directly but from out side of any method
2. Inside constructor by using class name
3. Inside instance method by using class name
4. Inside classmethod by using either class name or cls variable
5. Inside static method by using class name

```
1) class Test:  
2)     a=10  
3)     def __init__(self):  
4)         Test.b=20  
5)     def m1(self):  
6)         Test.c=30  
7)     @classmethod  
8)     def m2(cls):  
9)         cls.d1=40  
10)    Test.d2=400  
11)    @staticmethod  
12)    def m3():  
13)        Test.e=50  
14) print(Test.__dict__)  
15) t=Test()  
16) print(Test.__dict__)  
17) t.m1()  
18) print(Test.__dict__)  
19) Test.m2()  
20) print(Test.__dict__)  
21) Test.m3()  
22) print(Test.__dict__)  
23) Test.f=60  
24) print(Test.__dict__)
```

How to access static variables:

1. inside constructor: by using either self or classname
2. inside instance method: by using either self or classname
3. inside class method: by using either cls variable or classname
4. inside static method: by using classname
5. From outside of class: by using either object reference or classnmae

```
1) class Test:  
2)     a=10  
3)     def __init__(self):  
4)         print(self.a)  
5)         print(Test.a)  
6)     def m1(self):  
7)         print(self.a)
```

```
8)     print(Test.a)
9)     @classmethod
10)    def m2(cls):
11)        print(cls.a)
12)        print(Test.a)
13)    @staticmethod
14)    def m3():
15)        print(Test.a)
16) t=Test()
17) print(Test.a)
18) print(t.a)
19) t.m1()
20) t.m2()
21) t.m3()
```

Where we can modify the value of static variable:

Anywhere either with in the class or outside of class we can modify by using classname.
But inside class method, by using cls variable.

Example:

```
1) class Test:
2)     a=777
3)     @classmethod
4)     def m1(cls):
5)         cls.a=888
6)     @staticmethod
7)     def m2():
8)         Test.a=999
9)     print(Test.a)
10)    Test.m1()
11)    print(Test.a)
12)    Test.m2()
13)    print(Test.a)
```

Output

777
888
999

If we change the value of static variable by using either self or object reference variable:

If we change the value of static variable by using either self or object reference variable, then the value of static variable won't be changed, just a new instance variable with that name will be added to that particular object.

Example 1:

```
1) class Test:  
2)     a=10  
3)     def m1(self):  
4)         self.a=888  
5) t1=Test()  
6) t1.m1()  
7) print(Test.a)  
8) print(t1.a)
```

Output

10
888

Example:

```
1) class Test:  
2)     x=10  
3)     def __init__(self):  
4)         self.y=20  
5)  
6) t1=Test()  
7) t2=Test()  
8) print('t1:',t1.x,t1.y)  
9) print('t2:',t2.x,t2.y)  
10) t1.x=888  
11) t1.y=999  
12) print('t1:',t1.x,t1.y)  
13) print('t2:',t2.x,t2.y)
```

Output

t1: 10 20
t2: 10 20
t1: 888 999
t2: 10 20

Example:

```
1) class Test:  
2)     a=10  
3)     def __init__(self):  
4)         self.b=20  
5) t1=Test()  
6) t2=Test()  
7) Test.a=888  
8) t1.b=999  
9) print(t1.a,t1.b)  
10) print(t2.a,t2.b)
```

Output

```
888 999  
888 20
```

```
1) class Test:  
2)     a=10  
3)     def __init__(self):  
4)         self.b=20  
5)     def m1(self):  
6)         self.a=888  
7)         self.b=999  
8)  
9) t1=Test()  
10) t2=Test()  
11) t1.m1()  
12) print(t1.a,t1.b)  
13) print(t2.a,t2.b)
```

Output

```
888 999  
10 20
```

Example:

```
1) class Test:  
2)     a=10  
3)     def __init__(self):  
4)         self.b=20  
5)     @classmethod  
6)     def m1(cls):  
7)         cls.a=888  
8)         cls.b=999  
9)  
10) t1=Test()  
11) t2=Test()
```

```
12) t1.m1()
13) print(t1.a,t1.b)
14) print(t2.a,t2.b)
15) print(Test.a,Test.b)
```

Output

888 20
888 20
888 999

How to delete static variables of a class:

We can delete static variables from anywhere by using the following syntax

```
del classname.variablename
```

But inside classmethod we can also use cls variable

```
del cls.variablename
```

```
1) class Test:
2)     a=10
3)     @classmethod
4)     def m1(cls):
5)         del cls.a
6)     Test.m1()
7)     print(Test.__dict__)
```

Example:

```
1) class Test:
2)     a=10
3)     def __init__(self):
4)         Test.b=20
5)     del Test.a
6)     def m1(self):
7)         Test.c=30
8)     del Test.b
9)     @classmethod
10)    def m2(cls):
11)        cls.d=40
12)    del Test.c
13)    @staticmethod
14)    def m3():
15)        Test.e=50
16)    del Test.d
17)    print(Test.__dict__)
18) t=Test()
```

```
19) print(Test.__dict__)
20) t.m1()
21) print(Test.__dict__)
22) Test.m2()
23) print(Test.__dict__)
24) Test.m3()
25) print(Test.__dict__)
26) Test.f=60
27) print(Test.__dict__)
28) del Test.e
29) print(Test.__dict__)
```

Note: By using object reference variable/self we can read static variables, but we cannot modify or delete.

If we are trying to modify, then a new instance variable will be added to that particular object.
t1.a = 70

If we are trying to delete then we will get error.

Example:

```
1) class Test:
2)     a=10
3)
4) t1=Test()
5) del t1.a    ==>AttributeError: a
```

We can modify or delete static variables only by using classname or cls variable.

```
1) import sys
2) class Customer:
3)     """ Customer class with bank operations.. """
4)     bankname='ACYTECHBANK'
5)     def __init__(self,name,balance=0.0):
6)         self.name=name
7)         self.balance=balance
8)     def deposit(self,amt):
9)         self.balance=self.balance+amt
10)        print('Balance after deposit:',self.balance)
11)    def withdraw(self,amt):
12)        if amt>self.balance:
13)            print('Insufficient Funds..cannot perform this operation')
14)            sys.exit()
15)        self.balance=self.balance-amt
16)        print('Balance after withdraw:',self.balance)
17)
18) print('Welcome to',Customer.bankname)
```

```
19) name=input('Enter Your Name: ')
20) c=Customer(name)
21) while True:
22)     print('d-Deposit \nw-Withdraw \ne-exit')
23)     option=input('Choose your option:')
24)     if option=='d' or option=='D':
25)         amt=float(input('Enter amount:'))
26)         c.deposit(amt)
27)     elif option=='w' or option=='W':
28)         amt=float(input('Enter amount:'))
29)         c.withdraw(amt)
30)     elif option=='e' or option=='E':
31)         print('Thanks for Banking')
32)         sys.exit()
33)     else:
34)         print('Invalid option..Plz choose valid option')
```

output:

```
D:\ACYTech_classes>py
test.pyWelcome to
ACYTECHBANK Enter Your
Name:ACYTech
d-Deposit
w-Withdraw
e-exit
Choose your option:d
Enter amount:10000
Balance after deposit: 10000.0
d-Deposit
w-Withdraw
e-exit
Choose your option:d
Enter amount:20000
Balance after deposit: 30000.0
d-Deposit
w-Withdraw
e-exit
Choose your option:w
Enter amount:2000
Balance after withdraw: 28000.0
d-Deposit
w-Withdraw
e-exit
Choose your option:r
Invalid option..Plz choose valid option
d-Deposit
w-Withdraw
e-exit
Choose your option:e
Thanks for Banking
```

Local variables:

Sometimes to meet temporary requirements of programmer, we can declare variables inside a method directly, such type of variables are called local variable or temporary variables.

Local variables will be created at the time of method execution and destroyed once method completes.

Local variables of a method cannot be accessed from outside of method.

Example:

```
1) class Test:  
2)     def m1(self):  
3)         a=1000  
4)         print(a)  
5)     def m2(self):  
6)         b=2000  
7)         print(b)  
8) t=Test()  
9) t.m1()  
10) t.m2()
```

Output

1000

2000

Example 2:

```
1) class Test:  
2)     def m1(self):  
3)         a=1000  
4)         print(a)  
5)     def m2(self):  
6)         b=2000  
7)         print(a) #NameError: name 'a' is not defined  
8)         print(b)  
9) t=Test()  
10) t.m1()  
11) t.m2()
```

Types of Methods:

Inside Python class 3 types of methods are allowed

1. Instance Methods
2. Class Methods
3. Static Methods

1. Instance Methods:

Inside method implementation if we are using instance variables then such type of methods are called instance methods.

Inside instance method declaration, we have to pass self variable.

```
def m1(self):
```

By using self variable inside method we can able to access instance variables.

Within the class we can call instance method by using self variable and from outside of the class we can call by using object reference.

```
1) class Student:  
2)     def __init__(self,name,marks):  
3)         self.name=name  
4)         self.marks=marks  
5)     def display(self):  
6)         print('Hi',self.name)  
7)         print('Your Marks are:',self.marks)  
8)     def grade(self):  
9)         if self.marks>=60:  
10)             print('You got First Grade')  
11)         elif self.marks>=50:  
12)             print('You got Second Grade')  
13)         elif self.marks>=35:  
14)             print('You got Third Grade')  
15)         else:  
16)             print('You are Failed')  
17) n=int(input('Enter number of students:'))  
18) for i in range(n):  
19)     name=input('Enter Name:')  
20)     marks=int(input('Enter Marks:'))  
21)     s= Student(name,marks)  
22)     s.display()  
23)     s.grade()  
24)     print()
```

output:

```
D:\ACYTech_classes>py  
test.py Enter number of  
students:2          Enter  
Name:ACYTech  
Enter Marks:90  
Hi ACYTech  
Your Marks are: 90  
You got First Grade
```

```
Enter Name:Ravi
```

```
Enter Marks:12
```

```
Hi Ravi
```

```
Your Marks are: 12
```

```
You are Failed
```

Setter and Getter Methods:

We can set and get the values of instance variables by using getter and setter methods.

Setter Method:

setter methods can be used to set values to the instance variables. setter methods also known as mutator methods.

syntax:

```
def setVariable(self,variable):  
    self.variable=variable
```

Example:

```
def setName(self,name):  
    self.name=name
```

Getter Method:

Getter methods can be used to get values of the instance variables. Getter methods also known as accessor methods.

syntax:

```
def getVariable(self):  
    return self.variable
```

Example:

```
def getName(self):  
    return self.name
```

Demo Program:

```
1) class Student:  
2)     def setName(self,name):  
3)         self.name=name  
4)  
5)     def getName(self):  
6)         return self.name  
7)  
8)     def setMarks(self,marks):  
9)         self.marks=marks  
10)  
11)    def getMarks(self):  
12)        return self.marks  
13)  
14) n=int(input('Enter number of students:'))  
15) for i in range(n):  
16)     s=Student()  
17)     name=input('Enter Name:')  
18)     s.setName(name)  
19)     marks=int(input('Enter Marks:'))  
20)     s.setMarks(marks)  
21)  
22)     print('Hi',s.getName())  
23)     print('Your Marks are:',s.getMarks())  
24)     print()
```

output:

D:\python_classes>py test.py

Enter number of students:2

Enter Name:ACYTech

Enter Marks:100

Hi ACYTech

Your Marks are: 100

Enter Name:Ravi

Enter Marks:80

Hi Ravi

Your Marks are: 80

2. Class Methods:

Inside method implementation if we are using only class variables (static variables), then such type of methods we should declare as class method.

We can declare class method explicitly by using `@classmethod` decorator.

For class method we should provide `cls` variable at the time of declaration

We can call classmethod by using classname or object reference variable.

Demo Program:

```
1) class Animal:  
2)     legs=4  
3)     @classmethod  
4)     def walk(cls,name):  
5)         print('{} walks with {} legs...'.format(name,cls.legs))  
6) Animal.walk('Dog')  
7) Animal.walk('Cat')
```

Output

D:\python_classes>py test.py

Dog walks with 4 legs...

Cat walks with 4 legs...

Program to track the number of objects created for a class:

```
1) class Test:  
2)     count=0  
3)     def __init__(self):  
4)         Test.count =Test.count+1  
5)     @classmethod  
6)     def noOfObjects(cls):  
7)         print('The number of objects created for test class:',cls.count)  
8)  
9) t1=Test()  
10) t2=Test()  
11) Test.noOfObjects()  
12) t3=Test()  
13) t4=Test()  
14) t5=Test()  
15) Test.noOfObjects()
```

3. Static Methods:

In general these methods are general utility methods.

Inside these methods we won't use any instance or class variables.

Here we won't provide self or cls arguments at the time of declaration.

We can declare static method explicitly by using @staticmethod decorator

We can access static methods by using classname or object reference

```
1) class ACYTechMath:  
2)  
3)     @staticmethod  
4)     def add(x,y):
```

```
5)     print('The Sum:',x+y)
6)
7) @staticmethod
8) def product(x,y):
9)     print('The Product:',x*y)
10)
11) @staticmethod
12) def average(x,y):
13)     print('The average:',(x+y)/2)
14)
15) PrabhaMath.add(10,20)
16) PrabhaMath.product(10,20)
17)
```

Output

The Sum: 30
The Product: 200
The average: 15.0

Note: In general we can use only instance and static methods. Inside static method we can access class level variables by using class name.

class methods are most rarely used methods in python.

Passing members of one class to another class:

We can access members of one class inside another class.

```
1) class Employee:
2)     def __init__(self,eno,ename,esal):
3)         self.eno=eno
4)         self.ename=ename
5)         self.esal=esal
6)     def display(self):
7)         print('Employee Number:',self.eno)
8)         print('Employee Name:',self.ename)
9)         print('Employee Salary:',self.esal)
10)    class Test:
11)        def modify(emp):
12)            emp.esal=emp.esal+10000
13)            emp.display()
14)    e=Employee(100,'Prabha',10000)
15)    Test.modify(e)
```

Output

D:\python_classes>py test.py
Employee Number: 100
Employee Name: ACYTech

Employee Salary: 20000

In the above application, Employee class members are available to Test class.

Inner classes:

Sometimes we can declare a class inside another class, such type of classes are called inner classes.

Without existing one type of object if there is no chance of existing another type of object, then we should go for inner classes.

Example: Without existing Car object there is no chance of existing Engine object. Hence Engine class should be part of Car class.

class Car:

.....

 class Engine:

.....

Example: Without existing university object there is no chance of existing Department object

class University:

.....

 class Department:

.....

eg3:

Without existing Human there is no chance of existin Head. Hence Head should be part of Human.

class Human:

 class Head:

Note: Without existing outer class object there is no chance of existing inner class object. Hence inner class object is always associated with outer class object.

Demo Program-1:

```
1) class Outer:  
2)     def __init__(self):  
3)         print("outer class object creation")  
4)     class Inner:  
5)         def __init__(self):  
6)             print("inner class object creation")  
7)         def m1(self):  
8)             print("inner class method")  
9)     o=Outer()  
10)    i=o.Inner()  
11)    i.m1()
```

Output

```
outer class object creation
inner class object creation
inner class method
```

Note: The following are various possible syntaxes for calling inner class method

1.

```
o=Outer()
i=o.Inner()
i.m1()
```

2.

```
i=Outer().Inner()
i.m1()
```

3. Outer().Inner().m1()

Demo Program-2:

```
1) class Person:
2)     def __init__(self):
3)         self.name='prabh'
4)         self.db=self.Dob()
5)     def display(self):
6)         print('Name:',self.name)
7)     class Dob:
8)         def __init__(self):
9)             self.dd=10
10)            self.mm=5
11)            self.yy=1947
12)        def display(self):
13)            print('Dob={}/{}/{}'.format(self.dd,self.mm,self.yy))
14) p=Person()
15) p.display()
16) x=p.db
17) x.display()
```

Output

```
Name: ACYTech
Dob=10/5/1947
```

Demo Program-3:

Inside a class we can declare any number of inner classes.

```
1) class Human:
2)
3)     def __init__(self):
```

```
4)     self.name = 'Sunny'
5)     self.head = self.Head()
6)     self.brain = self.Brain()
7)     def display(self):
8)         print("Hello..",self.name)
9)
10)    class Head:
11)        def talk(self):
12)            print('Talking...')
13)
14)    class Brain:
15)        def think(self):
16)            print('Thinking...')
17)
18) h=Human()
19) h.display()
20) h.head.talk()
21) h.brain.think()
```

Output

Hello.. Sunny
Talking...
Thinking...

Garbage Collection:

In old languages like C++, programmer is responsible for both creation and destruction of objects. Usually programmer taking very much care while creating object, but neglecting destruction of useless objects. Because of his negligence, total memory can be filled with useless objects which creates memory problems and total application will be down with Out of memory error.

But in Python, We have some assistant which is always running in the background to destroy useless objects. Because this assistant the chance of failing Python program with memory problems is very less. This assistant is nothing but Garbage Collector.

Hence the main objective of Garbage Collector is to destroy useless objects.

If an object does not have any reference variable then that object eligible for Garbage Collection.

How to enable and disable Garbage Collector in our program:

By default Garbage collector is enabled, but we can disable based on our requirement. In this context we can use the following functions of gc module.

1. gc.isenabled()

Returns True if GC enabled

2. gc.disable()

To disable GC explicitly

3. gc.enable()

To enable GC explicitly

Example:

```
1) import gc  
2) print(gc.isenabled())  
3) gc.disable()  
4) print(gc.isenabled())  
5) gc.enable()  
6) print(gc.isenabled())
```

Output

True

False

True

Destructors:

Destructor is a special method and the name should be `__del__`

Just before destroying an object Garbage Collector always calls destructor to perform clean up activities (Resource deallocation activities like close database connection etc).

Once destructor execution completed then Garbage Collector automatically destroys that object.

Note: The job of destructor is not to destroy object and it is just to perform clean up activities.

Example:

```
1) import time  
2) class Test:  
3)     def __init__(self):  
4)         print("Object Initialization...")  
5)     def __del__(self):  
6)         print("Fulfilling Last Wish and performing clean up activities...")  
7)  
8) t1=Test()  
9) t1=None  
10) time.sleep(5)  
11) print("End of application")
```

Output

Object Initialization...

Fulfilling Last Wish and performing clean up activities...

End of application

Note:

If the object does not contain any reference variable then only it is eligible for GC. ie if the reference count is zero then only object eligible for GC

Example:

```
1) import time
2) class Test:
3)     def __init__(self):
4)         print("Constructor Execution...")
5)     def __del__(self):
6)         print("Destructor Execution...")
7)
8) t1=Test()
9) t2=t1
10) t3=t2
11) del t1
12) time.sleep(5)
13) print("object not yet destroyed after deleting t1")
14) del t2
15) time.sleep(5)
16) print("object not yet destroyed even after deleting t2")
17) print("I am trying to delete last reference variable...")
18) del t3
```

Example:

```
1) import time
2) class Test:
3)     def __init__(self):
4)         print("Constructor Execution...")
5)     def __del__(self):
6)         print("Destructor Execution...")
7)
8) list=[Test(),Test(),Test()]
9) del list
10) time.sleep(5)
11) print("End of application")
```

Output

Constructor Execution...
Constructor Execution...
Constructor Execution...
Destructor Execution...
Destructor Execution...
Destructor Execution...
End of application

How to find the number of references of an object:

sys module contains getrefcount() function for this purpose.

```
sys.getrefcount(objectreference)
```

Example:

```
1) import sys  
2) class Test:  
3)     pass  
4) t1=Test()  
5) t2=t1  
6) t3=t1  
7) t4=t1  
8) print(sys.getrefcount(t1))
```

Output 5

Note: For every object, Python internally maintains one default reference variable self.

Polymorphism

Poly means many. Morphs means forms.

Polymorphism means 'Many Forms'.

Eg1: Yourself is best example of polymorphism. In front of Your parents You will have one type of behaviour and with friends another type of behaviour. Same person but different behaviours at different places, which is nothing but polymorphism.

Eg2: + operator acts as concatenation and arithmetic addition

Eg3: * operator acts as multiplication and repetition operator

Eg4: The Same method with different implementations in Parent class and child classes. (overriding)

Related to polymorphism the following 4 topics are important

1. Duck Typing Philosophy of Python

2. Overloading

1. Operator Overloading
2. Method Overloading
3. Constructor Overloading

3. Overriding

1. Method overriding
2. constructor overriding

1. Duck Typing Philosophy of Python:

In Python we cannot specify the type explicitly. Based on provided value at runtime the type will be considered automatically. Hence Python is considered as Dynamically Typed Programming Language.

```
def f1(obj):  
    obj.talk()
```

What is the type of obj? We cannot decide at the beginning. At runtime we can pass any type. Then how we can decide the type?

At runtime if 'it walks like a duck and talks like a duck, it must be duck'. Python follows this principle. This is called Duck Typing Philosophy of Python.

Demo Program:

```
1) class Duck:  
2)     def talk(self):  
3)         print('Quack.. Quack..')  
4)  
5) class Dog:  
6)     def talk(self):  
7)         print('Bow Bow..')  
8)  
9) class Cat:  
10)    def talk(self):  
11)        print('Moew Moew ..')  
12)  
13) class Goat:  
14)    def talk(self):  
15)        print('Myaah Myaah ..')  
16)  
17) def f1(obj):  
18)     obj.talk()  
19)  
20) l=[Duck(),Cat(),Dog(),Goat()]  
21) for obj in l:  
22)     f1(obj)
```

Output:

Quack.. Quack..
Moew Moew ..
Bow Bow..
Myaah Myaah ..

The problem in this approach is if obj does not contain talk() method then we will get
AttributeError

Eg:

```
1) class Duck:  
2)     def talk(self):  
3)         print('Quack.. Quack..')  
4)  
5) class Dog:  
6)     def bark(self):  
7)         print('Bow Bow..')  
8) def f1(obj):  
9)     obj.talk()  
10)  
11) d=Duck()  
12) f1(d)  
13)
```

```
14) d=Dog()  
15) f1(d)
```

Output:

```
D:\ACYTech_classes>py  
test.pyQuack.. Quack..  
Traceback (most recent call last):  
  File "test.py", line 22, in <module>  
    f1(d)  
  File "test.py", line 13, in f1  
    obj.talk()  
AttributeError: 'Dog' object has no attribute 'talk'
```

But we can solve this problem by using `hasattr()` function.

```
hasattr(obj,'attributename')  
attributename can be method name or variable name
```

Demo Program with `hasattr()` function:

```
1) class Duck:  
2)   def talk(self):  
3)     print('Quack.. Quack..')  
4)  
5) class Human:  
6)   def talk(self):  
7)     print('Hello Hi...')  
8)  
9) class Dog:  
10)  def bark(self):  
11)    print('Bow Bow..')  
12)  
13) def f1(obj):  
14)   if hasattr(obj,'talk'):  
15)     obj.talk()  
16)   elif hasattr(obj,'bark'):  
17)     obj.bark()  
18)  
19) d=Duck()  
20) f1(d)  
21)  
22) h=Human()  
23) f1(h)  
24)  
25) d=Dog()  
26) f1(d)  
27) Myaah Myaah Myaah...
```

Overloading:

We can use same operator or methods for different purposes.

Eg1: + operator can be used for Arithmetic addition and String concatenation

```
print(10+20)#30  
print('ACYTech'+'soft')#
```

Eg2: * operator can be used for multiplication and string repetition purposes.

```
print(10*20)#200  
print('ACYTech'*3)#ACYTechACYTechACYTech
```

Eg3: We can use deposit() method to deposit cash or cheque or dd

```
deposit(cash)  
deposit(cheque)  
deposit(dd)
```

There are 3 types of overloading

1. Operator Overloading
2. Method Overloading
3. Constructor Overloading

1. Operator Overloading:

We can use the same operator for multiple purposes, which is nothing but operator overloading.

Python supports operator overloading.

Eg1: + operator can be used for Arithmetic addition and String concatenation

```
print(10+20)#30  
print('ACYTech'+'soft')#
```

Eg2: * operator can be used for multiplication and string repetition purposes.

```
print(10*20)#200  
print('ACYTech'*3)#ACYTechACYTechACYTech
```

Demo program to use + operator for our class objects:

```
1) class Book:  
2)     def __init__(self, pages):  
3)         self.pages = pages  
4)  
5) b1 = Book(100)  
6) b2 = Book(200)  
7) print(b1+b2)
```

```
D:\ACYTech_classes>py test.py
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print(b1+b2)
TypeError: unsupported operand type(s) for +: 'Book' and 'Book'
```

We can overload + operator to work with Book objects also. i.e Python supports Operator Overloading.

For every operator Magic Methods are available. To overload any operator we have to override that Method in our class.

Internally + operator is implemented by using `__add__()` method. This method is called magic method for + operator. We have to override this method in our class.

Demo program to overload + operator for our Book class objects:

```
1) class Book:
2)     def __init__(self, pages):
3)         self.pages = pages
4)
5)     def __add__(self, other):
6)         return self.pages + other.pages
7)
8) b1 = Book(100)
9) b2 = Book(200)
10) print('The Total Number of Pages:', b1 + b2)
```

Output: The Total Number of Pages: 300

The following is the list of operators and corresponding magic methods.

+ --->	object.__add__(self,other)
- --->	object.__sub__(self,other)
*	---> object.__mul__(self,other)
/ --->	object.__div__(self,other)
// --->	object.__floordiv__(self,other)
% --->	object.__mod__(self,other)
** --->	object.__pow__(self,other)
+= --->	object.__iadd__(self,other)
-= --->	object.__isub__(self,other)
*= --->	object.__imul__(self,other)
/= --->	object.__idiv__(self,other)
//= --->	object.__ifloordiv__(self,other)
%= --->	object.__imod__(self,other)
**= --->	object.__ipow__(self,other)
< --->	object.__lt__(self,other)
<= --->	object.__le__(self,other)
> --->	object.__gt__(self,other)
>= --->	object.__ge__(self,other)

```
== ---> object.__eq__(self,other)
!= ---> object.__ne__(self,other)
```

Overloading > and <= operators for Student class objects:

```
1) class Student:
2)     def __init__(self,name,marks):
3)         self.name=name
4)         self.marks=marks
5)     def __gt__(self,other):
6)         return self.marks>other.marks
7)     def __le__(self,other):
8)         return self.marks<=other.marks
9)
10)
11) print("10>20 =",10>20)
12) s1=Student("Prabha",100)
13) s2=Student("Ravi",200)
14) print("s1>s2 =",s1>s2)
15) print("s1<s2 =",s1<s2)
16) print("s1<=s2 =",s1<=s2)
17) print("s1>=s2 =",s1>=s2)
```

Output:

```
10>20 = False
s1>s2= False
s1<s2= True
s1<=s2= True
s1>=s2= False
```

Program to overload multiplication operator to work on Employee objects:

```
1) class Employee:
2)     def __init__(self,name,salary):
3)         self.name=name
4)         self.salary=salary
5)     def __mul__(self,other):
6)         return self.salary*other.days
7)
8) class TimeSheet:
9)     def __init__(self,name,days):
10)        self.name=name
11)        self.days=days
12)
13) e=Employee('Prabha',500)
14) t=TimeSheet('Prabha',25)
15) print('This Month Salary:',e*t)
```

Output: This Month Salary: 12500

2. Method Overloading:

If 2 methods having same name but different type of arguments then those methods are said to be overloaded methods.

Eg: m1(int a)
m1(double d)

But in Python Method overloading is not possible.

If we are trying to declare multiple methods with same name and different number of arguments then Python will always consider only last method.

Demo Program:

```
1) class Test:  
2)     def m1(self):  
3)         print('no-arg method')  
4)     def m1(self,a):  
5)         print('one-arg method')  
6)     def m1(self,a,b):  
7)         print('two-arg method')  
8)  
9) t=Test()  
10) #t.m1()  
11) #t.m1(10)  
12) t.m1(10,20)
```

Output: two-arg method

In the above program python will consider only last method.

How we can handle overloaded method requirements in Python:

Most of the times, if method with variable number of arguments required then we can handle with default arguments or with variable number of argument methods.

Demo Program with Default Arguments:

```
1) class Test:  
2)     def sum(self,a=None,b=None,c=None):  
3)         if a!=None and b!= None and c!= None:  
4)             print('The Sum of 3 Numbers:',a+b+c)  
5)         elif a!=None and b!= None:  
6)             print('The Sum of 2 Numbers:',a+b)  
7)         else:  
8)             print('Please provide 2 or 3 arguments')  
9)  
10) t=Test()
```

```
11) t.sum(10,20)
12) t.sum(10,20,30)
13) t.sum(10)
```

Output:

The Sum of 2 Numbers: 30
The Sum of 3 Numbers: 60
Please provide 2 or 3 arguments

Demo Program with Variable Number of Arguments:

```
1) class Test:
2)     def sum(self,*a):
3)         total=0
4)         for x in a:
5)             total=total+x
6)         print('The Sum:',total)
7)
8)
9) t=Test()
10) t.sum(10,20)
11) t.sum(10,20,30)
12) t.sum(10)
13) t.sum()
```

3. Constructor Overloading:

Constructor overloading is not possible in Python.

If we define multiple constructors then the last constructor will be considered.

```
1) class Test:
2)     def __init__(self):
3)         print('No-Arg Constructor')
4)
5)     def __init__(self,a):
6)         print('One-Ar constructor')
7)
8)     def __init__(self,a,b):
9)         print('Two-Ar constructor')
10) #t1=Test()
11) #t1=Test(10)
12) t1=Test(10,20)
```

Output: Two-Ar constructor

In the above program only Two-Arg Constructor is available.

But based on our requirement we can declare constructor with default arguments and variable number of arguments.

Constructor with Default Arguments:

```
1) class Test:  
2)     def __init__(self,a=None,b=None,c=None):  
3)         print('Constructor with 0|1|2|3 number of arguments')  
4)  
5) t1=Test()  
6) t2=Test(10)  
7) t3=Test(10,20)  
8) t4=Test(10,20,30)
```

Output:

Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments
Constructor with 0|1|2|3 number of arguments

Constructor with Variable Number of Arguments:

```
1) class Test:  
2)     def __init__(self,*a):  
3)         print('Constructor with variable number of arguments')  
4)  
5) t1=Test()  
6) t2=Test(10)  
7) t3=Test(10,20)  
8) t4=Test(10,20,30)  
9) t5=Test(10,20,30,40,50,60)
```

Output:

Constructor with variable number of arguments
Constructor with variable number of arguments

Method overriding:

Whatever members available in the parent class are by default available to the child class through inheritance. If the child class does not satisfy with parent class implementation then child class is allowed to redefine that method in the child class based on its requirement. This concept is called overriding.

Overriding concept applicable for both methods and constructors.

Demo Program for Method overriding:

```
1) class P:  
2)     def property(self):  
3)         print('Gold+Land+Cash+Power')  
4)     def marry(self):  
5)         print('Appalamma')  
6) class C(P):  
7)     def marry(self):  
8)         print('Katrina Kaif')  
9)  
10) c=C()  
11) c.property()  
12) c.marry()
```

Output:

Gold+Land+Cash+Power

Katrina Kaif

From Overriding method of child class, we can call parent class method also by using super() method.

```
1) class P:  
2)     def property(self):  
3)         print('Gold+Land+Cash+Power')  
4)     def marry(self):  
5)         print('Appalamma')  
6) class C(P):  
7)     def marry(self):  
8)         super().marry()  
9)         print('Katrina Kaif')  
10)  
11) c=C()  
12) c.property()  
13) c.marry()
```

Output:

Gold+Land+Cash+Power

Appalamma

Katrina Kaif

Demo Program for Constructor overriding:

```
1) class P:  
2)     def __init__(self):  
3)         print('Parent Constructor')  
4)  
5) class C(P):  
6)     def __init__(self):  
7)         print('Child Constructor')  
8)  
9) c=C()
```

Output: Child Constructor

In the above example, if child class does not contain constructor then parent class constructor will be executed

From child class constructor we can call parent class constructor by using super() method.

Demo Program to call Parent class constructor by using super():

```
1) class Person:  
2)     def __init__(self,name,age):  
3)         self.name=name  
4)         self.age=age  
5)  
6) class Employee(Person):  
7)     def __init__(self,name,age,eno,esal):  
8)         super().__init__(name,age)  
9)         self.eno=eno  
10)        self.esal=esal  
11)  
12)    def display(self):  
13)        print('Employee Name:',self.name)  
14)        print('Employee Age:',self.age)  
15)        print('Employee Number:',self.eno)  
16)        print('Employee Salary:',self.esal)  
17)  
18) e1=Employee('Prabha',48,872425,26000)  
19) e1.display()  
20) e2=Employee('Sunny',39,872426,36000)  
21) e2.display()
```

Output:

Employee Name:

ACYTechEmployee Age:

48

Employee Number: 872425

Employee Salary: 26000

Employee Name: Sunny

Employee Age: 39

Employee Number: 872426

Employee Salary: 36000

Multi Threading

Multi Tasking:

Executing several tasks simultaneously is the concept of multitasking.

There are 2 types of Multi Tasking

1. Process based Multi Tasking
2. Thread based Multi Tasking

1. Process based Multi Tasking:

Executing several tasks simultaneously where each task is a separate independent process is called process based multi tasking.

Eg: while typing python program in the editor we can listen mp3 audio songs from the same system. At the same time we can download a file from the internet. All these tasks are executing simultaneously and independent of each other. Hence it is process based multi tasking.

This type of multi tasking is best suitable at operating system level.

2. Thread based MultiTasking:

Executing several tasks simultaneously where each task is a separate independent part of the same program, is called Thread based multi tasking, and each independent part is called a Thread.

This type of multi tasking is best suitable at programmatic level.

Note: Whether it is process based or thread based, the main advantage of multi tasking is to improve performance of the system by reducing response time.

The main important application areas of multi threading are:

1. To implement Multimedia graphics
 2. To develop animations
 3. To develop video games
 4. To develop web and application servers
- etc...

Note: Where ever a group of independent jobs are available, then it is highly recommended to execute simultaneously instead of executing one by one. For such type of cases we should go for Multi Threading.

Python provides one inbuilt module "threading" to provide support for developing threads. Hence developing multi threaded Programs is very easy in python.

Every Python Program by default contains one thread which is nothing but MainThread.

Q.Program to print name of current executing thread:

```
1) import threading  
2) print("Current Executing Thread:",threading.current_thread().getName())
```

o/p: Current Executing Thread: MainThread

Note: threading module contains function `current_thread()` which returns the current executing Thread object. On this object if we call `getName()` method then we will get current executing thread name.

The ways of Creating Thread in Python:

We can create a thread in Python by using 3 ways

1. Creating a Thread without using any class
2. Creating a Thread by extending Thread class
3. Creating a Thread without extending Thread class

1. Creating a Thread without using any class:

```
1) from threading import *  
2) def display():  
3)     for i in range(1,11):  
4)         print("Child Thread")  
5) t=Thread(target=display) #creating Thread object  
6) t.start() #starting of Thread  
7) for i in range(1,11):  
8)     print("Main Thread")
```

If multiple threads present in our program, then we cannot expect execution order and hence we cannot expect exact output for the multi threaded programs. B'z of this we cannot provide exact output for the above program. It is varied from machine to machine and run to run.

Note: Thread is a pre defined class present in threading module which can be used to create our own Threads.

2. Creating a Thread by extending Thread class

We have to create child class for Thread class. In that child class we have to override `run()` method with our required job. Whenever we call `start()` method then automatically `run()` method will be executed and performs our job.

```
1) from threading import *  
2) class MyThread(Thread):
```

```
3) def run(self):  
4)     for i in range(10):  
5)         print("Child Thread-1")  
6) t=MyThread()  
7) t.start()  
8) for i in range(10):  
9)     print("Main Thread-1")
```

3. Creating a Thread without extending Thread class:

```
1) from threading import *  
2) class Test:  
3)     def display(self):  
4)         for i in range(10):  
5)             print("Child Thread-2")  
6) obj=Test()  
7) t=Thread(target=obj.display)  
8) t.start()  
9) for i in range(10):  
10)    print("Main Thread-2")
```

Without multi threading:

```
1) from threading import *  
2) import time  
3) def doubles(numbers):  
4)     for n in numbers:  
5)         time.sleep(1)  
6)         print("Double:",2*n)  
7) def squares(numbers):  
8)     for n in numbers:  
9)         time.sleep(1)  
10)        print("Square:",n*n)  
11) numbers=[1,2,3,4,5,6]  
12) begintime=time.time()  
13) doubles(numbers)  
14) squares(numbers)  
15) print("The total time taken:",time.time()-begintime)
```

With multithreading:

```
1) from threading import *  
2) import time  
3) def doubles(numbers):  
4)     for n in numbers:  
5)         time.sleep(1)  
6)         print("Double:",2*n)  
7) def squares(numbers):  
8)     for n in numbers:
```

```
9)     time.sleep(1)
10)    print("Square:",n*n)
11)
12) numbers=[1,2,3,4,5,6]
13) begintime=time.time()
14) t1=Thread(target=doubles,args=(numbers,))
15) t2=Thread(target=squares,args=(numbers,))
16) t1.start()
17) t2.start()
18) t1.join()
19) t2.join()
20) print("The total time taken:",time.time()-begintime)
```

Setting and Getting Name of a Thread:

Every thread in python has name. It may be default name generated by Python or Customized Name provided by programmer.

We can get and set name of thread by using the following Thread class methods.

t.getName() → Returns Name of Thread

t.setName(newName) → To set our own name

Note: Every Thread has implicit variable "name" to represent name of Thread.

Eg:

```
1) from threading import *
2) print(current_thread().getName())
3) current_thread().setName("Pawan Kalyan")
4) print(current_thread().getName())
5) print(current_thread().name)
```

Output:

MainThread
Pawan Kalyan
Pawan Kalyan

Thread Identification Number (ident):

For every thread internally a unique identification number is available. We can access this id by using implicit variable "ident"

```
1) from threading import *
2) def test():
3)     print("Child Thread")
4) t=Thread(target=test)
```

```
5) t.start()
6) print("Main Thread Identification Number:",current_thread().ident)
7) print("Child Thread Identification Number:",t.ident)
```

Output:

Child Thread
Main Thread Identification Number: 2492
Child Thread Identification Number: 2768

active_count():

This function returns the number of active threads currently running.

Eg:

```
1) from threading import *
2) import time
3) def display():
4)     print(current_thread().getName(),"...started")
5)     time.sleep(3)
6)     print(current_thread().getName(),"...ended")
7)     print("The Number of active Threads:",active_count())
8) t1=Thread(target=display,name="ChildThread1")
9) t2=Thread(target=display,name="ChildThread2")
10) t3=Thread(target=display,name="ChildThread3")
11) t1.start()
12) t2.start()
13) t3.start()
14) print("The Number of active Threads:",active_count())
15) time.sleep(5)
16) print("The Number of active Threads:",active_count())
```

Output:

D:\python_classes>py test.py
The Number of active Threads: 1
ChildThread1 ...started
ChildThread2 ...started
ChildThread3 ...started
The Number of active Threads: 4
ChildThread1 ...ended
ChildThread2 ...ended
ChildThread3 ...ended
The Number of active Threads: 1

enumerate() function:

This function returns a list of all active threads currently running.

Eg:

```
1) from threading import *
2) import time
3) def display():
4)     print(current_thread().getName(),"...started")
5)     time.sleep(3)
6)     print(current_thread().getName(),"...ended")
7) t1=Thread(target=display,name="ChildThread1")
8) t2=Thread(target=display,name="ChildThread2")
9) t3=Thread(target=display,name="ChildThread3")
10) t1.start()
11) t2.start()
12) t3.start()
13) l=enumerate()
14) for t in l:
15)     print("Thread Name:",t.name)
16) time.sleep(5)
17) l=enumerate()
18) for t in l:
19)     print("Thread Name:",t.name)
```

Output:

```
D:\python_classes>py test.py
ChildThread1 ...started
ChildThread2 ...started
ChildThread3 ...started
Thread Name: MainThread
Thread Name: ChildThread1
Thread Name: ChildThread2
Thread Name: ChildThread3
ChildThread1 ...ended
ChildThread2 ...ended
ChildThread3 ...ended
Thread Name: MainThread
```

isAlive():

isAlive() method checks whether a thread is still executing or not.

Eg:

```
1) from threading import *
2) import time
3) def display():
4)     print(current_thread().getName(),"...started")
5)     time.sleep(3)
6)     print(current_thread().getName(),"...ended")
7) t1=Thread(target=display,name="ChildThread1")
8) t2=Thread(target=display,name="ChildThread2")
```

```
9) t1.start()
10) t2.start()
11)
12) print(t1.name,"is Alive :",t1.isAlive())
13) print(t2.name,"is Alive :",t2.isAlive())
14) time.sleep(5)
15) print(t1.name,"is Alive :",t1.isAlive())
16) print(t2.name,"is Alive :",t2.isAlive())
```

Output:

```
D:\python_classes>py test.py
ChildThread1 ...started
ChildThread2 ...started
ChildThread1 is Alive : True
ChildThread2 is Alive : True
ChildThread1 ...ended
ChildThread2 ...ended
ChildThread1 is Alive : False
ChildThread2 is Alive : False
```

join() method:

If a thread wants to wait until completing some other thread then we should go for join() method.

Eg:

```
1) from threading import *
2) import time
3) def display():
4)     for i in range(10):
5)         print("Seetha Thread")
6)         time.sleep(2)
7)
8) t=Thread(target=display)
9) t.start()
10) t.join()#This Line executed by Main Thread
11) for i in range(10):
12)     print("Rama Thread")
```

In the above example Main Thread waited until completing child thread. In this case output is:

```
Seetha Thread
```

Note: We can call join() method with time period also.

`t.join(seconds)`

In this case thread will wait only specified amount of time.

Eg.:

```
1) from threading import *
2) import time
3) def display():
4)     for i in range(10):
5)         print("Seetha Thread")
6)         time.sleep(2)
7)
8) t=Thread(target=display)
9) t.start()
10) t.join(5)#This Line executed by Main Thread
11) for i in range(10):
12)     print("Rama Thread")
```

In this case Main Thread waited only 5 seconds.

Output:

```
Rama Thread  
Rama Thread  
Rama Thread  
Seetha Thread
```

Summary of all methods related to threading module and Thread

Daemon Threads:

The threads which are running in the background are called Daemon Threads.

The main objective of Daemon Threads is to provide support for Non Daemon Threads(like main thread)

Eg: Garbage Collector

Whenever Main Thread runs with low memory, immediately PVM runs Garbage Collector to destroy useless objects and to provide free memory, so that Main Thread can continue its execution without having any memory problems.

We can check whether thread is Daemon or not by using t.isDaemon() method of Thread class or by using daemon property.

Eg:

```
1) from threading import *  
2) print(current_thread().isDaemon()) #False  
3) print(current_thread().daemon) #False
```

We can change Daemon nature by using setDaemon() method of Thread class.

```
t.setDaemon(True)
```

But we can use this method before starting of Thread.i.e once thread started, we cannot change its Daemon nature, otherwise we will get

RuntimeException:cannot set daemon status of active thread

Eg:

```
1) from threading import *  
2) print(current_thread().isDaemon())
```

```
3) current_thread().setDaemon(True)
```

RuntimeError: cannot set daemon status of active thread

Default Nature:

By default Main Thread is always non-daemon. But for the remaining threads Daemon nature will be inherited from parent to child.i.e if the Parent Thread is Daemon then child thread is also Daemon and if the Parent Thread is Non Daemon then ChildThread is also Non Daemon.

Eg:

```
1) from threading import *
2) def job():
3)     print("Child Thread")
4) t=Thread(target=job)
5) print(t.isDaemon())#False
6) t.setDaemon(True)
7) print(t.isDaemon()) #True
```

Note: Main Thread is always Non-Daemon and we cannot change its Daemon Nature b'z it is already started at the beginning only.

Whenever the last Non-Daemon Thread terminates automatically all Daemon Threads will be terminated.

Eg:

```
1) from threading import *
2) import time
3) def job():
4)     for i in range(10):
5)         print("Lazy Thread")
6)         time.sleep(2)
7)
8) t=Thread(target=job)
9) #t.setDaemon(True)==>Line-1
10) t.start()
11) time.sleep(5)
12) print("End Of Main Thread")
```

In the above program if we comment Line-1 then both Main Thread and Child Threads are Non Daemon and hence both will be executed until their completion.

In this case output is:

```
Lazy Thread
Lazy Thread
Lazy Thread
```

End Of Main Thread

Lazy Thread

If we are not commenting Line-1 then Main Thread is Non-Daemon and Child Thread is Daemon. Hence whenever MainThread terminates automatically child thread will be terminated. In this case output is

Lazy Thread

Lazy Thread

Lazy Thread

End of Main Thread

Synchronization:

If multiple threads are executing simultaneously then there may be a chance of data inconsistency problems.

Eg:

```
1) from threading import *
2) import time
3) def wish(name):
4)     for i in range(10):
5)         print("Good Evening:",end="")
6)         time.sleep(2)
7)         print(name)
8) t1=Thread(target=wish,args=("Dhoni",))
9) t2=Thread(target=wish,args=("Yuvraj",))
10) t1.start()
11) t2.start()
```

Output:

Good Evening:Good Evening:Yuvraj

Dhoni

Good Evening:Good Evening:Yuvraj

Dhoni

....

We are getting irregular output b'z both threads are executing simultaneously wish() function.

To overcome this problem we should go for synchronization.

In synchronization the threads will be executed one by one so that we can overcome data inconsistency problems.

Synchronization means at a time only one Thread

The main application areas of synchronization are

1. Online Reservation system
2. Funds Transfer from joint accounts
- etc

In Python, we can implement synchronization by using the following

1. Lock
2. RLock
3. Semaphore

Synchronization By using Lock concept:

Locks are the most fundamental synchronization mechanism provided by threading module.

We can create Lock object as follows

`l=Lock()`

The Lock object can be hold by only one thread at a time.If any other thread required the same lock then it will wait until thread releases lock.(similar to common wash rooms,public telephone booth etc)

A Thread can acquire the lock by using `acquire()` method.

`l.acquire()`

A Thread can release the lock by using `release()` method.

`l.release()`

Note: To call `release()` method compulsory thread should be owner of that lock.i.e thread should has the lock already,otherwise we will get Runtime Exception saying
`RuntimeError: release unlocked lock`

Eg:

```
1) from threading import *
2) l=Lock()
3) #l.acquire() ==>1
4) l.release()
```

If we are commenting line-1 then we will get

`RuntimeError: release unlocked lock`

Eg:

```
1) from threading import *
```

```

2) import time
3) l=Lock()
4) def wish(name):
5)     l.acquire()
6)     for i in range(10):
7)         print("Good Evening:",end="")
8)         time.sleep(2)
9)         print(name)
10)    l.release()
11)
12) t1=Thread(target=wish,args=("Dhoni",))
13) t2=Thread(target=wish,args=("Yuvraj",))
14) t3=Thread(target=wish,args=("Kohli",))
15) t1.start()
16) t2.start()
17) t3.start()

```

In the above program at a time only one thread is allowed to execute wish() method and hence we will get regular output.

Problem with Simple Lock:

The standard Lock object does not care which thread is currently holding that lock. If the lock is held and any thread attempts to acquire lock, then it will be blocked, even the same thread is already holding that lock.

Eg:

```

1) from threading import *
2) l=Lock()
3) print("Main Thread trying to acquire Lock")
4) l.acquire()
5) print("Main Thread trying to acquire Lock Again")
6) l.acquire()

```

Output:

```

D:\python_classes>py test.py
Main Thread trying to acquire Lock
Main Thread trying to acquire Lock Again
--
```

In the above Program main thread will be blocked b'z it is trying to acquire the lock second time.

Note: To kill the blocking thread from windows command prompt we have to use ctrl+break. Here ctrl+C won't work.

If the Thread calls recursive functions or nested access to resources, then the thread may try to acquire the same lock again and again, which may block our thread.

Hence Traditional Locking mechanism won't work for executing recursive functions.

To overcome this problem, we should go for RLock(Reentrant Lock). Reentrant means the thread can acquire the same lock again and again. If the lock is held by other threads then only the thread will be blocked.

Reentrant facility is available only for owner thread but not for other threads.

Eg:

```
1) from threading import *
2) l=RLock()
3) print("Main Thread trying to acquire Lock")
4) l.acquire()
5) print("Main Thread trying to acquire Lock Again")
6) l.acquire()
```

In this case Main Thread won't be Locked b'z thread can acquire the lock any number of times.

This RLock keeps track of recursion level and hence for every acquire() call compulsory release() call should be available. i.e the number of acquire() calls and release() calls should be matched then only lock will be released.

Eg:

```
l=RLock()
l.acquire()
l.acquire()
l.release()
l.release()
```

After 2 release() calls only the Lock will be released.

Note:

1. Only owner thread can acquire the lock multiple times
2. The number of acquire() calls and release() calls should be matched.

Demo Program for synchronization by using RLock:

```
1) from threading import *
2) import time
3) l=RLock()
4) def factorial(n):
5)     l.acquire()
6)     if n==0:
7)         result=1
8)     else:
9)         result=n*factorial(n-1)
10)    l.release()
11)    return result
```

```
12)
13) def results(n):
14)     print("The Factorial of",n,"is:",factorial(n))
15)
16) t1=Thread(target=results,args=(5,))
17) t2=Thread(target=results,args=(9,))
18) t1.start()
19) t2.start()
```

Output:

The Factorial of 5 is: 120
The Factorial of 9 is: 362880

In the above program instead of RLock if we use normal Lock then the thread will be blocked.

Difference between Lock and RLock:

table

Lock:

1. Lock object can be acquired by only one thread at a time. Even owner thread also cannot acquire multiple times.
2. Not suitable to execute recursive functions and nested access calls
3. In this case Lock object will takes care only Locked or unlocked and it never takes care about owner thread and recursion level.

RLock:

1. RLock object can be acquired by only one thread at a time, but owner thread can acquire same lock object multiple times.
2. Best suitable to execute recursive functions and nested access calls
3. In this case RLock object will takes care whether Locked or unlocked and owner thread information, recursion level.

Synchronization by using Semaphore:

In the case of Lock and RLock, at a time only one thread is allowed to execute.

Sometimes our requirement is at a time a particular number of threads are allowed to access (like at a time 10 members are allowed to access database server, 4 members are allowed to access Network connection etc). To handle this requirement we cannot use Lock and RLock concepts and we should go for Semaphore concept.

Semaphore can be used to limit the access to the shared resources with limited capacity.

Semaphore is advanced Synchronization Mechanism.

We can create Semaphore object as follows.

```
s=Semaphore(counter)
```

Here counter represents the maximum number of threads are allowed to access simultaneously. The default value of counter is 1.

Whenever thread executes acquire() method, then the counter value will be decremented by 1 and if thread executes release() method then the counter value will be incremented by 1.

i.e for every acquire() call counter value will be decremented and for every release() call counter value will be incremented.

Case-1: s=Semaphore()

In this case counter value is 1 and at a time only one thread is allowed to access. It is exactly same as Lock concept.

Case-2: s=Semaphore(3)

In this case Semaphore object can be accessed by 3 threads at a time. The remaining threads have to wait until releasing the semaphore.

Eg:

```
1) from threading import *
2) import time
3) s=Semaphore(2)
4) def wish(name):
5)     s.acquire()
6)     for i in range(10):
7)         print("Good Evening:",end="")
8)         time.sleep(2)
9)         print(name)
10)    s.release()
11)
12) t1=Thread(target=wish,args=("Dhoni",))
13) t2=Thread(target=wish,args=("Yuvraj",))
14) t3=Thread(target=wish,args=("Kohli",))
15) t4=Thread(target=wish,args=("Rohit",))
16) t5=Thread(target=wish,args=("Pandya",))
17) t1.start()
18) t2.start()
19) t3.start()
20) t4.start()
21) t5.start()
```

In the above program at a time 2 threads are allowed to access semaphore and hence 2 threads are allowed to execute wish() function.

BoundedSemaphore:

Normal Semaphore is an unlimited semaphore which allows us to call release() method any number of times to increment counter. The number of release() calls can exceed the number of acquire() calls also.

Eg:

```
1) from threading import *
2) s=Semaphore(2)
3) s.acquire()
4) s.acquire()
5) s.release()
6) s.release()
7) s.release()
8) s.release()
9) print("End")
```

It is valid because in normal semaphore we can call release() any number of times.

BoundedSemaphore is exactly same as Semaphore except that the number of release() calls should not exceed the number of acquire() calls, otherwise we will get

ValueError: Semaphore released too many times

Eg:

```
1) from threading import *
2) s=BoundedSemaphore(2)
3) s.acquire()
4) s.acquire()
5) s.release()
6) s.release()
7) s.release()
8) s.release()
9) print("End")
```

ValueError: Semaphore released too many times

It is invalid b'z the number of release() calls should not exceed the number of acquire() calls in BoundedSemaphore.

Note: To prevent simple programming mistakes, it is recommended to use BoundedSemaphore over normal Semaphore.

Difference between Lock and Semaphore:

At a time Lock object can be acquired by only one thread, but Semaphore object can be acquired by fixed number of threads specified by counter value.

Conclusion:

The main advantage of synchronization is we can overcome data inconsistency problems. But the main disadvantage of synchronization is it increases waiting time of threads and creates performance problems. Hence if there is no specific requirement then it is not recommended to use synchronization.

Inter Thread Communication:

Some times as the part of programming requirement, threads are required to communicate with each other. This concept is nothing but interthread communication.

Eg: After producing items Producer thread has to communicate with Consumer thread to notify about new item. Then consumer thread can consume that new item.

In Python, we can implement interthread communication by using the following ways

1. Event
 2. Condition
 3. Queue
- etc

Interthread communication by using Event Objects:

Event object is the simplest communication mechanism between the threads. One thread signals an event and other thereds wait for it.

We can create Event object as follows...

```
event = threading.Event()
```

Event manages an internal flag that can set() or clear()
Threads can wait until event set.

Methods of Event class:

1. **set()** → internal flag value will become True and it represents GREEN signal for all waiting threads.
2. **clear()** → internal flag value will become False and it represents RED signal for all waiting threads.
3. **isSet()** → This method can be used whether the event is set or not

4. `wait()` | `wait(seconds)` ➔ Thread can wait until event is set

Pseudo Code:

```
event = threading.Event()

#consumer thread has to wait until event is set
event.wait()

#producer thread can set or clear event
event.set()
event.clear()
```

Demo Program-1:

```
1) from threading import *
2) import time
3) def producer():
4)     time.sleep(5)
5)     print("Producer thread producing items:")
6)     print("Producer thread giving notification by setting event")
7)     event.set()
8) def consumer():
9)     print("Consumer thread is waiting for updation")
10)    event.wait()
11)    print("Consumer thread got notification and consuming items")
12)
13) event=Event()
14) t1=Thread(target=producer)
15) t2=Thread(target=consumer)
16) t1.start()
17) t2.start()
```

Output:

Consumer thread is waiting for updation
Producer thread producing items
Producer thread giving notification by setting event
Consumer thread got notification and consuming items

Demo Program-2:

```
1) from threading import *
2) import time
3) def trafficpolice():
4)     while True:
5)         time.sleep(10)
6)         print("Traffic Police Giving GREEN Signal")
```

```

7)     event.set()
8)     time.sleep(20)
9)     print("Traffic Police Giving RED Signal")
10)    event.clear()
11)   def driver():
12)     num=0
13)     while True:
14)       print("Drivers waiting for GREEN Signal")
15)       event.wait()
16)       print("Traffic Signal is GREEN...Vehicles can move")
17)       while event.isSet():
18)         num=num+1
19)         print("Vehicle No:",num,"Crossing the Signal")
20)         time.sleep(2)
21)         print("Traffic Signal is RED...Drivers have to wait")
22)   event=Event()
23) t1=Thread(target=trafficpolice)
24) t2=Thread(target=driver)
25) t1.start()
26) t2.start()

```

In the above program driver thread has to wait until Trafficpolice thread sets event.i.e until giving GREEN signal.Once Traffic police thread sets event(giving GREEN signal),vehicles can cross the signal.Once traffic police thread clears event (giving RED Signal)then the driver thread has to wait.

Interthread communication by using Condition Object:

Condition is the more advanced version of Event object for interthread communication.A condition represents some kind of state change in the application like producing item or consuming item. Threads can wait for that condition and threads can be notified once condition happens.i.e Condition object allows one or more threads to wait until notified by another thread.

Condition is always associated with a lock (ReentrantLock).

A condition has acquire() and release() methods that call the corresponding methods of the associated lock.

We can create Condition object as follows

```
condition = threading.Condition()
```

Methods of Condition:

1. acquire() ➔ To acquire Condition object before producing or consuming items.i.e thread acquiring internal lock.
2. release() ➔ To release Condition object after producing or consuming items. i.e thread releases internal lock
3. wait() | wait(time) ➔ To wait until getting Notification or time expired

4. `notify()` → To give notification for one waiting thread

5. `notifyAll()` → To give notification for all waiting threads

Case Study:

The producing thread needs to acquire the Condition before producing item to the resource and notifying the consumers.

```
#Producer Thread
...generate item..
condition.acquire()
...add item to the resource...
condition.notify()#signal that a new item is available(notifyAll())
condition.release()
```

The Consumer must acquire the Condition and then it can consume items from the resource

```
#Consumer Thread
condition.acquire()
condition.wait()
consume item
condition.release()
```

Demo Program-1:

```
1) from threading import *
2) def consume(c):
3)     c.acquire()
4)     print("Consumer waiting for updation")
5)     c.wait()
6)     print("Consumer got notification & consuming the item")
7)     c.release()
8)
9) def produce(c):
10)    c.acquire()
11)    print("Producer Producing Items")
12)    print("Producer giving Notification")
13)    c.notify()
14)    c.release()
15)
16) c=Condition()
17) t1=Thread(target=consume,args=(c,))
18) t2=Thread(target=produce,args=(c,))
19) t1.start()
20) t2.start()
```

Output:

Consumer waiting for updation
Producer Producing Items
Producer giving Notification
Consumer got notification & consuming the item

Demo Program-2:

```
1) from threading import *
2) import time
3) import random
4) items=[]
5) def produce(c):
6)     while True:
7)         c.acquire()
8)         item=random.randint(1,100)
9)         print("Producer Producing Item:",item)
10)        items.append(item)
11)        print("Producer giving Notification")
12)        c.notify()
13)        c.release()
14)        time.sleep(5)
15)
16) def consume(c):
17)     while True:
18)         c.acquire()
19)         print("Consumer waiting for updation")
20)         c.wait()
21)         print("Consumer consumed the item",items.pop())
22)         c.release()
23)         time.sleep(5)
24)
25) c=Condition()
26) t1=Thread(target=consume,args=(c,))
27) t2=Thread(target=produce,args=(c,))
28) t1.start()
29) t2.start()
```

Output:

Consumer waiting for updation
Producer Producing Item: 49
Producer giving Notification
Consumer consumed the item 49

.....

In the above program Consumer thread expecting updation and hence it is responsible to call wait() method on Condition object.

Producer thread performing updation and hence it is responsible to call notify() or notifyAll() on Condition object.

Interthread communication by using Queue:

Queues Concept is the most enhanced Mechanism for interthread communication and to share data between threads.

Queue internally has Condition and that Condition has Lock.Hence whenever we are using Queue we are not required to worry about Synchronization.

If we want to use Queues first we should import queue module.

```
import queue
```

We can create Queue object as follows

```
q = queue.Queue()
```

Important Methods of Queue:

1. put(): Put an item into the queue.
2. get(): Remove and return an item from the queue.

Producer Thread uses put() method to insert data in the queue. Internally this method has logic to acquire the lock before inserting data into queue. After inserting data lock will be released automatically.

put() method also checks whether the queue is full or not and if queue is full then the Producer thread will entered in to waiting state by calling wait() method internally.

Consumer Thread uses get() method to remove and get data from the queue. Internally this method has logic to acquire the lock before removing data from the queue.Once removal completed then the lock will be released automatically.

If the queue is empty then consumer thread will entered into waiting state by calling wait() method internally.Once queue updated with data then the thread will be notified automatically.

Note:

The queue module takes care of locking for us which is a great advantage.

Eg:

- 1) `from threading import *`
- 2) `import time`
- 3) `import random`
- 4) `import queue`
- 5) `def produce(q):`

```
6) while True:  
7)     item=random.randint(1,100)  
8)     print("Producer Producing Item:",item)  
9)     q.put(item)  
10)    print("Producer giving Notification")  
11)    time.sleep(5)  
12) def consume(q):  
13)     while True:  
14)         print("Consumer waiting for updation")  
15)         print("Consumer consumed the item:",q.get())  
16)         time.sleep(5)  
17)  
18) q=queue.Queue()  
19) t1=Thread(target=consume,args=(q,))  
20) t2=Thread(target=produce,args=(q,))  
21) t1.start()  
22) t2.start()
```

Output:

Consumer waiting for updation
Producer Producing Item: 58
Producer giving Notification
Consumer consumed the item: 58

Types of Queues:

Python Supports 3 Types of Queues.

1. FIFO Queue:

```
q = queue.Queue()
```

This is Default Behaviour. In which order we put items in the queue, in the same order the items will come out (FIFO-First In First Out).

Eg:

```
1) import queue  
2) q=queue.Queue()  
3) q.put(10)  
4) q.put(5)  
5) q.put(20)  
6) q.put(15)  
7) while not q.empty():  
8)     print(q.get(),end=' ')
```

Output: 10 5 20 15

2. LIFO Queue:

The removal will be happen in the reverse order of insertion(Last In First Out)

Eg:

```
1) import queue
2) q=queue.LifoQueue()
3) q.put(10)
4) q.put(5)
5) q.put(20)
6) q.put(15)
7) while not q.empty():
8)     print(q.get(),end=' ')
```

Output: 15 20 5 10

3. Priority Queue:

The elements will be inserted based on some priority order.

```
1) import queue
2) q=queue.PriorityQueue()
3) q.put(10)
4) q.put(5)
5) q.put(20)
6) q.put(15)
7) while not q.empty():
8)     print(q.get(),end=' ')
```

Output: 5 10 15 20

Eg 2: If the data is non-numeric, then we have to provide our data in the form of tuple.

(x,y)

x is priority

y is our element

```
1) import queue
2) q=queue.PriorityQueue()
3) q.put((1,"AAA"))
4) q.put((3,"CCC"))
5) q.put((2,"BBB"))
6) q.put((4,"DDD"))
7) while not q.empty():
8)     print(q.get()[1],end=' ')
```

Output: AAA BBB CCC DDD

Good Programming Practices with usage of Locks:

Case-1:

It is highly recommended to write code of releasing locks inside finally block. The advantage is lock will be released always whether exception raised or not raised and whether handled or not handled.

```
l=threading.Lock()
l.acquire()
try:
    perform required safe operations
finally:
    l.release()
```

Demo Program:

```
1) from threading import *
2) import time
3) l=Lock()
4) def wish(name):
5)     l.acquire()
6)     try:
7)         for i in range(10):
8)             print("Good Evening:",end="")
9)             time.sleep(2)
10)            print(name)
11)    finally:
12)        l.release()
13)
14) t1=Thread(target=wish,args=("Dhoni",))
15) t2=Thread(target=wish,args=("Yuvraj",))
16) t3=Thread(target=wish,args=("Kohli",))
17) t1.start()
18) t2.start()
19) t3.start()
```

Case-2:

It is highly recommended to acquire lock by using with statement. The main advantage of with statement is the lock will be released automatically once control reaches end of with block and we are not required to release explicitly.

This is exactly same as usage of with statement for files.

Example for File:

```
with open('demo.txt','w') as f:  
    f.write("Hello...")
```

Example for Lock:

```
lock=threading.  
Lock()with  
lock:  
    perform required safe  
    operations lock will be  
    released automatically
```

Demo Program:

```
1) from threading import *  
2) import time  
3) lock=Lock()  
4) def wish(name):  
5)     with lock:  
6)         for i in range(10):  
7)             print("Good Evening:",end="")  
8)             time.sleep(2)  
9)             print(name)  
10) t1=Thread(target=wish,args=("Dhoni",))  
11) t2=Thread(target=wish,args=("Yuvraj",))  
12) t3=Thread(target=wish,args=("Kohli",))  
13) t1.start()  
14) t2.start()  
15) t3.start()
```

Q. What is the advantage of using with statement to acquire a lock in threading? Lock will be released automatically once control reaches end of with block and We are not required to release explicitly.

Note: We can use with statement in multithreading for the following cases:

1. Lock
2. RLock
3. Semaphore
4. Condition

Simple Graphics and Image Processing

The Plan For Today

- Website Updates
- Intro to Python Quiz Corrections
- Missing Assignments
- Graphics and Images
 - Simple Graphics – Turtle Graphics
 - Image Processing
- Assignment
- Work Time

Upcoming

Python Exam – Fri. 4/24

Simple Graphics

- **Graphics:** Discipline that underlies the representation and display of geometric shapes in two- and three-dimensional space
- A **Turtle graphics** toolkit provides a simple and enjoyable way to draw pictures in a window
 - `turtle` is a non-standard, open-source Python module

Fundamentals of Python: First Programs

5

Overview of Turtle Graphics

- Turtle graphics originally developed as part of the children's programming language Logo
 - Created by Seymour Papert and his colleagues at MIT in the late 1960s
- Analogy: Turtle crawling on a piece of paper, with a pen tied to its tail
 - Sheet of paper is a window on a display screen
 - Position specified with (x, y) coordinates
 - Cartesian **coordinate system**, with origin $(0, 0)$ at the center of a window

Fundamentals of Python: First Programs

6

Overview of Turtle Graphics (continued)

Heading	Specified in degrees, the heading or direction increases in value as the turtle turns to the left, or counterclockwise. Conversely, a negative quantity of degrees indicates a right, or clockwise, turn. The turtle is initially facing east, or 0 degrees. North is 90 degrees.
Color	Initially black, the color can be changed to any of more than 16 million other colors.
Width	This is the width of the line drawn when the turtle moves. The initial width is 1 pixel. (You'll learn more about pixels shortly.)
Down	This attribute, which can be either true or false, controls whether the turtle's pen is up or down. When true (that is, when the pen is down), the turtle draws a line when it moves. When false (that is, when the pen is up), the turtle can move without drawing a line.

[TABLE 7.1] Some attributes of a turtle

- Together, these attributes make up a turtle's state

Fundamentals of Python: First Programs

7

Turtle Operations

Turtle METHOD	WHAT IT DOES
<code>t = Turtle()</code>	Creates a new <code>Turtle</code> object and opens its window.
<code>t.home()</code>	Moves <code>t</code> to the center of the window and then points <code>t</code> east.
<code>t.up()</code>	Raises <code>t</code> 's pen from the drawing surface.
<code>t.down()</code>	Lowers <code>t</code> 's pen to the drawing surface.
<code>t.setheading(degrees)</code>	Points <code>t</code> in the indicated direction, which is specified in degrees. East is 0 degrees, north is 90 degrees, west is 180 degrees, and south is 270 degrees.
<code>t.left(degrees)</code> <code>t.right(degrees)</code>	Rotates <code>t</code> to the left or the right by the specified degrees.

[TABLE 7.2] The `Turtle` methods

Fundamentals of Python: First Programs

8

Turtle Operations (continued)

<code>t.goto(x, y)</code> <code>t.forward(distance)</code>	Moves <code>t</code> to the specified position. Moves <code>t</code> the specified distance in the current direction.
<code>t.pencolor(r, g, b)</code> <code>t.pencolor(string)</code>	Changes the pen color of <code>t</code> to the specified RGB value or to the specified string, such as ' <code>red</code> '. Returns the current color of <code>t</code> when the arguments are omitted.
<code>t.fillcolor(r, g, b)</code> <code>t.fillcolor(string)</code>	Changes the fill color of <code>t</code> to the specified RGB value or to the specified string, such as ' <code>red</code> '. Returns the current fill color of <code>t</code> when the arguments are omitted.
<code>t.begin_fill()</code> <code>t.end_fill()</code>	Ends a set of turtle commands that will draw a filled shape using the current fill color.
<code>t.width(pixels)</code>	Changes the width of <code>t</code> to the specified number of pixels. Returns <code>t</code> 's current width when the argument is omitted.
<code>t.hideturtle()</code> <code>t.showturtle()</code>	Makes the turtle invisible or visible.
<code>t.position()</code>	Returns the current position (x, y) of <code>t</code> .
<code>t.heading()</code>	Returns the current direction of <code>t</code> .
<code>t.isdown()</code>	Returns <code>true</code> if <code>t</code> 's pen is down or <code>false</code> otherwise.

Fundamentals of Python: First Programs

9

Object Instantiation and the `turtle` Module

- Before you apply any methods to an object, you must create the object (i.e., an **instance** of)
- Instantiation:** Process of creating an object
- Use a **constructor** to instantiate an object:

```
<variable name> = <class name>(<any arguments>)
```

- To instantiate the **Turtle** class:

```
>>> from turtle import Turtle
>>>
>>> t = Turtle()
```

Fundamentals of Python: First Programs

11

Drawing Two-Dimensional Shapes

- Many graphics applications use **vector graphics**, or the drawing of simple two-dimensional shapes, such as rectangles, triangles, and circles

```
def drawPolygon(t, vertices):
    """Draws a polygon from a list of vertices.
    The list has the form [(x1, y1), ..., (xn, yn)]"""
    t.up()
    (x, y) = vertices[-1]
    t.goto(x, y)
    t.down()
    for (x, y) in vertices:
        t.goto(x, y)

>>> from turtle import Turtle
>>> t = Turtle()
>>> t.hideturtle()
>>> drawPolygon(t, [(20, 20), (-20, 20), (-20, -20)])
```

Fundamentals of Python: First Programs

14

Taking a Random Walk

- Like any animal, a turtle can wander around randomly:

```
from turtle import Turtle
import random

def randomWalk(t, turns, distance = 20):
    """Takes a random number of degrees and moves
    a given distance for a fixed number of turns."""
    for x in range(turns):
        t.left(random.randint(0, 360))
        t.forward(distance)

randomWalk(Turtle(), 40)
```

Fundamentals of Python: First Programs

16

Colors and the RGB System

- Display area on a computer screen is made up of colored dots called picture elements or **pixels**
- Each pixel represents a color – the default is black
- RGB** is a common system for representing colors
 - RGB stands for red, green, and blue
 - Each color component can range from 0 – 255
 - 255 → maximum saturation of a color component
 - 0 → total absence of that color component
 - A **true color** system

Fundamentals of Python: First Programs

18

Colors and the RGB System (cont'd)

COLOR	RGB VALUE
Black	(0, 0, 0)
Red	(255, 0, 0)
Green	(0, 255, 0)
Blue	(0, 0, 255)
Yellow	(255, 255, 0)
Gray	(127, 127, 127)
White	(255, 255, 255)

[TABLE 7.3] Some example colors and their RGB values

- Each color component requires 8 bits; total number of bits needed to represent a color value is 24
 - Total number of RGB colors is 2^{24} (16,777,216)

Fundamentals of Python: First Programs

19

Example: Drawing with Random Colors

- The **Turtle** class includes a `pencolor` method for changing the turtle's drawing color
 - Expects integers for the three RGB components

```
from turtle import Turtle
import random

def drawSquare(t, x, y, length):
    """ Draws a square with the upper-left corner (x, y)
    and the given length. """
    t.up()
    t.goto(x, y)
    t.setheading(270)
    t.down()
    for count in range(4):
        t.forward(length)
        t.left(90)
```

Fundamentals of Python: First Programs

20

Examining an Object's Attributes

- **Mutator methods** change the internal state of a `Turtle` method
 - Example: `pencolor` method
- **Accessor methods** return the values of a `Turtle` object's attributes without altering its state
 - Example: `position` method

Fundamentals of Python: First Programs

21

Manipulating a Turtle's Screen

- The `Screen` object's attributes include its width and height in pixels and its background color
- Use `t.screen` to access a turtle's `Screen` object, then call a `Screen` method on this object

```
>>> from turtle import Turtle
>>> t = Turtle()
>>> t.screen.bgcolor("orange")
```

Fundamentals of Python: From First Programs Through Data Structures

22

Image Processing

- Digital image processing includes the principles and techniques for the following:
 - The capture of images with devices such as flatbed scanners and digital cameras
 - The representation and storage of images in efficient file formats
 - Constructing the algorithms in image-manipulation programs such as Adobe Photoshop

Fundamentals of Python: First Programs

29

Analog and Digital Information

- Computers must use digital information which consists of **discrete values**
 - Example: Individual integers, characters of text, or bits
- The information contained in images, sound, and much of the rest of the physical world is analog
 - **Analog information** contains a **continuous range** of values
- Ticks representing seconds on an analog clock's face represent an attempt to **sample** moments of time as discrete values (time itself is analog)

Fundamentals of Python: First Programs

30

Sampling and Digitizing Images

- A visual scene projects an infinite set of color and intensity values onto a two-dimensional sensing medium
 - If you sample enough of these values, digital information can represent an image more or less indistinguishable (to human eye) from original scene
- Sampling devices measure discrete color values at distinct points on a two-dimensional grid
 - These values are pixels
 - As more pixels are sampled, the more realistic the resulting image will appear

Fundamentals of Python: First Programs

31

Image File Formats

- Once an image has been sampled, it can be stored in one of many file formats
- A **raw image file** saves all of the sampled information
- Data can be compressed to minimize its file size
 - JPEG (Joint Photographic Experts Group)
 - Uses **lossless compression** and a **lossy scheme**
 - GIF (Graphic Interchange Format)
 - Uses a lossy compression and a **color palette** of up to 256 of the most prevalent colors in the image

Fundamentals of Python: First Programs

32

Image-Manipulation Operations

- Image-manipulation programs either transform the information in the pixels or alter the arrangement of the pixels in the image
- Examples:
 - Rotate an image
 - Convert an image from color to grayscale
 - Blur all or part of an image
 - Sharpen all or part of an image
 - Control the brightness of an image
 - Perform edge detection on an image
 - Enlarge or reduce an image's size

Fundamentals of Python: First Programs

33

The Properties of Images

- The coordinates of pixels in the two-dimensional grid of an image range from $(0, 0)$ at the upper-left corner to $(width-1, height-1)$ at lower-right corner
 - *width/height* are the image's dimensions in pixels
 - Thus, the **screen coordinate system** for the display of an image is different from the standard Cartesian coordinate system that we used with Turtle graphics
- The RGB color system is a common way of representing the colors in images

Fundamentals of Python: First Programs

34

The `images` Module

- Non-standard, open-source Python tool
 - `Image` class represents an image as a two-dimensional grid of RGB values

```
>>> from images import Image
>>> image = Image("smokey.gif")
>>> image.draw()
```



[FIGURE 7.9] An image display window

Fundamentals of Python: First Programs

35

The `images` Module (continued)

```
>>> image = Image(150, 150)
>>> image.draw()
>>> blue = (0, 0, 255)
>>> y = image.getHeight() // 2
>>> for x in range(image.getWidth()):
    image.setPixel(x, y, blue)

>>> image.draw()
```



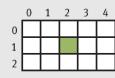
[FIGURE 7.10] An image before and after replacing the pixels

Fundamentals of Python: First Programs

36

A Loop Pattern for Traversing a Grid

- Most of the loops we have used in this book have had a **linear loop structure**
- Many image-processing algorithms use a **nested loop structure** to traverse a two-dimensional grid of pixels



[FIGURE 7.11] A grid with 3 rows and 5 columns

Fundamentals of Python: First Programs

37

A Loop Pattern for Traversing a Grid (continued)

```
>>> width = 2
>>> height = 3
>>> for y in range(height):
    for x in range(width):
        print((x, y))
    print()

(0, 0) (1, 0)
(0, 1) (1, 1)
(0, 2) (1, 2)
>>>
```

- Previous loop uses a **row-major traversal**
 - We use this template to develop many of the algorithms that follow:

```
for y in range(height):
    for x in range(width):
        do something at position (x, y)
```

Fundamentals of Python: First Programs

38

A Word on Tuples

- A pixel's RGB values are stored in a tuple:

```
>>> (r, g, b) = image.getPixel(0, 0)

>>> r
194
>>> g
221
>>> b
114

>>> image.setPixel(0, 0, (r + 10, g + 10, b + 10))
```

Fundamentals of Python: First Programs

39

Converting an Image to Black and White

- For each pixel, compute average of R/G/B values
- Then, reset pixel's color values to 0 (black) if the average is closer to 0, or to 255 (white) if the average is closer to 255

```
def blackAndWhite(image):
    """Converts the argument image to black and white."""
    blackPixel = (0, 0, 0)
    whitePixel = (255, 255, 255)
    for y in range(image.getHeight()):
        for x in range(image.getWidth()):
            (r, g, b) = image.getPixel(x, y)
            average = (r + g + b) // 3
            if average < 128:
                image.setPixel(x, y, blackPixel)
            else:
                image.setPixel(x, y, whitePixel)
```

Fundamentals of Python: First Programs

40

Converting an Image to Black and White (continued)



[FIGURE 7.12] Converting a color image to black and white

```
from images import Image

# Code for blackAndWhite's function definition goes here

def main(filename = "smokey.gif"):
    image = Image.open(filename)
    print("Open the image window to continue. ")
    image.draw()
    blackAndWhite(image)
    print("Close the image window to quit. ")
    image.draw()
    image.close()

main()
```

Fundamentals of Python: First Programs

41

Converting an Image to Grayscale

- Black and white photographs contain various shades of gray known as **grayscale**
- Grayscale can be an economical scheme (the only color values might be 8, 16, or 256 shades of gray)
- A simple method:

```
average = (r + g + b) // 3
image.setPixel(x, y, (average, average, average))
```

- Problem: Does not reflect manner in which different color components affect human perception
- Scheme needs to take differences in **luminance** into account

Fundamentals of Python: First Programs

42

Converting an Image to Grayscale (continued)

```
def grayscale(image):
    """Converts the argument image to grayscale."""
    for y in range(image.getHeight()):
        for x in range(image.getWidth()):
            (r, g, b) = image.getPixel(x, y)
            r = int(r * 0.299)
            g = int(g * 0.587)
            b = int(b * 0.114)
            lum = r + g + b
            image.setPixel(x, y, (lum, lum, lum))
```



[FIGURE 7.13] Converting a color image to grayscale

Fundamentals of Python: First Programs

43

Copying an Image

- The method **clone** builds and returns a new image with the same attributes as the original one, but with an empty string as the filename

```
>>> from images import Image
>>> image = Image("smokey.gif")
>>> image.draw()
>>> newImage = image.clone()      # Create a copy of image
>>> newImage.draw()              # Change in second window only
>>> grayscale(newImage)
>>> newImage.draw()
>>> image.draw()
```

Fundamentals of Python: First Programs

44

Blurring an Image

- Pixilation can be mitigated by blurring

```
def blur(image):
    """Builds and returns a new image which is a blurred
    copy of the argument image."""

    def tripleSum(triple1, triple2):
        #1
        (r1, g1, b1) = triple1
        (r2, g2, b2) = triple2
        return (r1 + r2, g1 + g2, b1 + b2)

    new = image.clone()
    for y in range(1, image.getHeight() - 1):
        for x in range(1, image.getWidth() - 1):
            oldP = image.getPixel(x, y)
            left = image.getPixel(x - 1, y) # To left
            right = image.getPixel(x + 1, y) # To right
            top = image.getPixel(x, y - 1) # Above
            bottom = image.getPixel(x, y + 1) # Below
            sums = reduce(tripleSum,
                          [oldP, left, right, top, bottom]) #2
            averages = tuple(map(lambda x: x / 5, sums)) #3
            new.setPixel(x, y, averages)
    return new
```

Fundamentals of Python: First Programs

45

Edge Detection

- Edge detection removes the full colors to uncover the outlines of the objects represented in the image

```
def detectEdges(image, amount):
    """Builds and returns a new image in which the
    edges of the argument image are highlighted and
    the colors are reduced to black and white."""

    def average(triple):
        (r, g, b) = triple
        return (r + g + b) // 3

    blackPixel = (0, 0, 0)
    whitePixel = (255, 255, 255)
    new = image.clone()
```

continued

Fundamentals of Python: First Programs

46

Edge Detection (continued)

```
for y in range(image.getHeight() - 1):
    for x in range(1, image.getWidth()):
        oldPixel = image.getPixel(x, y)
        leftPixel = image.getPixel(x - 1, y)
        bottomPixel = image.getPixel(x, y + 1)
        oldLum = average(oldPixel)
        leftLum = average(leftPixel)
        bottomLum = average(bottomPixel)
        if abs(oldLum - leftLum) > amount or \
           abs(oldLum - bottomLum) > amount:
            new.setPixel(x, y, blackPixel)
        else:
            new.setPixel(x, y, whitePixel)
    return new
```

Fundamentals of Python: First Programs

47

Reducing the Image Size

- The size and the quality of an image on a display medium depend on two factors:
 - Image's width and height in pixels
 - Display medium's resolution
 - Measured in pixels, or dots per inch (DPI)
- The resolution of an image can be set before the image is captured
 - A higher DPI causes sampling device to take more samples (pixels) through the two-dimensional grid
- A size reduction usually preserves an image's aspect ratio

Fundamentals of Python: First Programs

48

Reducing the Image Size (continued)

```
def shrink(image, factor):
    """Builds and returns a new image which is a smaller
    copy of the argument image, by the factor argument."""

    width = image.getWidth()
    height = image.getHeight()
    new = Image(width // factor, height // factor)
    oldY = 0
    newY = 0
    while oldY < height - factor:
        oldX = 0
        newX = 0
        while oldX < width - factor:
            oldP = image.getPixel(oldX, oldY)
            new.setPixel(newX, newY, oldP)
            oldX += factor
            newX += 1
        oldY += factor
        newY += 1
    return new
```

- Reducing size throws away some pixel information

Fundamentals of Python: First Programs

49

Work Time – Remember...

- PyCharm is a free download!
- Keep up the great conversation and working together!
- Make use of example code!
- Try, try, try!
- Make sure you understand code you write.
- Thanks for leaving the lab looking great!



Python Turtle Programming

Turtle is a Python library which used to create graphics, pictures, and games. It was developed by **Wally Feurzeig, Seymour Papert** and **Cynthia Solomon** in 1967. It was a part of the original Logo programming language.

The Logo programming language was popular among the kids because it enables us to draw attractive graphs to the screen in the simple way. It is like a little object on the screen, which can move according to the desired position. Similarly, turtle library comes with the interactive feature that gives the flexibility to work with Python.

In this tutorial, we will learn the basic concepts of the turtle library, how to set the turtle up on a computer, programming with the Python turtle library, few important turtle commands, and develop a short but attractive design using the Python turtle library.

Introduction

Turtle is a pre-installed library in Python that is similar to the virtual canvas that we can draw pictures and attractive shapes. It provides the onscreen pen that we can use for drawing...

The **turtle** Library is primarily designed to introduce children to the world of programming. With the help of Turtle's library, new programmers can get an idea of how we can do programming with Python in a fun and interactive way.

It is beneficial to the children and for the experienced programmer because it allows designing unique shapes, attractive pictures, and various games. We can also design the mini games and animation. In the upcoming section, we will learn to various functionality of turtle library.

Getting started with turtle

Before working with the turtle library, we must ensure the two most essential things to do programming.

- **Python Environment** - We must be familiar with the working Python environment. We can use applications such as **IDLE** or **Jupiter Notebook**. We can also use the Python interactive shell.
- **Python Version** - We must have Python 3 in our system; if not, then download it from Python's official website.

The turtle is built in library so we don't need to install separately. We just need to import the library into our Python environment.

The Python turtle library consists of all important methods and functions that we will need to create our designs and images. Import the turtle library using the following command.

1. **import** turtle

Now, we can access all methods and functions. First, we need to create a dedicated window where we carry out each drawing command. We can do it by initializing a variable for it.

1. s = turtle.getscreen()

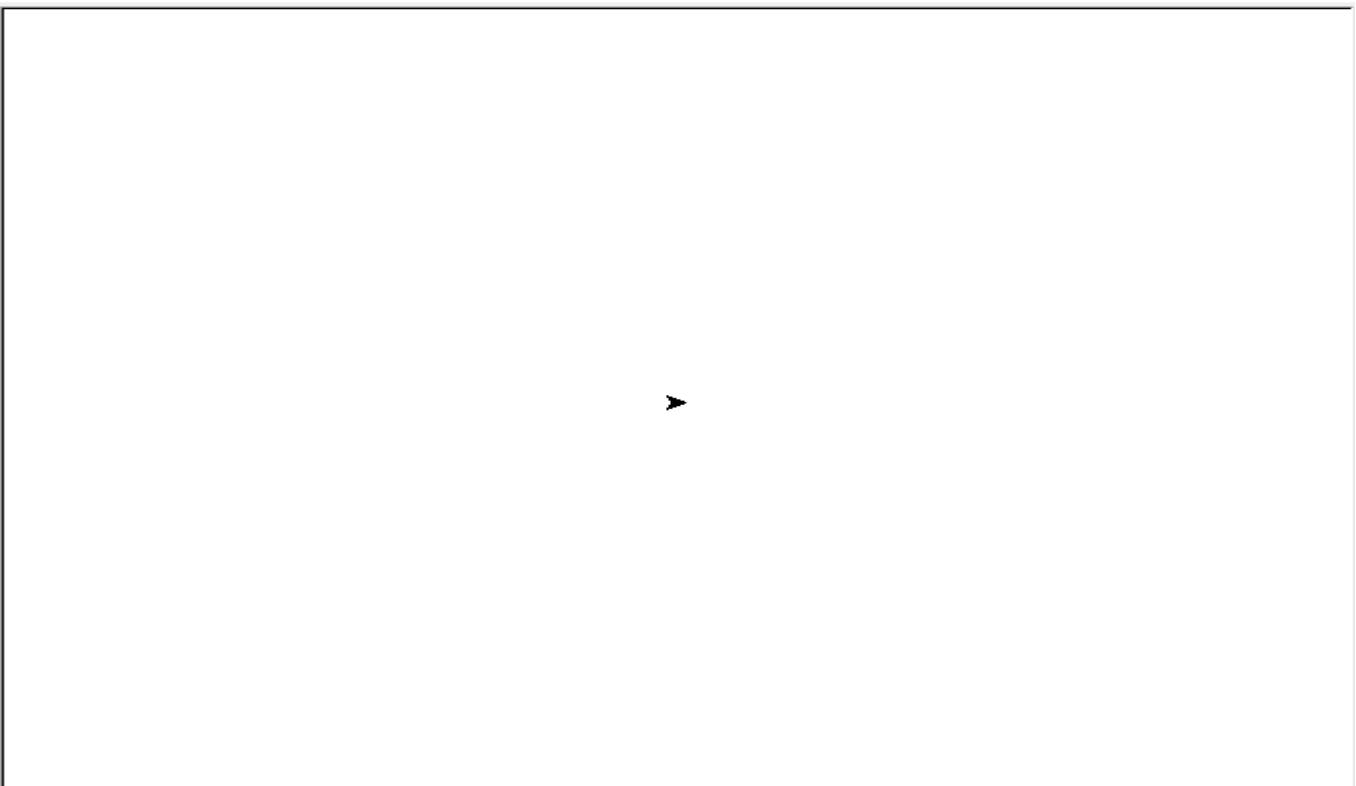


It will look like an above image and the little triangle in the middle of the screen is a turtle. If the screen is not appearing in your computer system, use the below code.

Example -

```
import turtle  
# Creating turtle screen  
s = turtle.getscreen()  
# To stop the screen to display  
turtle.mainloop()
```

Output:



The screen same as the canvas and turtle acts like a pen. You can move the turtle to design the desired shape. The turtle has certain changeable features such as color, speed, and size. It can be moved to a specific direction, and move in that direction unless we tell it otherwise.

In the next section, we will learn to program with the Python turtle library.

Programming with turtle

First, we need to learn to move the turtle all direction as we want. We can customize the pen like turtle and its environment. Let's learn the couple of commands to perform a few specific tasks.

Turtle can be moved in four directions.

- Forward
- Backward
- Left
- Right

Turtle motion

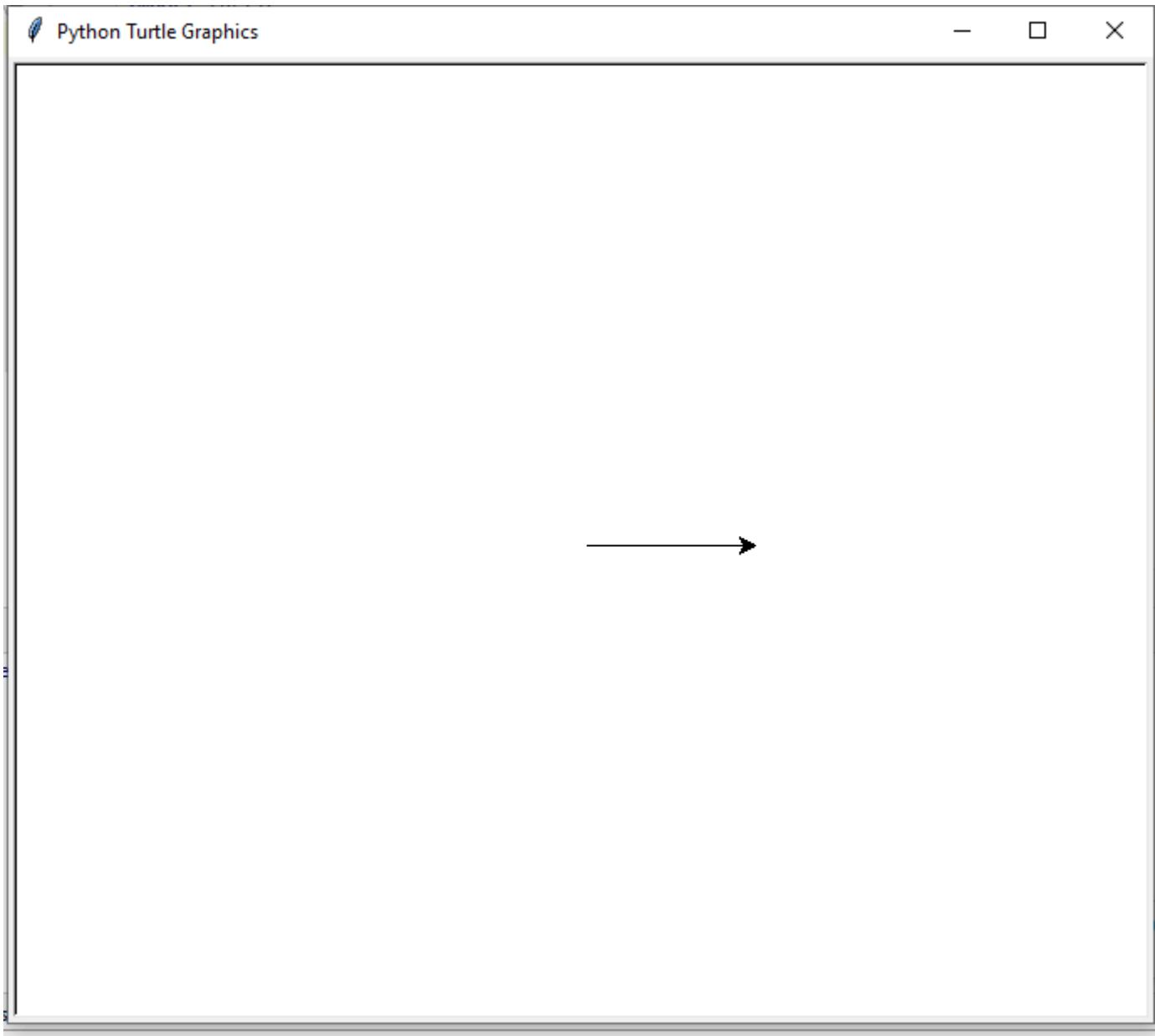
The turtle can move forward and backward in direction that it's facing. Let's see the following functions.

- o **forward(*distance*) or turtle.fd(*distance*)** - It moves the turtle in the forward direction by a certain distance. It takes one parameter **distance**, which can be an integer or float.

Example - 3:

```
import turtle  
# Creating turtle screen  
t = turtle.Turtle()  
# To stop the screen to display  
t.forward(100)  
turtle.mainloop()
```

Output:



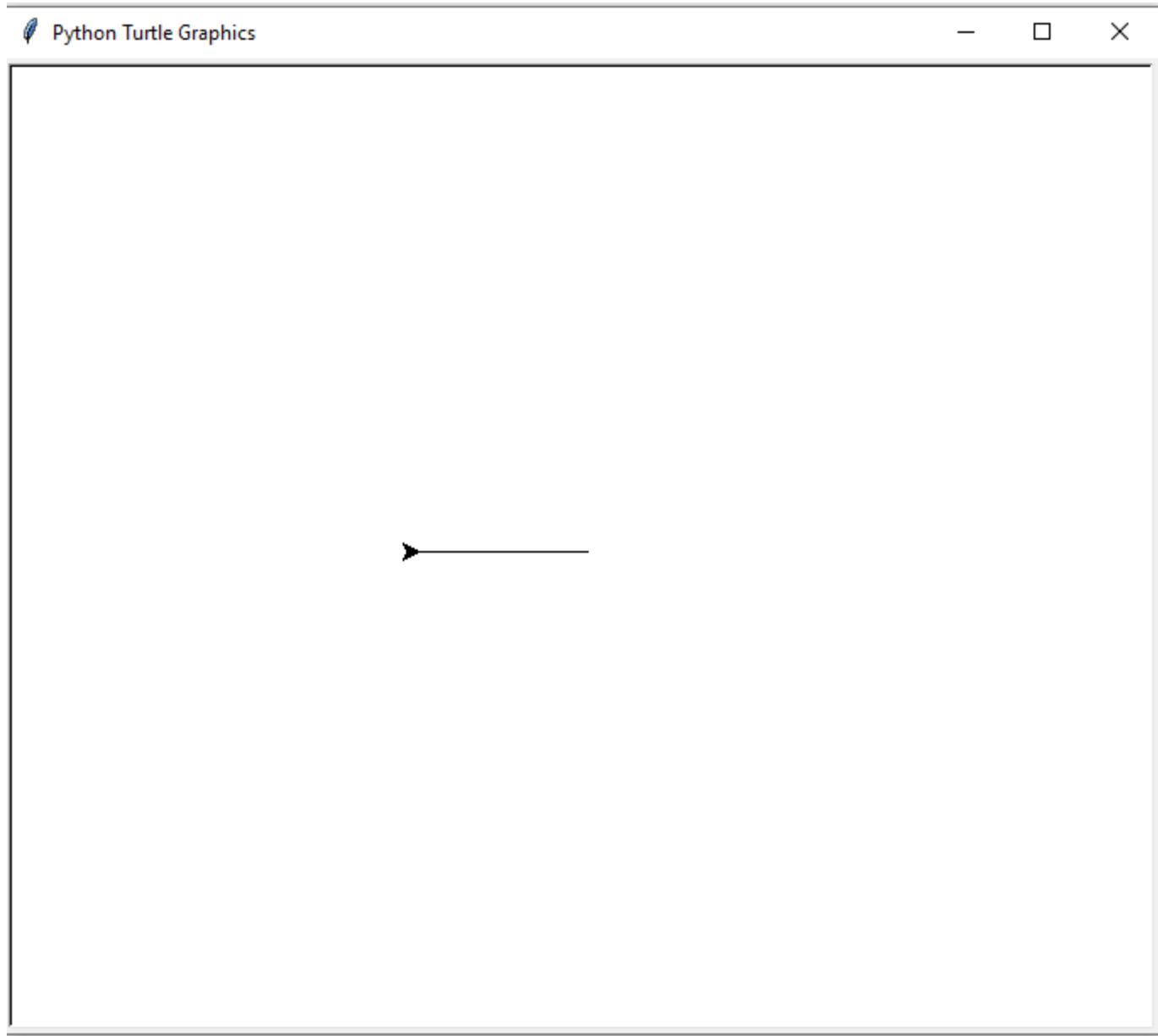
- **back(distance)** or **turtle.bk** or **turtle.backward(distance)** - This method moves the turtle in the opposite direction the turtle is headed. It doesn't change the turtle heading.

Example - 2:

```
import turtle  
# Creating turtle screen  
t = turtle.Turtle()  
# Move turtle in opposite direction  
t.backward(100)  
# To stop the screen to display
```

```
turtle.mainloop()
```

Output:



- **right(angle)** or **turtle.rt(angle)** - This method moves the turtle right by **angle** units.

Example - 3:

```
import turtle  
# Creating turtle screen  
t = turtle.Turtle()  
t.heading()  
# Move turtle in opposite direction
```

```
t.right(25)
t.heading()
# To stop the screen to display
turtle.mainloop()
```

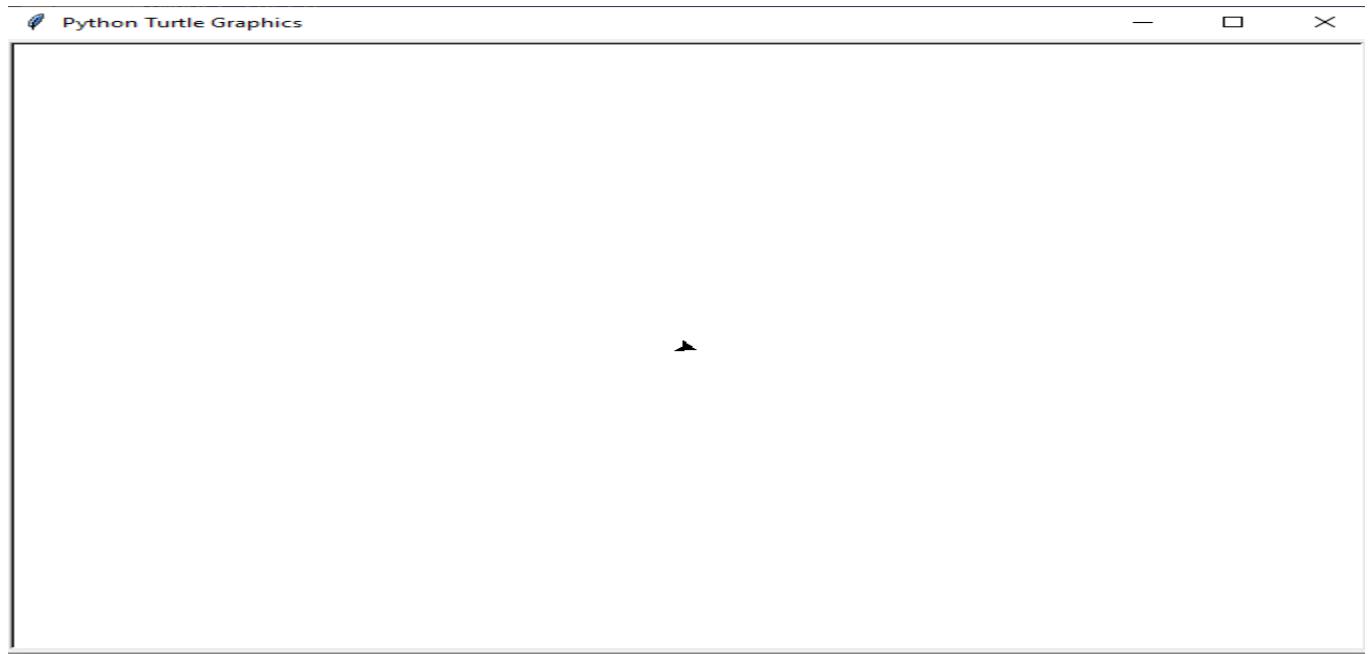
Output:

- **left(angle)** or **turtle.lt(angle)** - This method turn the turtle left by **angle** units. Let's understand the following example.

Example -

```
import turtle
# Creating turtle screen
t = turtle.Turtle()
t.heading()
# Move turtle in left
t.left(100)
t.heading()
# To stop the screen to display
turtle.mainloop()
```

Output:



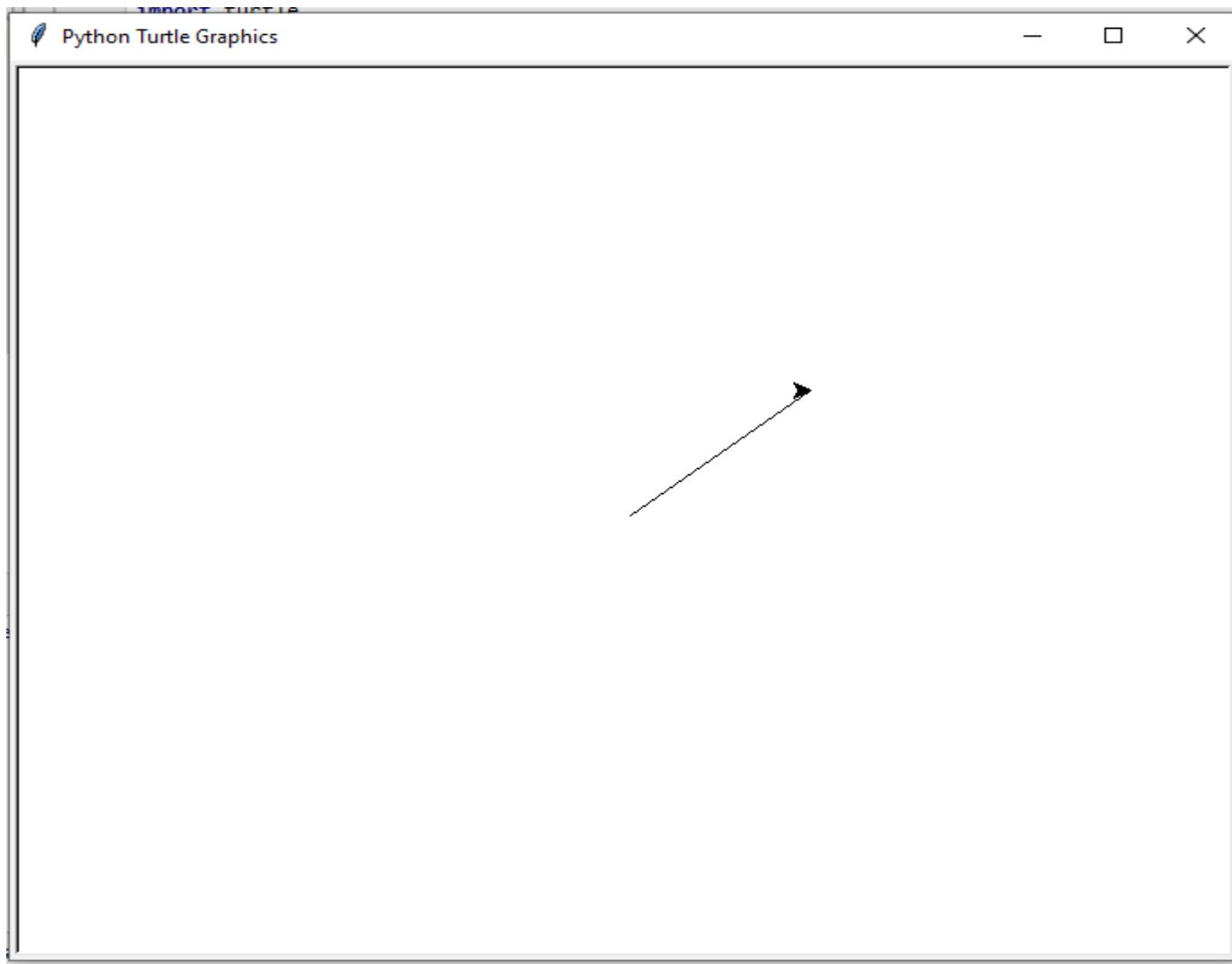
The screen is initially divided into four quadrants. The turtle is positioned at the beginning of the program is (0,0) known as the **Home**.

- **goto(x, y=None) or turtle.setpos(x, y=None) turtle.setposition(x, y=None)** - This method is used to move the turtle in the other area on the screen. It takes the two coordinates - **x and y**. Consider the following example.

Example -

```
import turtle  
# Creating turtle screen  
t = turtle.Turtle()  
# Move turtle with coordinates  
t.goto(100, 80)  
# To stop the screen to display  
turtle.mainloop()
```

Output:



Drawing a Shape

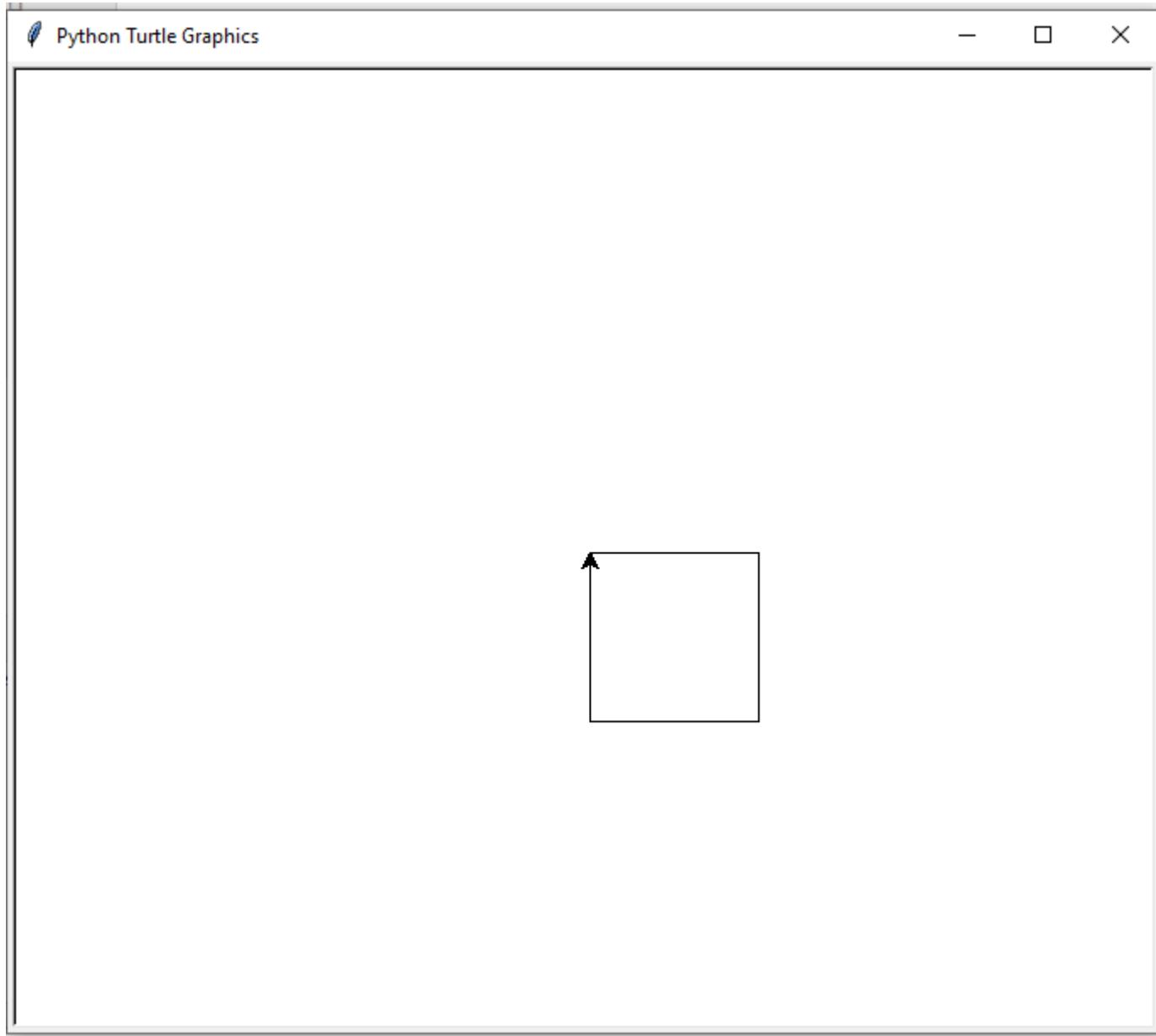
We discussed the movement of the turtle. Now, we learn to move on to making actual shape. First, we draw the **polygon** since they all consist of straight lines connected at the certain angles. Let's understand the following example.

Example -

1. t.fd(100)
2. t.rt(90)
3. t.fd(100)
4. t.rt(90)
5. t.fd(100)
6. t.rt(90)
7. t.fd(100)

It will look like the following image.

Output:



We can draw any shape using the turtle, such as a rectangle, triangle, square, and many more. But, we need to take care of coordinate while drawing the rectangle because all four sides are not equal. Once we draw the rectangle, we can even try creating other polygons by increasing number of side.

Drawing Preset Figures

Suppose you want to draw a **circle**. If you attempt to draw it in the same way as you drew the square, it would be extremely tedious, and you'd have to spend a lot of time just for that one shape. Thankfully, the Python turtle library provides a solution for this. You can use a single command to draw a circle.

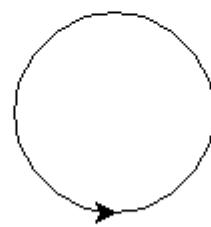
- **circle(radius, extent = None, steps = an integer)** - It is used to draw the circle to the screen. It takes three arguments.
 1. **radius** - It can be a number.
 2. **extent** - It can be a number or None.
 3. **steps** - It can be an integer.

The circle is drawn with the given radius. The extent determines which part of circle is drawn and if the extent is not provided or none, then draw the entire circle. Let's understand the following example.

Example -

1. **import** turtle
2. # Creating turtle screen
3. t = turtle.Turtle()
- 4.
5. t.circle(50)
- 6.
7. turtle.mainloop()

Output:



We can also draw a dot, which is also known as a filled-in circle. Follow the given method to draw a filled-in circle.

Example -

1. **import** turtle
2. **# Creating turtle screen**
3. t = turtle.Turtle()
- 4.
5. t.dot(50)
- 6.
7. turtle.mainloop()

Output:



The number we have passed in the **dot()** function is diameter of the dot. We can increase and decrease the size of the dot by changing its diameter.

So far, we have learned movement of turtle and design the various shapes. In the next few sections, we will learn the customization of turtle and its environment.

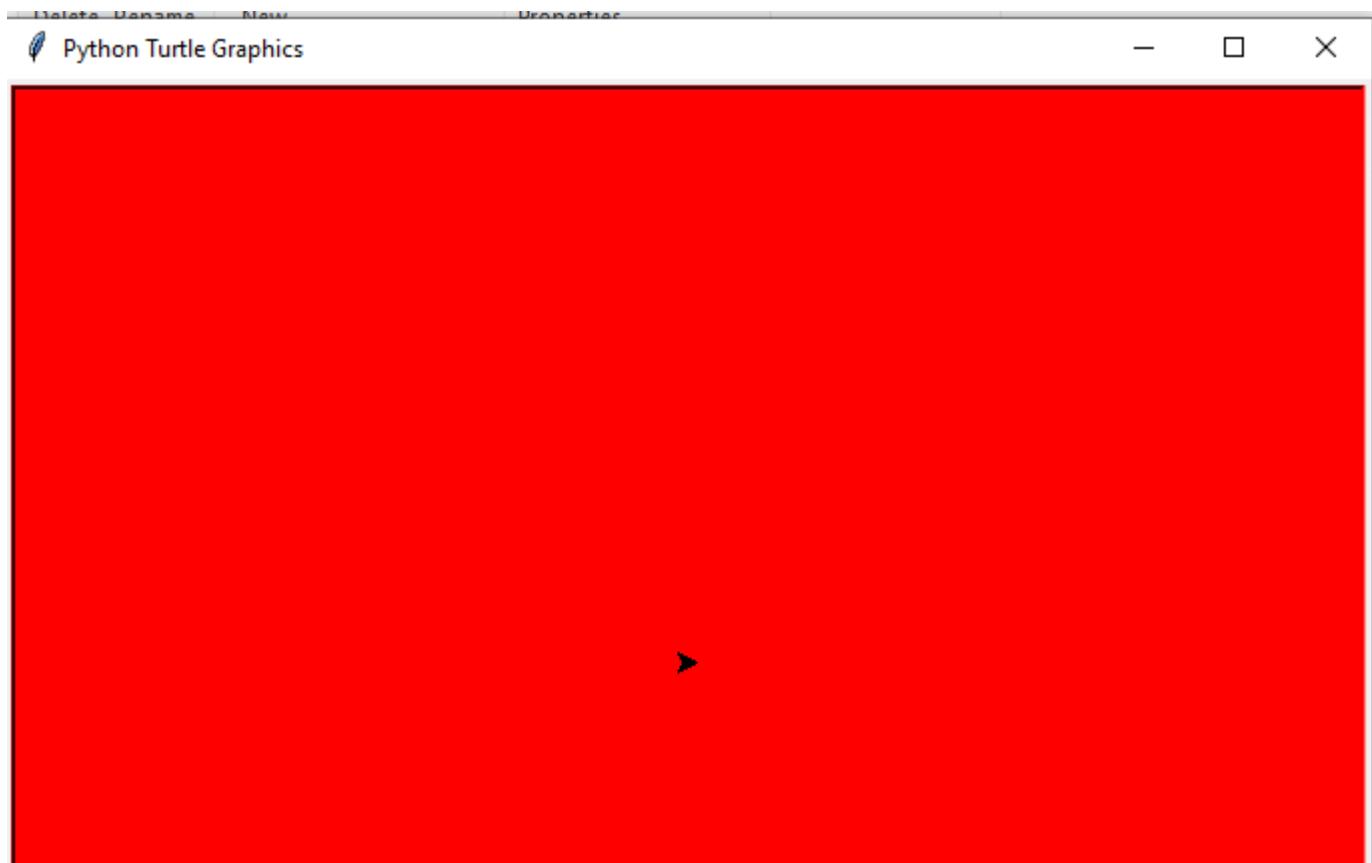
Changing the Screen Color

By default, the turtle screen is opened with the white background. However, we can modify the background color of the screen using the following function.

Example -

1. **import** turtle
2. # Creating turtle screen
3. t = turtle.Turtle()
- 4.
5. turtle.bgcolor("red")
- 6.
7. turtle.mainloop()

Output:



We have passed a red color. We can also replace it with any color or we can use the hex code to use variety of code for our screen.

Adding Image to the background

Same as the screen background color, we can add the background image using the following function.

- **bgpic (picname=None)** - It sets the background image or return name of current background image. It takes one argument picname which can be a string, name of a gif-file or "**nopic**" or "**none**". If the picname is "**nopic**", delete the background image. Let's see the following example.

Example -

1. **import** turtle
2. # Creating turtle turtle
3. t = turtle.Turtle()
- 4.
5. turtle.bgpic()
- 6.

```
7. turtle.bgpic(r"C:\Users\DEVANSH SHARMA\Downloads\person.jpg")
8. turtle.bgpic()
9.
10. turtle.mainloop()
```

Changing the Image Size

We can change the image size using the **screensize()** function. The syntax is given below.

Syntax -

```
1. turtle.screensize(canvwidth = None, canvheight = None, bg = None)
```

Parameter - It takes three parameters.

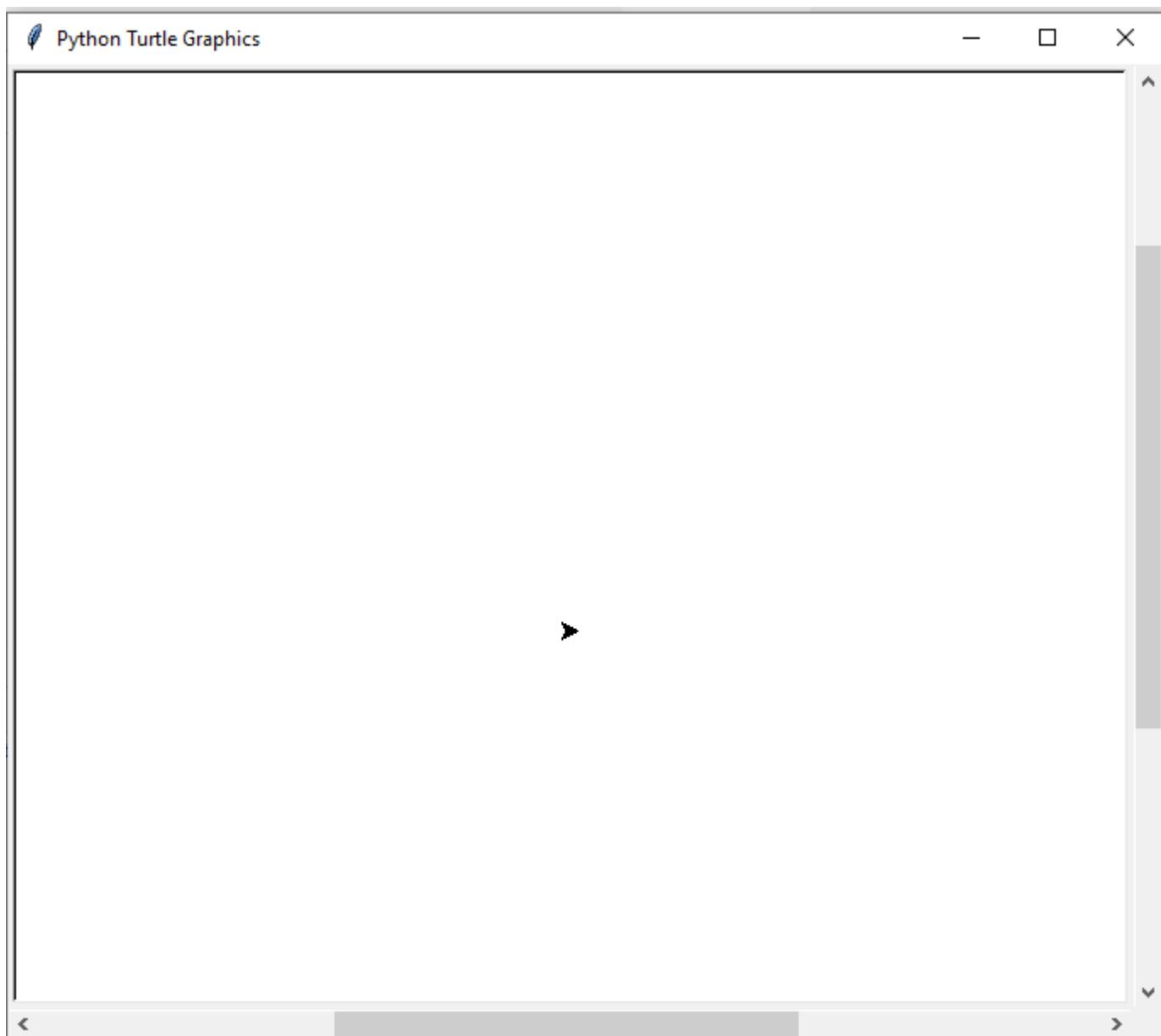
- **canvwidth** - It can be a positive number, new canvas width in pixel.
- **canvheight** - It can be a positive number, new height of the canvas in pixel.
- **bg** - It is colorstring or color-tuple. New background color.

Let's understand the following example.

Example -

```
1. import turtle
2. # Creating turtle turtle
3. t = turtle.Turtle()
4.
5. turtle.screensize()
6.
7. turtle.screensize(1500,1000)
8. turtle.screensize()
9.
10. turtle.mainloop()
```

Output:



Changing the Screen Title

Sometimes, we want to change the title of the screen. By default, it shows the *Python tutorial graphics*. We can make it personal such as "**My First Turtle Program**" or "**Drawing Shape with Python**". We can change the title of the screen using the following function.

1. `turtle.Title("Your Title")`

Let's see the example.

Example -

1. `import turtle`
2. `# Creating turtle`

```
3. t = turtle.Turtle()  
4.  
5. turtle.title("My Turtle Program")  
6.  
7. turtle.mainloop()
```

Output:



You can change the screen title according to preference.

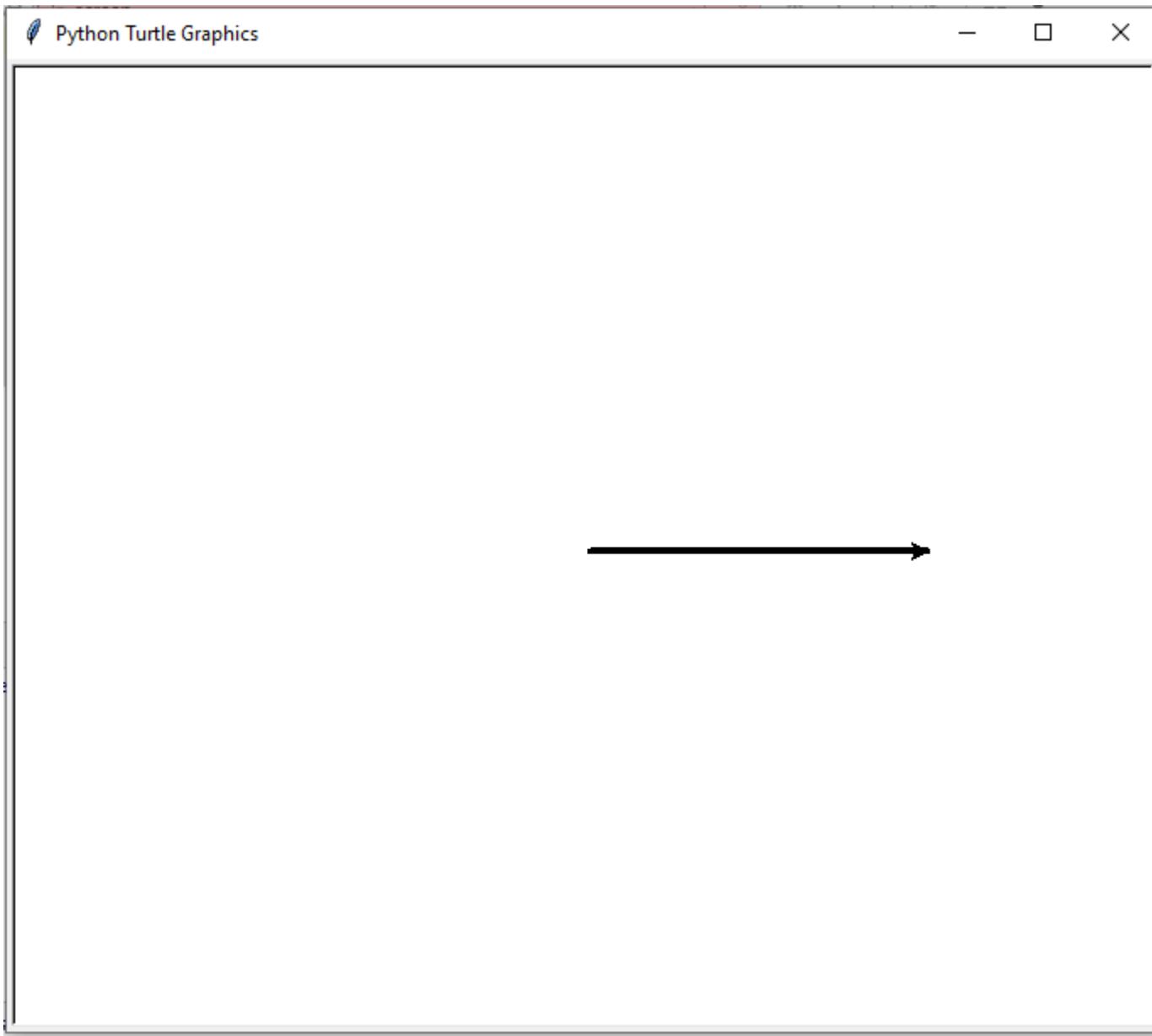
Changing the Pen Size

We can increase or decrease the turtle's size according the requirement. Sometimes, we need thickness in the pen. We can do this using the following example.

Example -

```
1. import turtle  
2. # Creating turtle turtle  
3. t = turtle.Turtle()  
4.  
5. t.pensize(4)  
6. t.forward(200)  
7.  
8. turtle.mainloop()
```

Output:



As we can see in the above image, the pen is four times the original size. We can use it draw lines of various sizes.

Pen Color Control

By default, when we open a new screen, the turtle comes up with the black color and draws with black ink. We can change it according the two things.

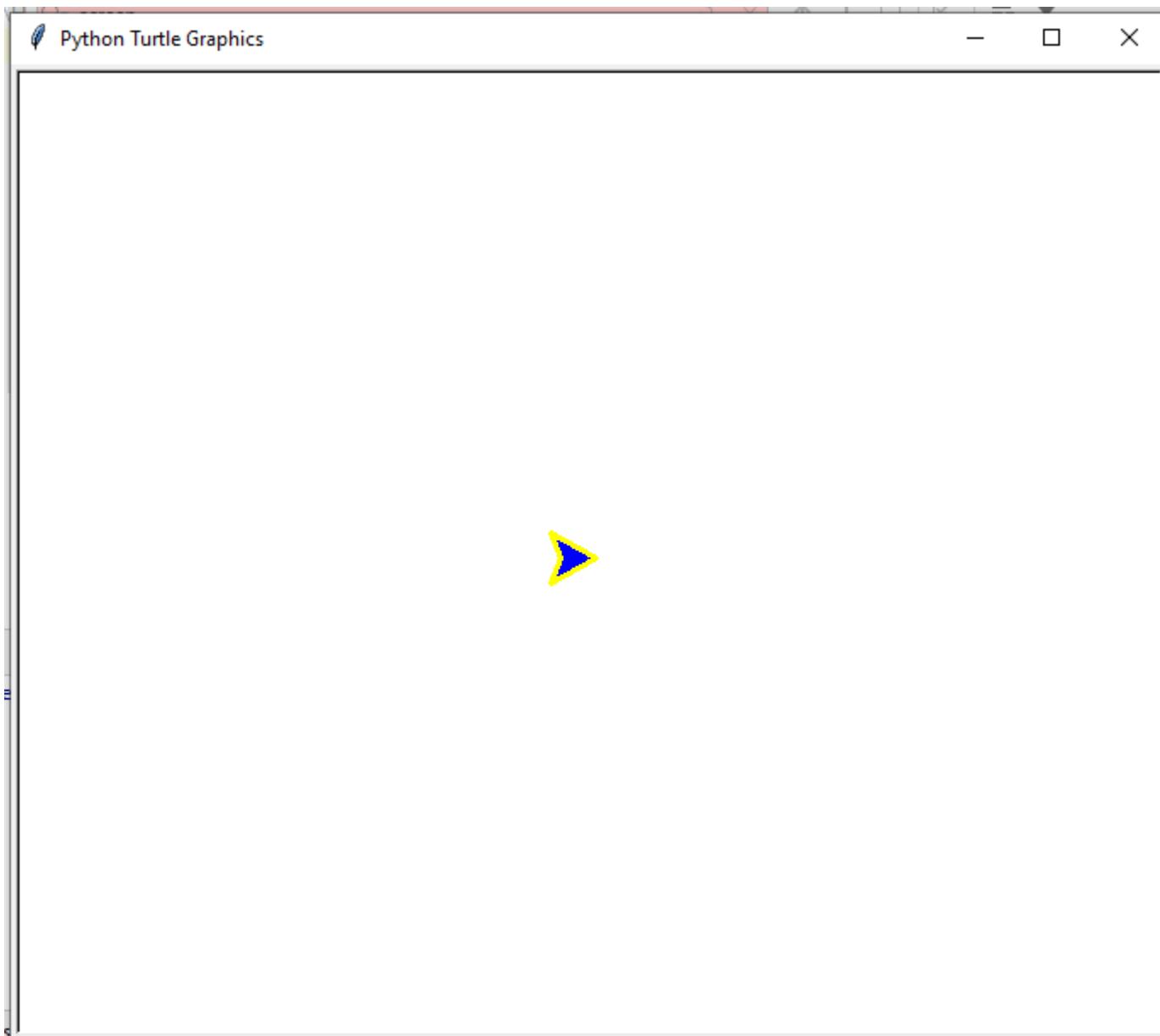
- We can change the color of the turtle, which is a fill color.
- We can change the pen's color, which basically a change of the outline or the ink color.

We can also change both the pen color and turtle color if we want. We suggest increasing the size of the turtle that changes in the color can be clearly visible. Let's understand the following code.

Example -

```
1. import turtle  
2. # Creating turtle turtle  
3. t = turtle.Turtle()  
4.  
5. # Increase the turtle size  
6. t.shapesize(3,3,3)  
7.  
8. # fill the color  
9. t.fillcolor("blue")  
10.  
11. # Change the pen color  
12. t.pencolor("yellow")  
13.  
14. turtle.mainloop()
```

Output:



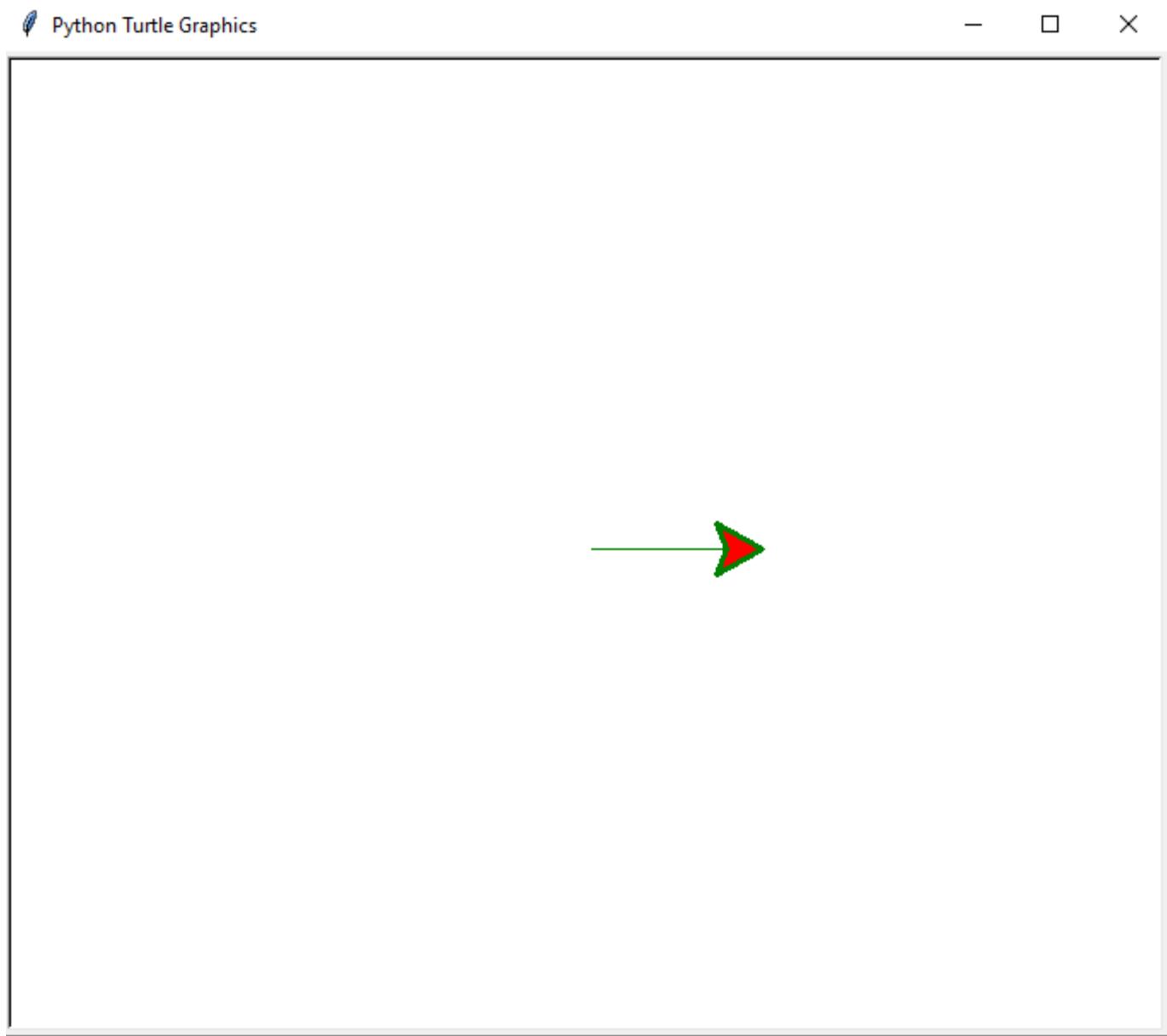
Type the following function to change the color of both.

Example - 2:

1. **import** turtle
2. # Creating turtle turtle
3. t = turtle.Turtle()
- 4.
5. t.shapesize(3,3,3)
- 6.
7. # Chnage the color of both
8. t.color("green", "red")

9.
10. t.forward(100)
11.
12. turtle.mainloop()

Output:



Explanation:

In the above code, the first color is a pen color and second is a fill color.

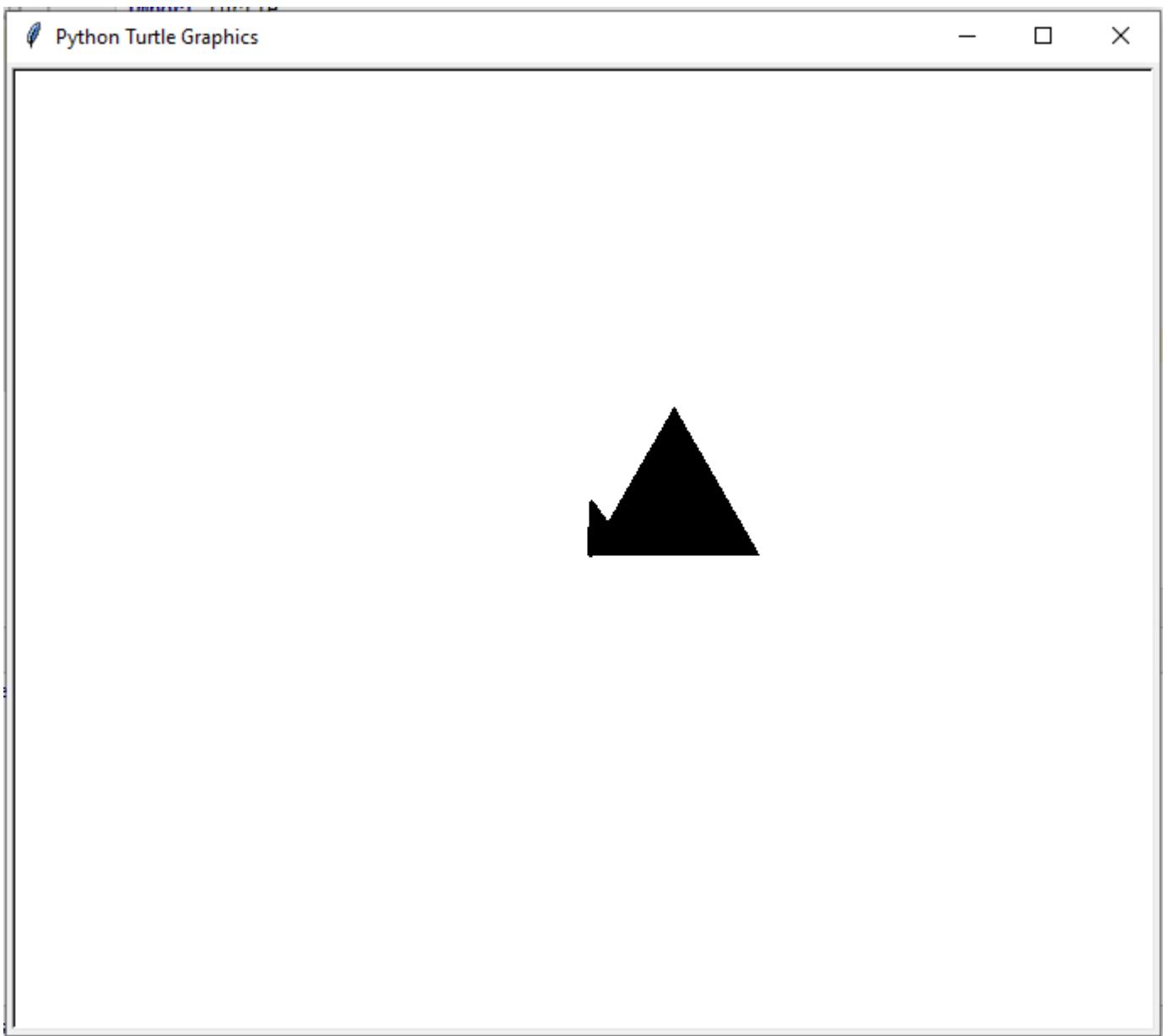
Turtle fill in the Image

Colors make an image or shapes very attractive. We can fill shapes with various colors. Let's understand the following example to add color to the drawings. Let's understand the following example.

Example -

```
1. import turtle  
2. # Creating turtle turtle  
3. t = turtle.Turtle()  
4.  
5. t.shapesize(3,3,3)  
6.  
7. t.begin_fill()  
8. t.fd(100)  
9. t.lt(120)  
10. t.fd(100)  
11. t.lt(120)  
12. t.fd(100)  
13. t.end_fill()  
14.  
15. turtle.mainloop()
```

Output:



Explanation:

When the program executes, it draw first the triangle and then filled it with the solid black color as the above output. We have used the **.begin_fill()** method which indicates that we will draw a closed shape to be filled. Then, we use the **.end_fill()**, which indicates that we have done with the creating shape. Now, it can be filled with color.

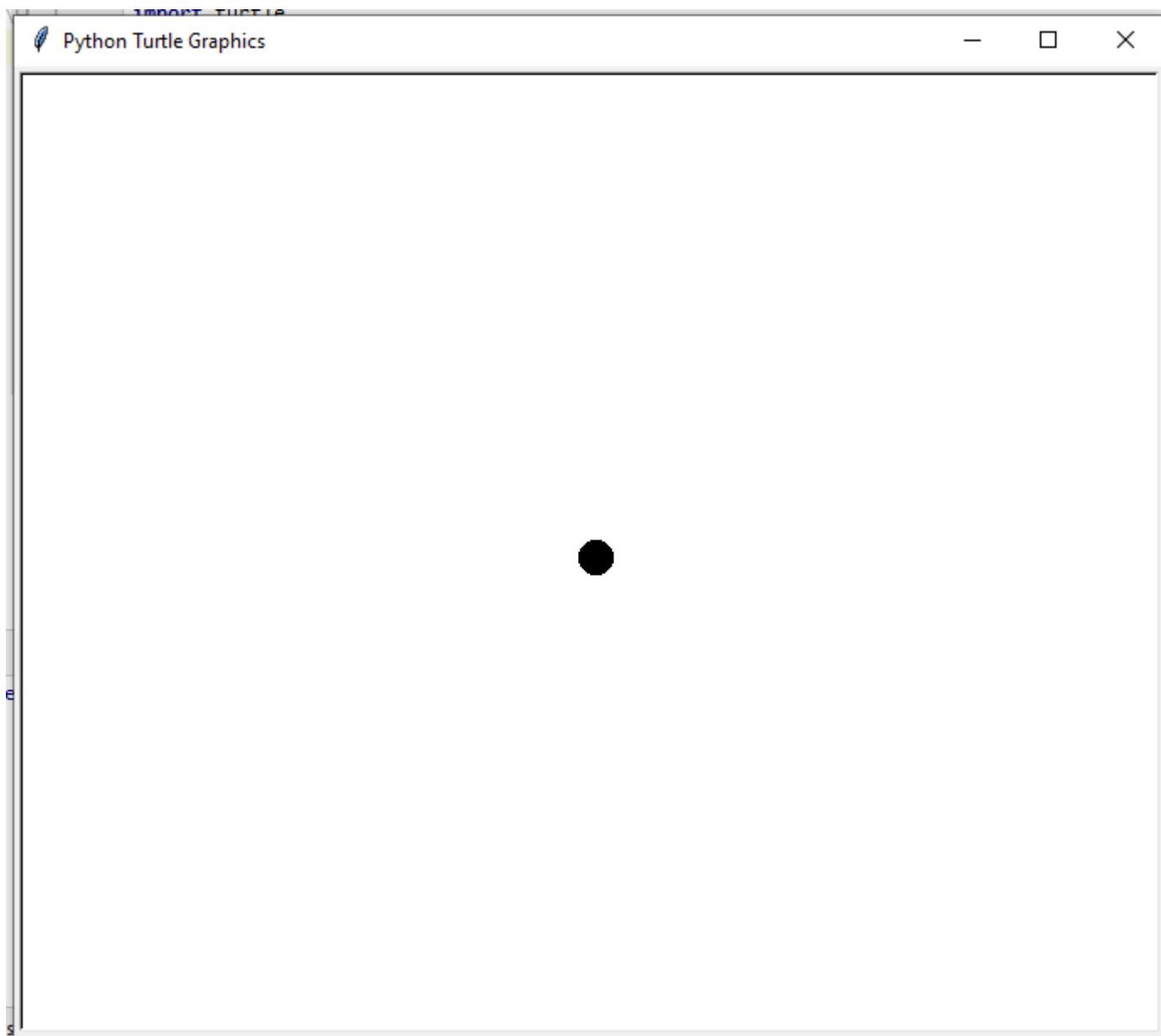
Changing the Turtle Shape

By default, the turtle shape is triangular. However, we can change the turtle's shape and this module provides many shapes for the turtle. Let's understand the following example.

Example -

```
1. import turtle  
2. # Creating turtle turtle  
3. t = turtle.Turtle()  
4.  
5. t.shape("turtle")  
6. # Change to arrow  
7. t.shape("arrow")  
8. # Chnage to circle  
9. t.shape("circle")  
10.  
11. turtle.mainloop()
```

Output:



We can change the turtle shape according to the requirement. These shapes can be a square, triangle, classic, turtle, arrow and circle. The **classic** is the original shape of the turtle.

Changing the Pen Speed

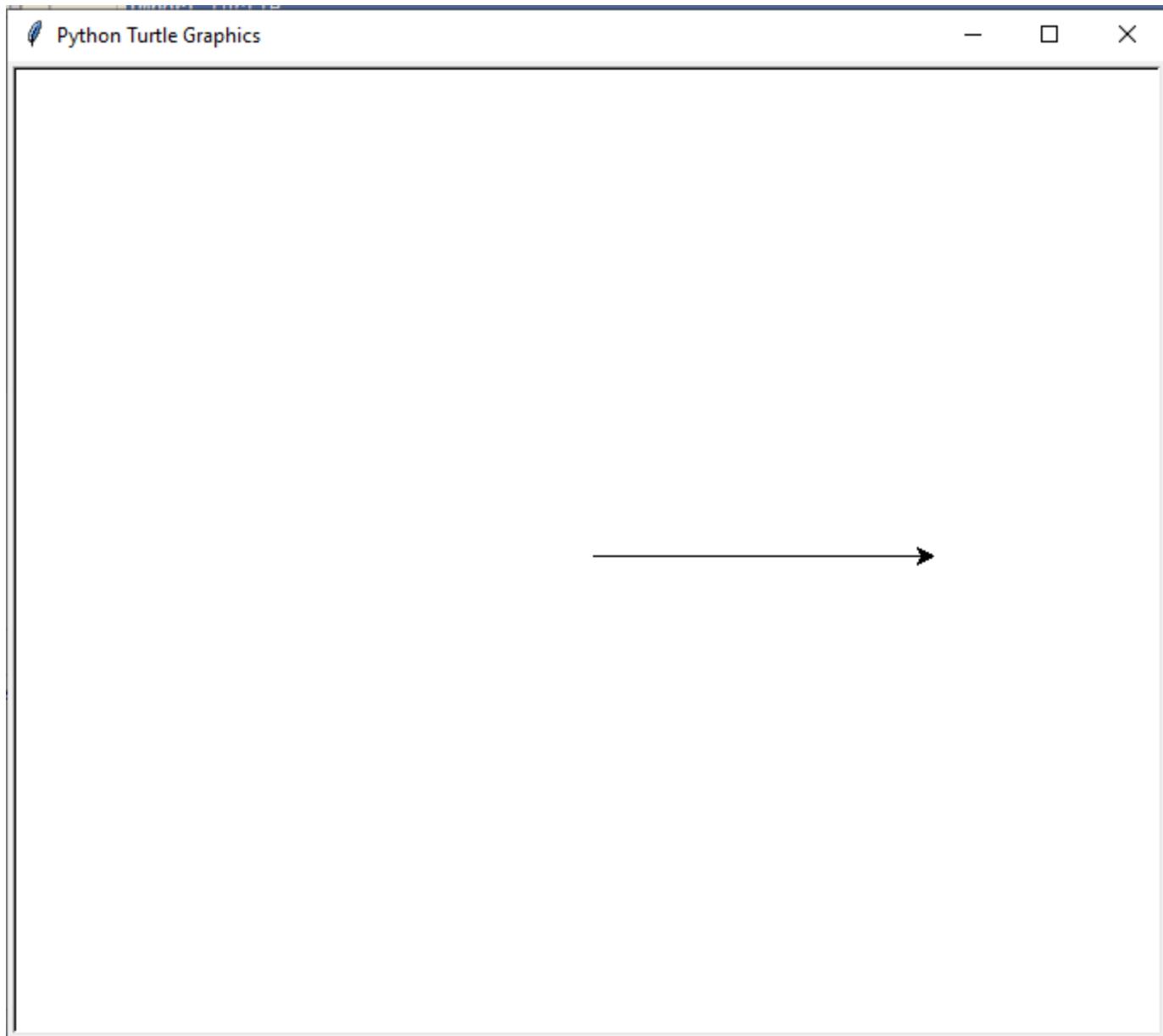
The speed of the turtle can be changed. Generally, it moves at a moderate speed over the screen but we can increase and decrease its speed. Below is the method to modify the turtle speed.

Example -

1. **import** turtle
2. # Creating turtle
3. t = turtle.Turtle()
- 4.

```
5. t.speed(3)
6. t.forward(100)
7. t.speed(7)
8. t.forward(100)
9.
10. turtle.mainloop()
```

Output:



The turtle speed can vary integer values in the range 0...10. No argument is passed in the **speed()** function, it returns the current speed. Speed strings are mapped to speed values as follows.

0	Fastest
10	Fast
6	Normal
3	Slow
1	Slowest

| Note - If speed is assigned to zero means no animation will take place.

1. turtle.speed()
2. turtle.speed('normal')
3. turtle.speed()
- 4.
5. turtle.speed(9)
6. turtle.speed()

Customization in One line

Suppose we want multiple changes within the turtle; we can do it by using just one line. Below are a few characteristics of the turtle.

- The pen color should be red.
- The fill color should be orange.
- The pen size should be 10.
- The pen speed should be 7
- The background color should be blue.

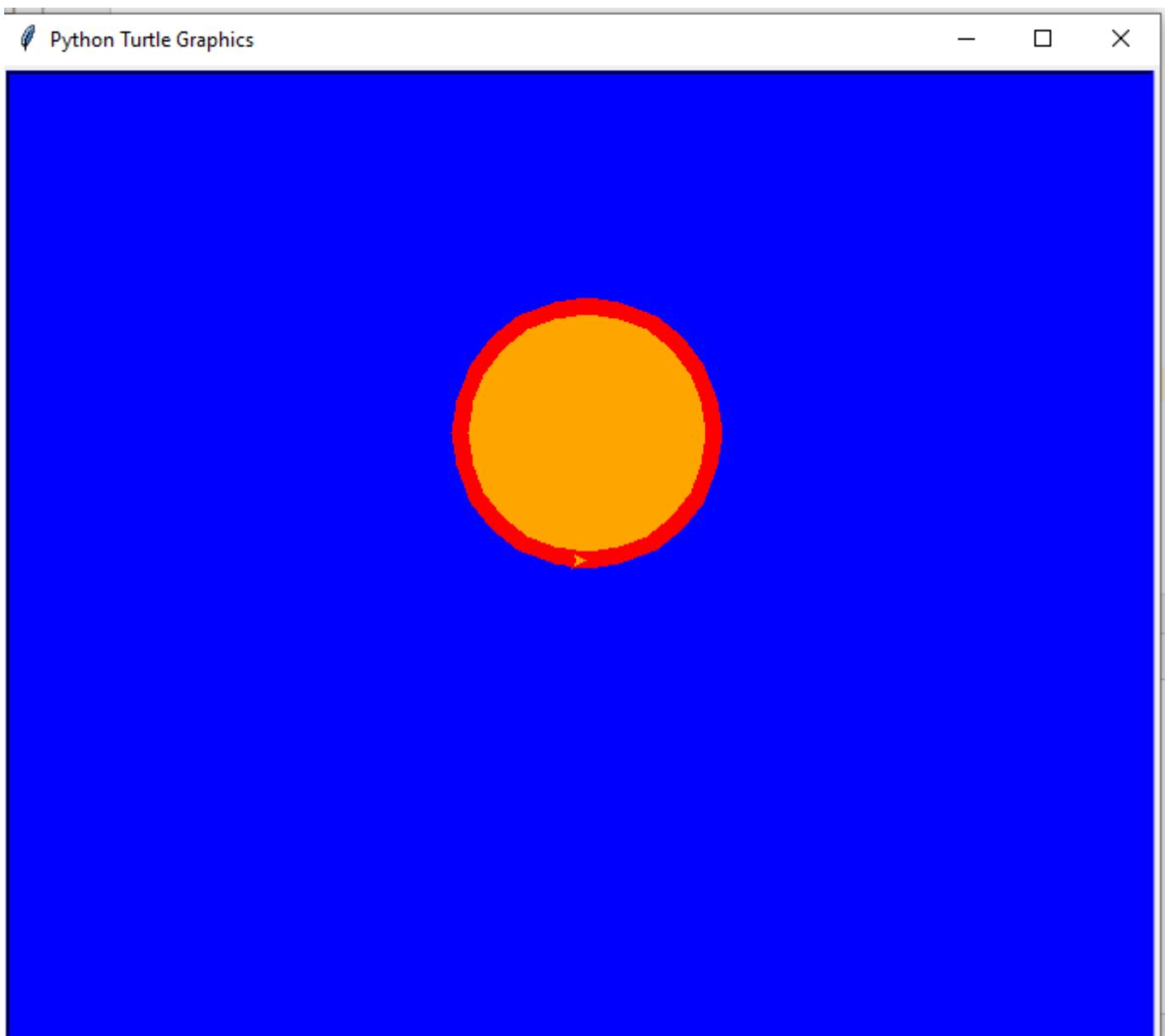
Let's see the following example.

Example -

1. **import** turtle
2. # Creating turtle
3. t = turtle.Turtle()
- 4.
5. t.pencolor("red")
6. t.fillcolor("orange")
7. t.pensize(10)

```
8. t.speed(7)
9. t.begin_fill()
10. t.circle(75)
11. turtle.bgcolor("blue")
12. t.end_fill()
13.
14. turtle.mainloop()
```

Output:



We used just one line and changed the turtle's characteristics. To learn about this command, you can learn from the **library official documentation**.

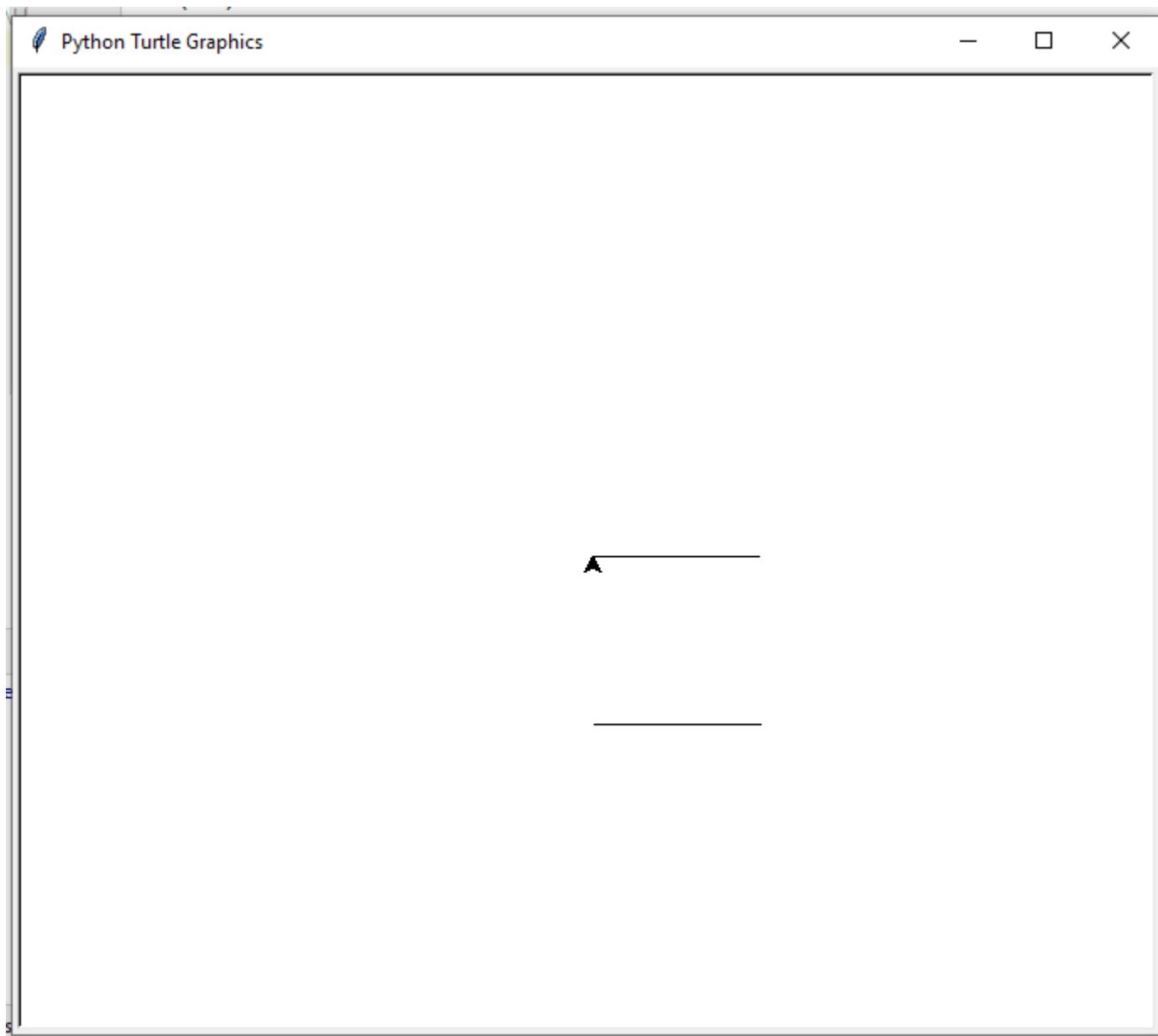
Change the Pen Direction

By default, the turtle points to the right on the screen. Sometimes, we require moving the turtle to the other side of the screen itself. To accomplish this, we can use the **penup()** method. The **pendown()** function uses to start drawing again. Consider the following example.

Example -

```
1. import turtle  
2. # Creating turtle  
3. t = turtle.Turtle()  
4.  
5. t.fd(100)  
6. t.rt(90)  
7. t.penup()  
8. t.fd(100)  
9. t.rt(90)  
10. t.pendown()  
11. t.fd(100)  
12. t.rt(90)  
13. t.penup()  
14. t.fd(100)  
15. t.pendown()  
16.  
17. turtle.mainloop()
```

Output:



As we can see in the above output, we have obtained two parallel lines instead of a square.

Clearing Screen

We have covered most of designing concepts of the turtle. Sometimes, we need a clear screen to draw more designs. We can do it using the following function.

1. `t.clear()`

The above method will clear the screen so that we can draw more designs. This function only removes the existing designs or shapes not make any changes in variable. The turtle will remain in the same position.

Resetting the Environment

We can also reset the current working using the reset function. It restores the **turtle's** setting and clears the screen. We just need to use the following function.

1. t.reset

All tasks will be removed and the turtle back to its home position. The default settings of turtle, such as color, size, and shape and other features will be restored.

We have learned the basic fundamental of the turtle programming. Now, we will discuss a few essential and advanced concepts of the turtle library.

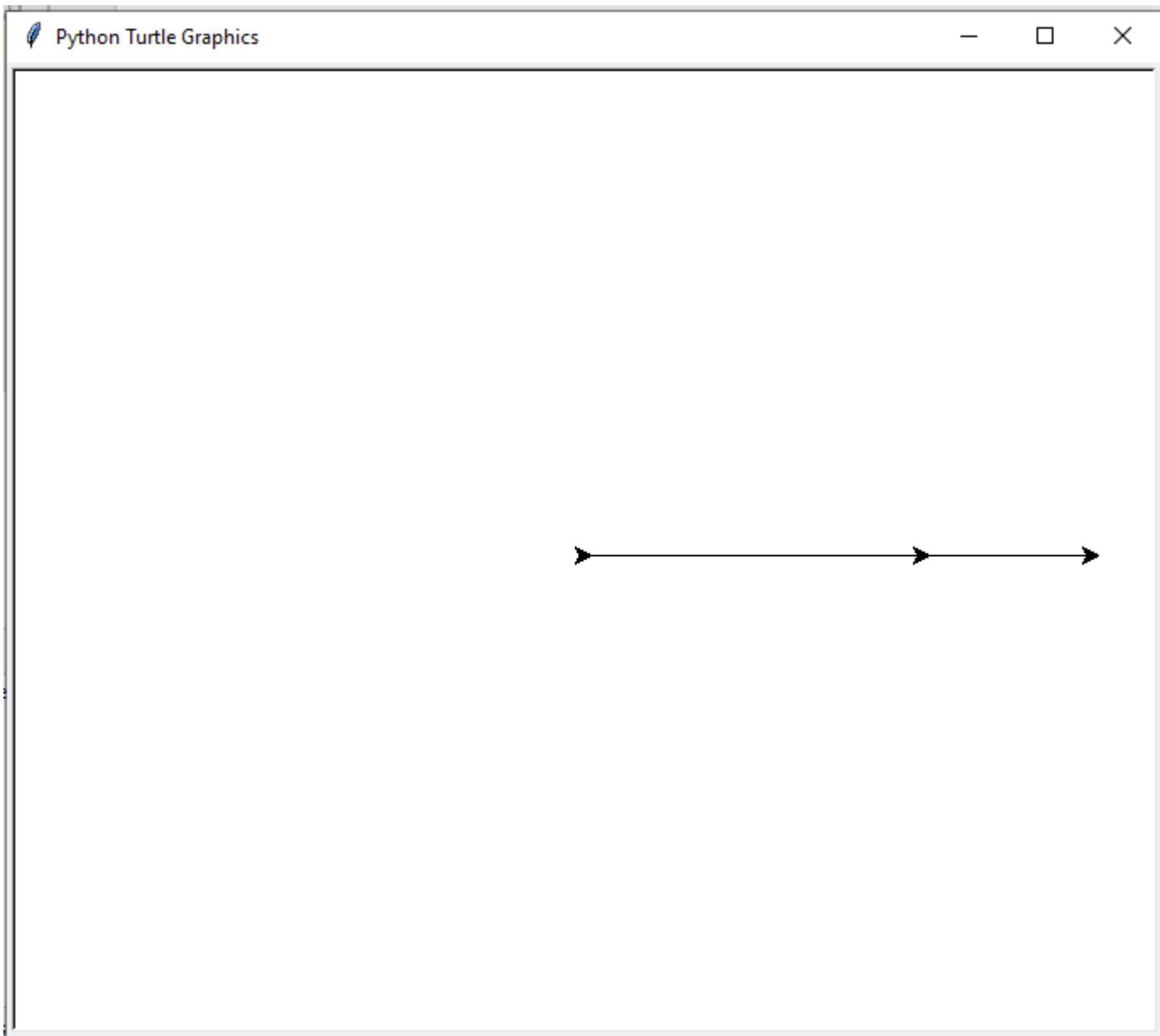
Leaving a Stamp

We can leave the stamp of turtle on the screen. The stamp is nothing but an imprint of the turtle. Let's understand the following example.

Example -

1. **import** turtle
2. # Creating turtle
3. t = turtle.Turtle()
4. t.stamp()
- 5.
6. t.fd(200)
7. t.stamp()
- 8.
9. t.fd(100)
- 10.
11. turtle.mainloop()

Output:



If we print the **stamp()** method, it will display a number which is nothing but a turtle's location or stamp ID. We can also remove a particular stamp by using the following command.

1. `t.clearstamp(8) # 8 is a stamp location.`

Cloning of a turtle

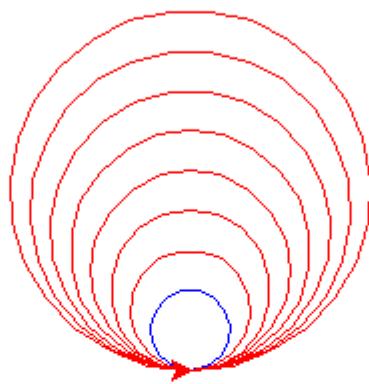
Sometimes, we look for the multiple turtle to design a unique shape. It provides the facility to clone the current working turtle into the environment and we can move both turtle on the screen. Let's understand the following example.

Example -

1. `import turtle`

```
2. # Creating turtle
3. t = turtle.Turtle()
4.
5. c = t.clone()
6. t.color("blue")
7. c.color("red")
8. t.circle(20)
9. c.circle(30)
10. for i in range(40, 100, 10):
11.     c.circle(i)
12.
13. turtle.mainloop()
```

Output:

**Explanation:**

In the above code, we cloned the turtle to the `c` variable and called the `circle` function. First, it draws the blue circle and then draw the outer circles based on the for loop conditions.

In the next section, we will discuss how we can use Python conditional and loop statements to create design using the turtle.

Turtle Programming Using Loops and Conditional Statements

We have learned the basic and advanced concepts of the turtle library so far. The next step is to explore those concepts with Python's loops and conditional statements. It will give us a

practical approach when it comes to an understanding of these concepts. Before moving further, we should remember the following concepts.

- **Loops** - These are used to repeat a set of code until a particular condition is matched.
- **Conditional Statements** - These are used to perform a task based on specific conditions.
- **Indentation** - It is used to define a block of code and it is essential when we are using loops and conditional statements. Indentation is nothing but a set of whitespaces. The statements that are on the same level are considered as the same block statements.

Let's understand the following examples.

for loops

In the previous example, we wrote multiple repeated lines in our code. Here, we will implement create a square program using for loop. For example -

Example:

1. t.fd(100)
2. t.rt(90)
3. t.fd(100)
4. t.rt(90)
5. t.fd(100)
6. t.rt(90)
7. t.fd(100)
8. t.rt(90)

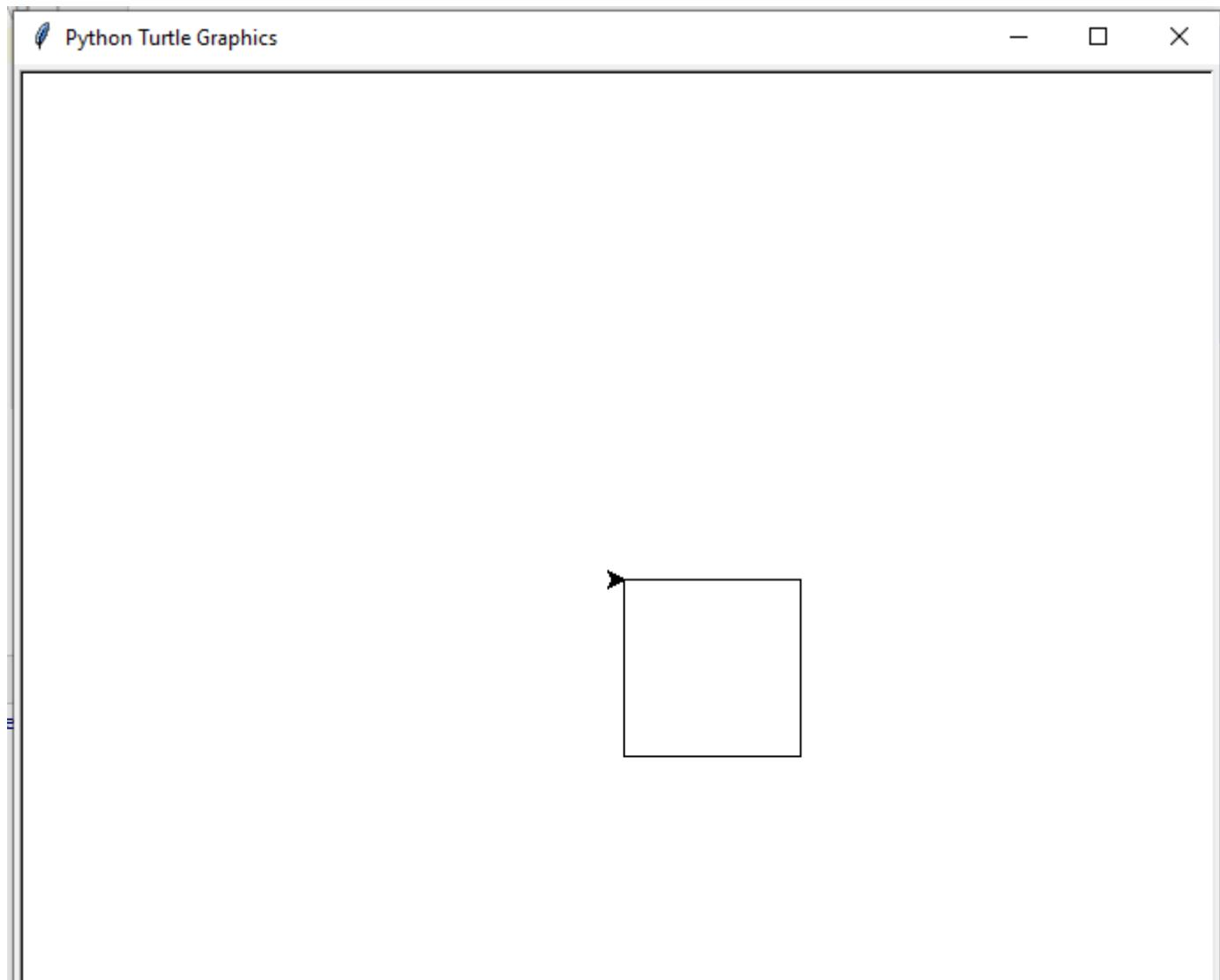
We can make it shorter using a for loop. Run the below code.

Example

1. **import** turtle
2. **# Creating turtle**
3. t = turtle.Turtle()
- 4.
5. **for i in range(4):**
6. t.fd(100)
7. t.rt(90)
- 8.
- 9.

10. turtle.mainloop()

Output:



Explanation

In the above code, for loop repeated the code until it reached at counter 4. The i is like a counter that starts from zero and keep increasing by one. Let's understand the above loop execution step by step.

- In the first iteration, $i = 0$, the turtle moves forward by 100 units and then turns 90 degrees to the right.
- In the second iteration, $i = 1$, the turtle moves forward by 100 units and then turns 90 degrees to the right.
- In the third iteration, $i = 2$, the turtle moves forward by 100 units and then turns 90 degrees to the right.

- In the third iteration, i = 3, the turtle moves forward by 100 units and then turns 90 degrees to the right.

After completing the iteration, the turtle will jump out of the loop.

while loops

It is used to run a block of code until a condition is satisfied. The code will be terminated when it finds a false condition. Let's understand the following example.

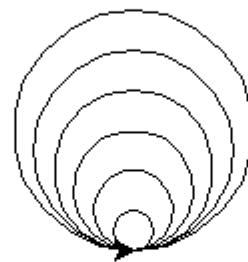
Example -

1. **import** turtle
2. # Creating turtle
3. t = turtle.Turtle()
- 4.
5. n=10
6. **while** n <= 60:
7. t.circle(n)
8. n = n+10
- 9.
- 10.
11. turtle.mainloop()

Output:

Python Turtle Graphics

- □ ×



As we can see in the output, we draw multiple circles using the while loop. Every time the loop executes the new circle will be larger than the previous one. The n is used as a counter where we specified the value of n increase in the each iteration. Let's understand the iteration of the loop.

- In the first iteration, the initial value of n is 10; it means the turtle draw the circle with the radius of 10 units.
- In the second iteration, the value of n is increased by $10 + 10 = 20$; the turtle draws the circle with the radius of 20 units.
- In the second iteration, the value of n is increased by $20 + 10 = 30$; the turtle draws the circle with the radius of 30 units.

- In the second iteration, the value of n is increased by $30 + 10 = 40$; the turtle draws the circle with the radius of 30 units.

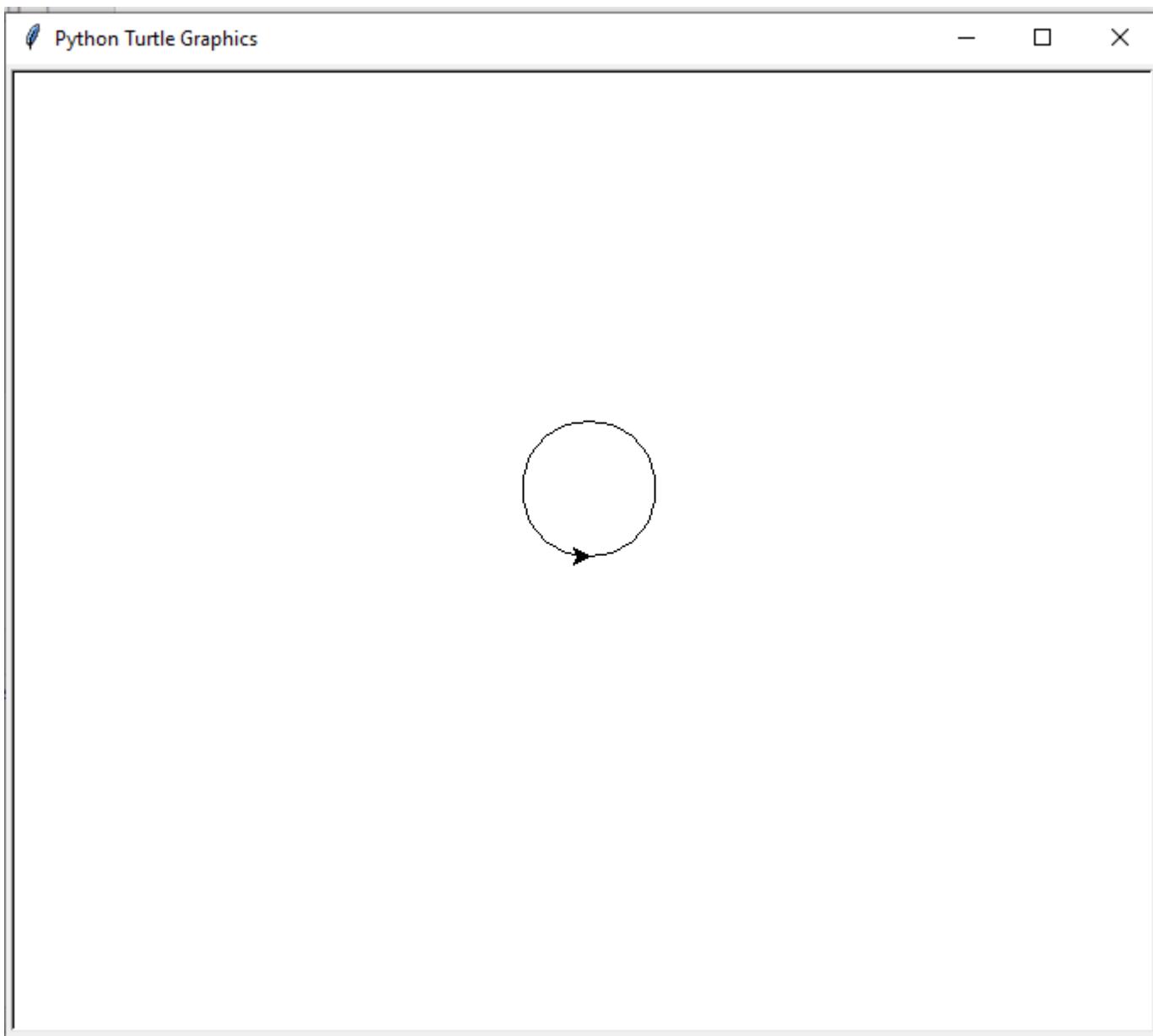
Conditional Statement

The conditional statement is used to check whether a given condition is true. If it is true, execute the corresponding lines of code. Let's understand the following example.

Example

```
1. import turtle  
2. # Creating turtle  
3. t = turtle.Turtle()  
4.  
5. n = 40  
6. if n<=50:  
7.     t.circle(n)  
8. else:  
9.     t.forward(n)  
10.    t.backward(n-10)  
11.  
12. turtle.mainloop()
```

Output:



Explanation

In the above program, we define the two outcomes based on user input. If the entered number is less than or equal to 50 means draw the circle otherwise else part. We gave the 40 as input so that if block got executed and drew the circle.

Now let's move to see a few cool designs using the turtle library.

Attractive Designs using Python Turtle Library

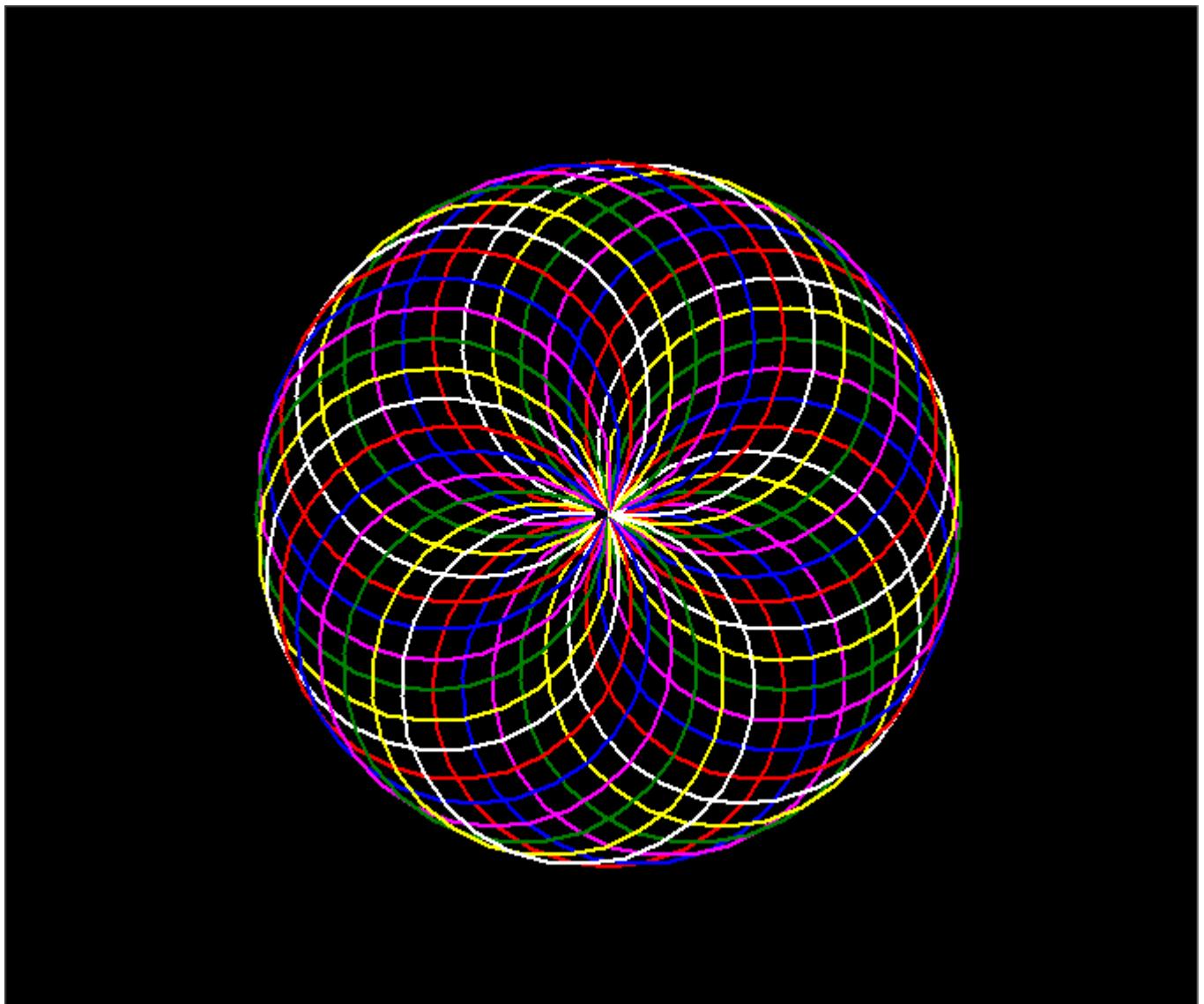
We have learned basic and advance concepts of Python turtle library. We explain every possible feature of this library. By using its function, we can design games, unique shapes and many more things. Here, we mention a few designs using the turtle library.

Design -1 Circle Spiro graph

Code

```
1. import turtle
2. # Creating turtle
3. t = turtle.Turtle()
4.
5. turtle.bgcolor("black")
6. turtle.pensize(2)
7. turtle.speed(0)
8.
9. while (True):
10.     for i in range(6):
11.         for colors in ["red", "blue", "magenta", "green", "yellow", "white"]:
12.             turtle.color(colors)
13.             turtle.circle(100)
14.             turtle.left(10)
15.
16.
17. turtle.hideturtle()
18. turtle.mainloop()
```

Output:



The turtle will move for the infinite time because we have used the infinite while loop. Copy the above code and see the magic.

Design - 2: Python Vibrate Circle

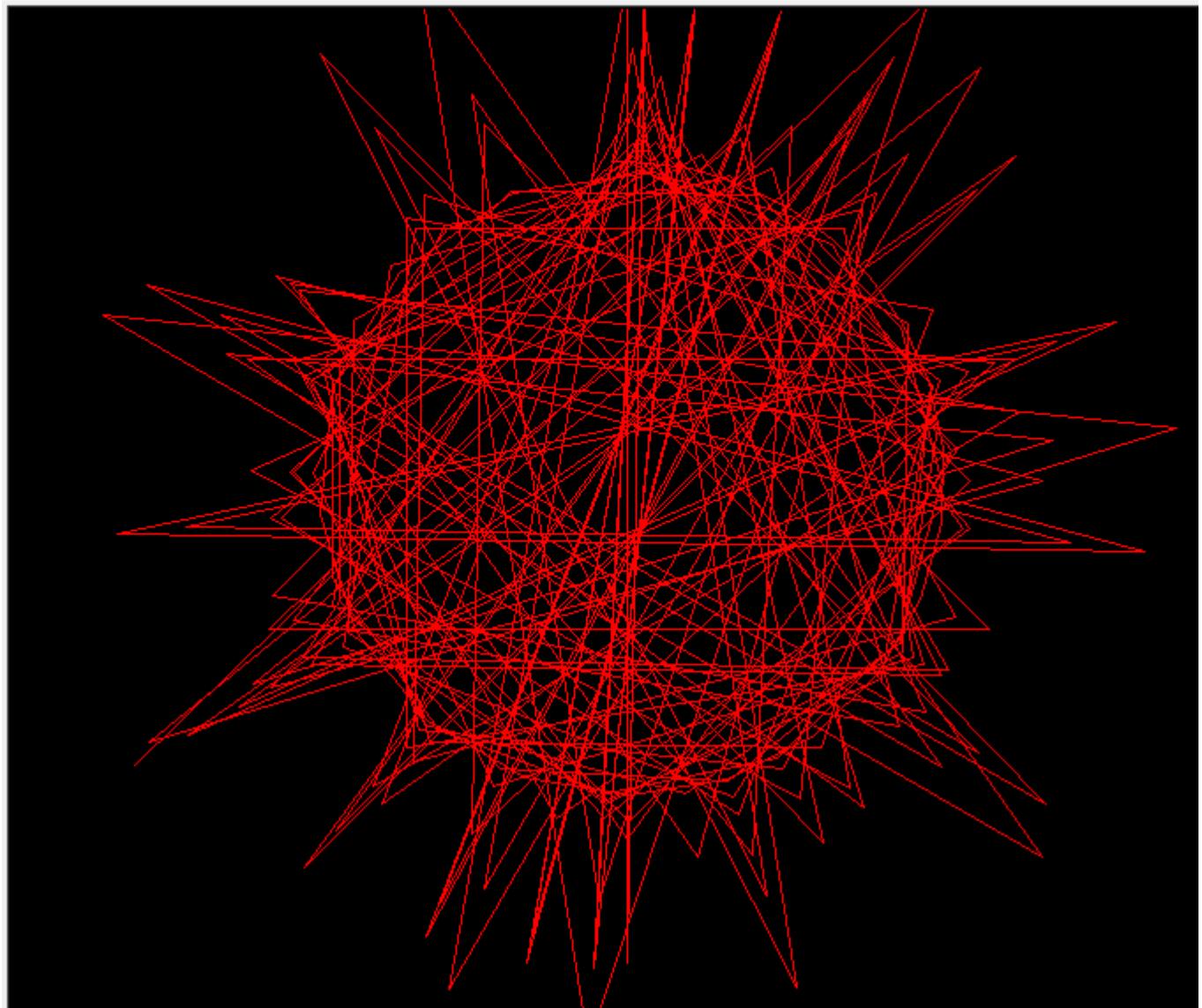
Code

1. **import** turtle
2. # Creating turtle
3. t = turtle.Turtle()
4. s = turtle.Screen()
5. s.bgcolor("black")
6. t.pencolor("red")
- 7.
8. a = 0

```
9. b = 0
10. t.speed(0)
11. t.penup()
12. t.goto(0,200)
13. t.pendown()
14. while(True):
15.     t.forward(a)
16.     t.right(b)
17.     a+=3
18.     b+=1
19.     if b == 210:
20.         break
21.     t.hideturtle()
22.
23. turtle.done()
```

Output:

Python Turtle Graphics

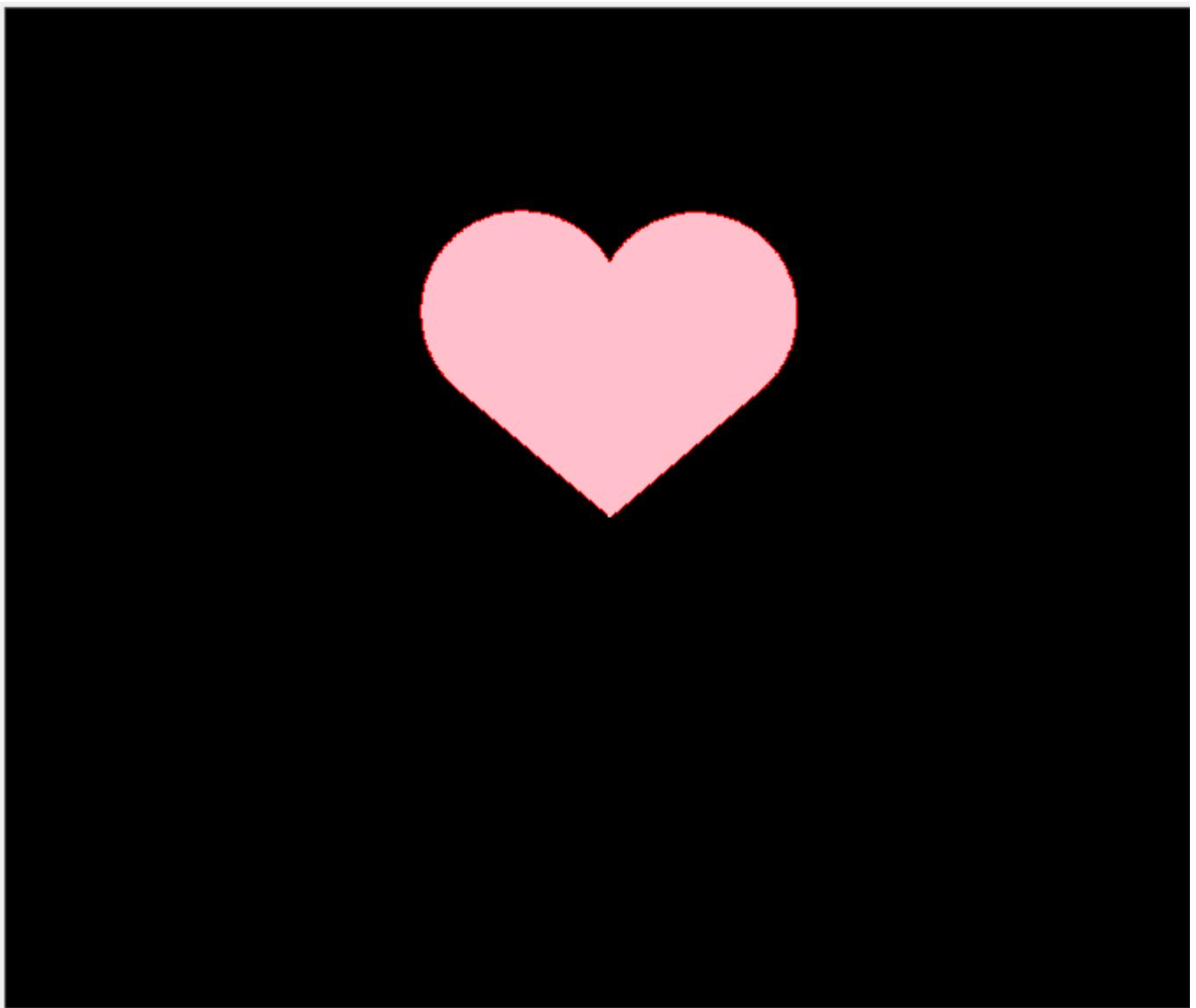


Code

```
1. import turtle  
2. # Creating turtle  
3. t = turtle.Turtle()  
4. s = turtle.Screen()  
5. s.bgcolor("black")  
6.  
7. turtle.pensize(2)  
8.  
9. # To design curve  
10. def curve():
```

```
11. for i in range(200):
12.     t.right(1)
13.     t.forward(1)
14.
15. t.speed(3)
16. t.color("red", "pink")
17.
18. t.begin_fill()
19. t.left(140)
20. t.forward(111.65)
21. curve()
22.
23. t.left(120)
24. curve()
25. t.forward(111.65)
26. t.end_fill()
27. t.hideturtle()
28.
29. turtle.mainloop()
```

Output:



In the above code, we define the curve function to create curve to screen. When it takes the complete heart shape, the color will fill automatically. Copy the above code and run, you can also modify it by adding more designs.

Python Tkinter Tutorial

Tkinter tutorial provides basic and advanced concepts of Python Tkinter. Our Tkinter tutorial is designed for beginners and professionals.

Python provides the standard library Tkinter for creating the graphical user interface for desktop based applications.

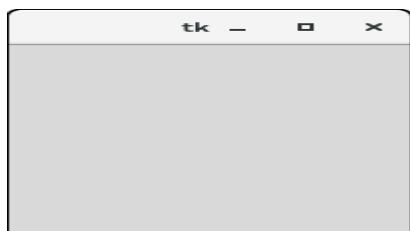
Developing desktop based applications with python Tkinter is not a complex task. An empty Tkinter top-level window can be created by using the following steps.

1. import the Tkinter module.
2. Create the main application window.
3. Add the widgets like labels, buttons, frames, etc. to the window.
4. Call the main event loop so that the actions can take place on the user's computer screen.

Example

```
#!/usr/bin/python3
from tkinter import *
#creating the application main window.
top = Tk()
#Entering the event main loop
top.mainloop()
```

Output: lay Video



Tkinter widgets

There are various widgets like button, canvas, checkbutton, entry, etc. that are used to build the python GUI applications.

SN	Widget	Description
1	<u>Button</u>	The Button is used to add various kinds of buttons to the python application.
2	<u>Canvas</u>	The canvas widget is used to draw the canvas on the window.
3	<u>Checkbutton</u>	The Checkbutton is used to display the CheckButton on the window.
4	<u>Entry</u>	The entry widget is used to display the single-line text field to the user. It is commonly used to accept user values.
5	<u>Frame</u>	It can be defined as a container to which, another widget can be added and organized.
6	<u>Label</u>	A label is a text used to display some message or information about the other widgets.
7	<u>ListBox</u>	The ListBox widget is used to display a list of options to the user.
8	<u>Menubutton</u>	The Menubutton is used to display the menu items to the user.
9	<u>Menu</u>	It is used to add menu items to the user.
10	<u>Message</u>	The Message widget is used to display the message-box to the user.
11	<u>Radiobutton</u>	The Radiobutton is different from a checkbox. Here, the user is provided with various options and the user can select only one option among them.
12	<u>Scale</u>	It is used to provide the slider to the user.
13	<u>Scrollbar</u>	It provides the scrollbar to the user so that the user can scroll the window up and down.
14	<u>Text</u>	It is different from Entry because it provides a multi-line text field to the user so that the user can write the text and edit the text inside it.
14	<u>Toplevel</u>	It is used to create a separate window container.
15	<u>Spinbox</u>	It is an entry widget used to select from options of values.

16	<u>PanedWindow</u>	It is like a container widget that contains horizontal or vertical panes.
17	<u>LabelFrame</u>	A LabelFrame is a container widget that acts as the container
18	<u>MessageBox</u>	This module is used to display the message-box in the desktop based applications.

Python Tkinter Geometry

The Tkinter geometry specifies the method by using which, the widgets are represented on display. The python Tkinter provides the following geometry methods.

1. The pack() method
2. The grid() method
3. The place() method

Let's discuss each one of them in detail.

Python Tkinter pack() method

The pack() widget is used to organize widget in the block. The positions widgets added to the python application using the pack() method can be controlled by using the various options specified in the method call.

However, the controls are less and widgets are generally added in the less organized manner.

The syntax to use the pack() is given below.

syntax

1. `widget.pack(options)`

A list of possible options that can be passed in pack() is given below.

- **expand:** If the expand is set to true, the widget expands to fill any space.
- **Fill:** By default, the fill is set to NONE. However, we can set it to X or Y to determine whether the widget contains any extra space.

- **size:** it represents the side of the parent to which the widget is to be placed on the window.

Example

```
#!/usr/bin/python3
from tkinter import *
parent = Tk()
redbutton = Button(parent, text = "Red", fg = "red")
redbutton.pack( side = LEFT)
greenbutton = Button(parent, text = "Black", fg = "black")
greenbutton.pack( side = RIGHT )
bluebutton = Button(parent, text = "Blue", fg = "blue")
bluebutton.pack( side = TOP )
blackbutton = Button(parent, text = "Green", fg = "red")
blackbutton.pack( side = BOTTOM)
parent.mainloop()
```

Output:



Python Tkinter grid() method

The grid() geometry manager organizes the widgets in the tabular form. We can specify the rows and columns as the options in the method call. We can also specify the column span (width) or rowspan(height) of a widget.

This is a more organized way to place the widgets to the python application. The syntax to use the grid() is given below.

Syntax

1. `widget.grid(options)`

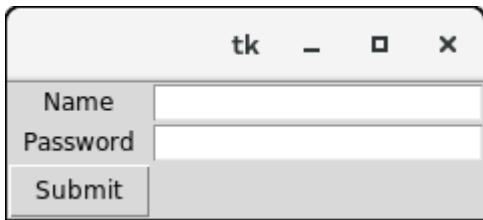
A list of possible options that can be passed inside the grid() method is given below.

- **Column**
The column number in which the widget is to be placed. The leftmost column is represented by 0.
- **Columnspan**
The width of the widget. It represents the number of columns up to which, the column is expanded.
- **ipadx,** **ipady**
It represents the number of pixels to pad the widget inside the widget's border.
- **padx,** **pady**
It represents the number of pixels to pad the widget outside the widget's border.
- **row**
The row number in which the widget is to be placed. The topmost row is represented by 0.
- **The height of the widget, i.e. the number of the row up to which the widget is expanded.**
- **Sticky**
If the cell is larger than a widget, then sticky is used to specify the position of the widget inside the cell. It may be the concatenation of the sticky letters representing the position of the widget. It may be N, E, W, S, NE, NW, NS, EW, ES.

Example

```
#!/usr/bin/python3
from tkinter import *
parent = Tk()
name = Label(parent, text = "Name").grid(row = 0, column = 0)
e1 = Entry(parent).grid(row = 0, column = 1)
password = Label(parent, text = "Password").grid(row = 1, column = 0)
e2 = Entry(parent).grid(row = 1, column = 1)
submit = Button(parent, text = "Submit").grid(row = 4, column = 0)
parent.mainloop()
```

Output:



Python Tkinter place() method

The `place()` geometry manager organizes the widgets to the specific x and y coordinates.

Syntax

1. `widget.place(options)`

A list of possible options is given below.

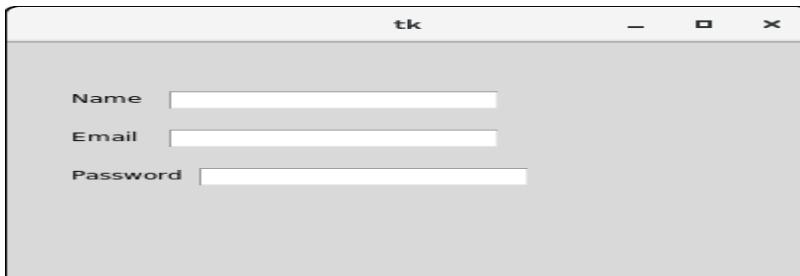
- **Anchor:** It represents the exact position of the widget within the container. The default value (direction) is NW (the upper left corner)
- **bordermode:** The default value of the border type is INSIDE that refers to ignore the parent's inside the border. The other option is OUTSIDE.
- **height, width:** It refers to the height and width in pixels.
- **relheight, relwidth:** It is represented as the float between 0.0 and 1.0 indicating the fraction of the parent's height and width.
- **relx, rely:** It is represented as the float between 0.0 and 1.0 that is the offset in the horizontal and vertical direction.
- **x, y:** It refers to the horizontal and vertical offset in the pixels.

Example

```
#!/usr/bin/python3
from tkinter import *
top = Tk()
top.geometry("400x250")
name = Label(top, text = "Name").place(x = 30,y = 50)
email = Label(top, text = "Email").place(x = 30, y = 90)
password = Label(top, text = "Password").place(x = 30, y = 130)
e1 = Entry(top).place(x = 80, y = 50)
e2 = Entry(top).place(x = 80, y = 90)
```

```
e3 = Entry(top).place(x = 95, y = 130)
top.mainloop()
```

Output:



Python Tkinter Button

The button widget is used to add various types of buttons to the python application. Python allows us to configure the look of the button according to our requirements. Various options can be set or reset depending upon the requirements.

We can also associate a method or function with a button which is called when the button is pressed.

The syntax to use the button widget is given below.

Syntax

1. W = Button(parent, options)

A list of possible options is given below.[Play Video](#)

SN	Option	Description
1	activebackground	It represents the background of the button when the mouse hover the button.
2	activeforeground	It represents the font color of the button when the mouse hover the button.
3	Bd	It represents the border width in pixels.
4	Bg	It represents the background color of the button.
5	Command	It is set to the function call which is scheduled when the function is called.

6	Fg	Foreground color of the button.
7	Font	The font of the button text.
8	Height	The height of the button. The height is represented in the number of text lines for the textual lines or the number of pixels for the images.
10	Highlightcolor	The color of the highlight when the button has the focus.
11	Image	It is set to the image displayed on the button.
12	justify	It illustrates the way by which the multiple text lines are represented. It is set to LEFT for left justification, RIGHT for the right justification, and CENTER for the center.
13	Padx	Additional padding to the button in the horizontal direction.
14	pady	Additional padding to the button in the vertical direction.
15	Relief	It represents the type of the border. It can be SUNKEN, RAISED, GROOVE, and RIDGE.
17	State	This option is set to DISABLED to make the button unresponsive. The ACTIVE represents the active state of the button.
18	Underline	Set this option to make the button text underlined.
19	Width	The width of the button. It exists as a number of letters for textual buttons or pixels for image buttons.
20	Wraplength	If the value is set to a positive number, the text lines will be wrapped to fit within this length.

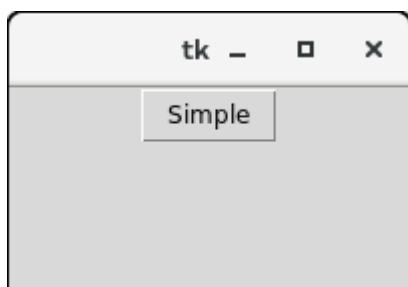
Example

```
#python application to create a simple button
```

```
from tkinter import *
top = Tk()
```

```
top.geometry("200x100")
b = Button(top,text = "Simple")
b.pack()
```

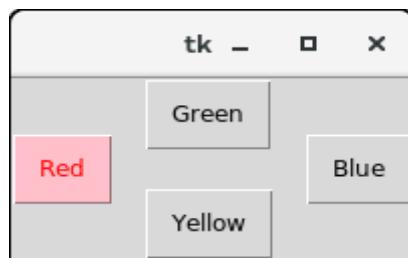
`top.mainloop()` **Output:**



Example

```
from tkinter import *
top = Tk()
top.geometry("200x100")
def fun():
    messagebox.showinfo("Hello", "Red Button clicked")
b1 = Button(top,text = "Red",command = fun,activeforeground = "red",activebackground = "pink",pady=10)
b2 = Button(top, text = "Blue",activeforeground = "blue",activebackground = "pink",pady=10)
b3 = Button(top, text = "Green",activeforeground = "green",activebackground = "pink",pady = 10)
b4 = Button(top, text = "Yellow",activeforeground = "yellow",activebackground = "pink",pady = 10)
b1.pack(side = LEFT)
b2.pack(side = RIGHT)
b3.pack(side = TOP)
b4.pack(side = BOTTOM)
top.mainloop()
```

Output:





Python Tkinter Canvas

The canvas widget is used to add the structured graphics to the python application. It is used to draw the graph and plots to the python application. The syntax to use the canvas is given below.

Syntax

1. `w = canvas(parent, options)`

A list of possible options is given below.

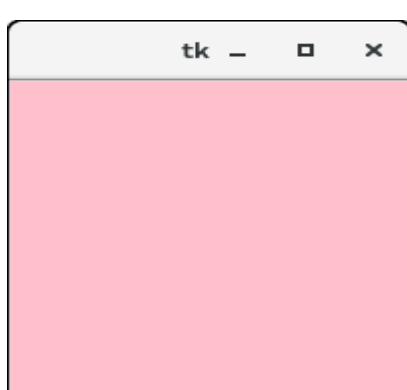
SN	Option	Description
1	<code>bd</code>	The represents the border width. The default width is 2.
2	<code>bg</code>	It represents the background color of the canvas.
3	<code>confine</code>	It is set to make the canvas unscrollable outside the scroll region.
4	<code>cursor</code>	The cursor is used as the arrow, circle, dot, etc. on the canvas.
5	<code>height</code>	It represents the size of the canvas in the vertical direction.
6	<code>highlightcolor</code>	It represents the highlight color when the widget is focused.
7	<code>relief</code>	It represents the type of the border. The possible values are SUNKEN, RAISED, GROOVE, and RIDGE.
8	<code>scrollregion</code>	It represents the coordinates specified as the tuple containing the area of the canvas.
9	<code>width</code>	It represents the width of the canvas.

10	xscrollincrement	If it is set to a positive value. The canvas is placed only to the multiple of this value.
11	xscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the horizontal scrollbar.
12	yscrollincrement	Works like xscrollincrement, but governs vertical movement.
13	yscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the vertical scrollbar.

Example

```
from tkinter import *
top = Tk()
top.geometry("200x200")
#creating a simple canvas
c = Canvas(top,bg = "pink",height = "200")
c.pack()
top.mainloop()
```

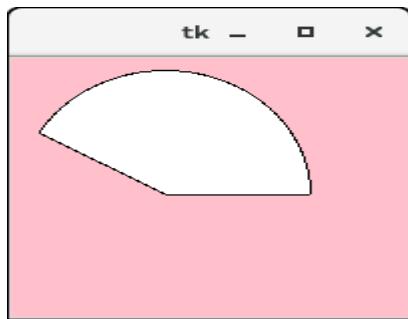
Output:



Example: Creating an arc

```
from tkinter import *
top = Tk()
top.geometry("200x200")
#creating a simple canvas
c = Canvas(top,bg = "pink",height = "200",width = 200)
arc = c.create_arc((5,10,150,200),start = 0,extent = 150, fill= "white")
c.pack()
top.mainloop()
```

Output:



Python Tkinter Checkbutton

The Checkbutton is used to track the user's choices provided to the application. In other words, we can say that Checkbutton is used to implement the on/off selections.

The Checkbutton can contain the text or images. The Checkbutton is mostly used to provide many choices to the user among which, the user needs to choose the one. It generally implements many of many selections.

The syntax to use the checkbutton is given below.

Syntax

1. w = checkbutton(master, options)

SN	Option	Description
----	--------	-------------

1	activebackground	It represents the background color when the checkbutton is under the cursor.
2	activeforeground	It represents the foreground color of the checkbutton when the checkbutton is under the cursor.
3	bg	The background color of the button.
4	bitmap	It displays an image (monochrome) on the button.
5	bd	The size of the border around the corner.
6	command	It is associated with a function to be called when the state of the checkbutton is changed.
7	cursor	The mouse pointer will be changed to the cursor name when it is over the checkbutton.
8	disableforeground	It is the color which is used to represent the text of a disabled checkbutton.
9	font	It represents the font of the checkbutton.
10	fg	The foreground color (text color) of the checkbutton.
11	height	It represents the height of the checkbutton (number of lines). The default height is 1.
12	highlightcolor	The color of the focus highlight when the checkbutton is under focus.
13	image	The image used to represent the checkbutton.
14	justify	This specifies the justification of the text if the text contains multiple lines.
15	offvalue	The associated control variable is set to 0 by default if the button is unchecked. We can change the state of an unchecked variable to some other one.
16	onvalue	The associated control variable is set to 1 by default if the button is checked. We can change the state of the checked variable to some other one.
17	padx	The horizontal padding of the checkbutton

18	pady	The vertical padding of the checkbutton.
19	relief	The type of the border of the checkbutton. By default, it is set to FLAT.
20	selectcolor	The color of the checkbutton when it is set. By default, it is red.
21	selectimage	The image is shown on the checkbutton when it is set.
22	state	It represents the state of the checkbutton. By default, it is set to normal. We can change it to DISABLED to make the checkbutton unresponsive. The state of the checkbutton is ACTIVE when it is under focus.
24	underline	It represents the index of the character in the text which is to be underlined. The indexing starts with zero in the text.
25	variable	It represents the associated variable that tracks the state of the checkbutton.
26	width	It represents the width of the checkbutton. It is represented in the number of characters that are represented in the form of texts.
27	wraplength	If this option is set to an integer number, the text will be broken into the number of pieces.

Methods

The methods that can be called with the Checkbuttons are described in the following table.

SN	Method	Description
1	deselect()	It is called to turn off the checkbutton.
2	flash()	The checkbutton is flashed between the active and normal colors.
3	invoke()	This will invoke the method associated with the checkbutton.
4	select()	It is called to turn on the checkbutton.
5	toggle()	It is used to toggle between the different Checkbuttons.

Example

```
from tkinter import *
top = Tk()
top.geometry("200x200")
checkvar1 = IntVar()
checkvar2 = IntVar()
checkvar3 = IntVar()
chkbtn1 = Checkbutton(top, text = "C", variable = checkvar1, onvalue = 1, offvalue = 0, height = 2, width = 10)
chkbtn2 = Checkbutton(top, text = "C++", variable = checkvar2, onvalue = 1, offvalue = 0, height = 2, width = 10)
chkbtn3 = Checkbutton(top, text = "Java", variable = checkvar3, onvalue = 1, offvalue = 0, height = 2, width = 10)

chkbtn1.pack()
chkbtn2.pack()
chkbtn3.pack()

top.mainloop()
```

Output:



Python Tkinter Entry

The Entry widget is used to provide the single line text-box to the user to accept a value from the user. We can use the Entry widget to accept the text strings from the user. It can only be used for one line of text from the user. For multiple lines of text, we must use the text widget.

The syntax to use the Entry widget is given below.

Syntax

1. `w = Entry (parent, options)`

A list of possible options is given below.

SN	Option	Description
1	<code>bg</code>	The background color of the widget.
2	<code>bd</code>	The border width of the widget in pixels.
3	<code>cursor</code>	The mouse pointer will be changed to the cursor type set to the arrow, dot, etc.
4	<code>exportselection</code>	The text written inside the entry box will be automatically copied to the clipboard by default. We can set the exportselection to 0 to not copy this.
5	<code>fg</code>	It represents the color of the text.
6	<code>font</code>	It represents the font type of the text.
7	<code>highlightbackground</code>	It represents the color to display in the traversal highlight region when the widget does not have the input focus.
8	<code>highlightcolor</code>	It represents the color to use for the traversal highlight rectangle that is drawn around the widget when it has the input focus.
9	<code>highlightthickness</code>	It represents a non-negative value indicating the width of the highlight rectangle to draw around the outside of the widget when it has the input focus.

10	insertbackground	It represents the color to use as background in the area covered by the insertion cursor. This color will normally override either the normal background for the widget.
11	insertborderwidth	It represents a non-negative value indicating the width of the 3-D border to draw around the insertion cursor. The value may have any of the forms acceptable to Tk_GetPixels.
12	insertofftime	It represents a non-negative integer value indicating the number of milliseconds the insertion cursor should remain "off" in each blink cycle. If this option is zero, then the cursor doesn't blink: it is on all the time.
13	insertontime	Specifies a non-negative integer value indicating the number of milliseconds the insertion cursor should remain "on" in each blink cycle.
14	insertwidth	It represents the value indicating the total width of the insertion cursor. The value may have any of the forms acceptable to Tk_GetPixels.
15	justify	It specifies how the text is organized if the text contains multiple lines.
16	relief	It specifies the type of the border. Its default value is FLAT.
17	selectbackground	The background color of the selected text.
18	selectborderwidth	The width of the border to display around the selected task.
19	selectforeground	The font color of the selected task.
20	show	It is used to show the entry text of some other type instead of the string. For example, the password is typed using stars (*).
21	textvariable	It is set to the instance of the StringVar to retrieve the text from the entry.
22	width	The width of the displayed text or image.

23	xscrollcommand	The entry widget can be linked to the horizontal scrollbar if we want the user to enter more text than the actual width of the widget.
----	----------------	--

Example

```
#!/usr/bin/python3

from tkinter import *

top = Tk()

top.geometry("400x250")

name = Label(top, text = "Name").place(x = 30,y = 50)

email = Label(top, text = "Email").place(x = 30, y = 90)

password = Label(top, text = "Password").place(x = 30, y = 130)

submitbtn = Button(top, text = "Submit",activebackground = "pink", activeforeground = "blue").place(x = 30, y = 170)

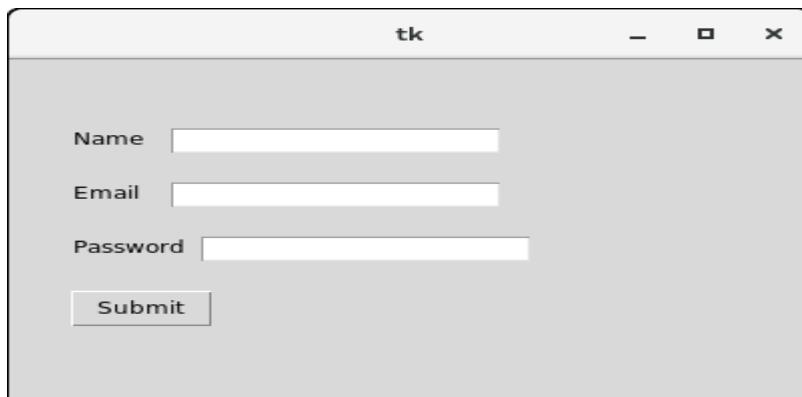
e1 = Entry(top).place(x = 80, y = 50)

e2 = Entry(top).place(x = 80, y = 90)

e3 = Entry(top).place(x = 95, y = 130)

top.mainloop()
```

Output:



Entry widget methods

Python provides various methods to configure the data written inside the widget. There are the following methods provided by the Entry widget.

SN	Method	Description
1	delete(first, last = none)	It is used to delete the specified characters inside the widget.
2	get()	It is used to get the text written inside the widget.
3	icursor(index)	It is used to change the insertion cursor position. We can specify the index of the character before which, the cursor to be placed.
4	index(index)	It is used to place the cursor to the left of the character written at the specified index.
5	insert(index,s)	It is used to insert the specified string before the character placed at the specified index.
6	select_adjust(index)	It includes the selection of the character present at the specified index.
7	select_clear()	It clears the selection if some selection has been done.
8	select_form(index)	It sets the anchor index position to the character specified by the index.
9	select_present()	It returns true if some text in the Entry is selected otherwise returns false.

10	<code>select_range(start,end)</code>	It selects the characters to exist between the specified range.
11	<code>select_to(index)</code>	It selects all the characters from the beginning to the specified index.
12	<code>xview(index)</code>	It is used to link the entry widget to a horizontal scrollbar.
13	<code>xview_scroll(number,what)</code>	It is used to make the entry scrollable horizontally.

Example: A simple calculator

```

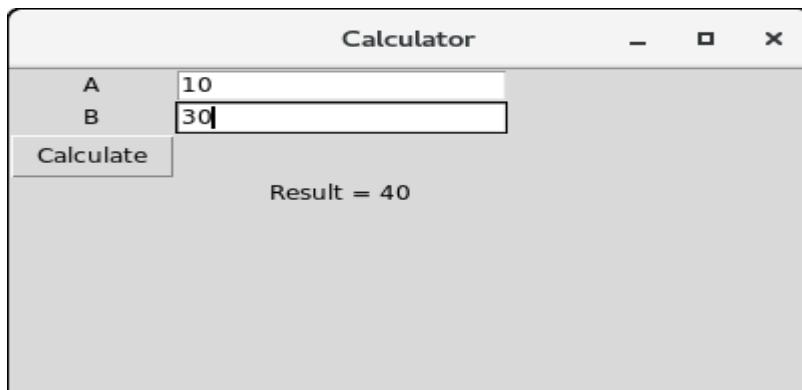
import tkinter as tk
from functools import partial

def call_result(label_result, n1, n2):
    num1 = (n1.get())
    num2 = (n2.get())
    result = int(num1)+int(num2)
    label_result.config(text="Result = %d" % result)
    return

root = tk.Tk()
root.geometry('400x200+100+200')
root.title('Calculator')
number1 = tk.StringVar()
number2 = tk.StringVar()
labelNum1 = tk.Label(root, text="A").grid(row=1, column=0)
labelNum2 = tk.Label(root, text="B").grid(row=2, column=0)
labelResult = tk.Label(root)
labelResult.grid(row=7, column=2)
entryNum1 = tk.Entry(root, textvariable=number1).grid(row=1, column=2)
entryNum2 = tk.Entry(root, textvariable=number2).grid(row=2, column=2)
call_result = partial(call_result, labelResult, number1, number2)
buttonCal = tk.Button(root, text="Calculate", command=call_result).grid(row=3, col
umn=0)
root.mainloop()

```

Output:



Python Tkinter Frame

Python Tkinter Frame widget is used to organize the group of widgets. It acts like a container which can be used to hold the other widgets. The rectangular areas of the screen are used to organize the widgets to the python application.

The syntax to use the Frame widget is given below.

Syntax

1. `w = Frame(parent, options)`

A list of possible options is given below.

SN	Option	Description
1	<code>bd</code>	It represents the border width.
2	<code>bg</code>	The background color of the widget.
3	<code>cursor</code>	The mouse pointer is changed to the cursor type set to different values like an arrow, dot, etc.
4	<code>height</code>	The height of the frame.
5	<code>highlightbackground</code>	The color of the background color when it is under focus.
6	<code>highlightcolor</code>	The text color when the widget is under focus.
7	<code>highlightthickness</code>	It specifies the thickness around the border when the widget is under the focus.
8	<code>relief</code>	It specifies the type of the border. The default value is FLAT.

9	width	It represents the width of the widget.
---	-------	--

Example

```
from tkinter import *

top = Tk()
top.geometry("140x100")
frame = Frame(top)
frame.pack()

leftframe = Frame(top)
leftframe.pack(side = LEFT)

rightframe = Frame(top)
rightframe.pack(side = RIGHT)

btn1 = Button(frame, text="Submit", fg="red",activebackground = "red")
btn1.pack(side = LEFT)

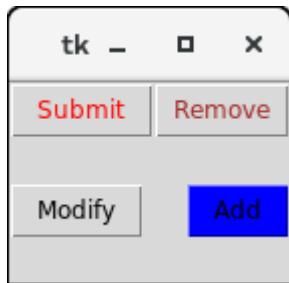
btn2 = Button(frame, text="Remove", fg="brown", activebackground = "brown")
btn2.pack(side = RIGHT)

btn3 = Button(rightframe, text="Add", fg="blue", activebackground = "blue")
btn3.pack(side = LEFT)

btn4 = Button(leftframe, text="Modify", fg="black", activebackground = "white")
btn4.pack(side = RIGHT)

top.mainloop()
```

Output:



Python Tkinter Label

The Label is used to specify the container box where we can place the text or images. This widget is used to provide the message to the user about other widgets used in the python application.

There are the various options which can be specified to configure the text or the part of the text shown in the Label.

The syntax to use the Label is given below.

Syntax

1. `w = Label (master, options)`

A list of possible options is given below.

SN	Option	Description
1	anchor	It specifies the exact position of the text within the size provided to the widget. The default value is CENTER, which is used to center the text within the specified space.
2	bg	The background color displayed behind the widget.
3	bitmap	It is used to set the bitmap to the graphical object specified so that, the label can represent the graphics instead of text.
4	bd	It represents the width of the border. The default is 2 pixels.
5	cursor	The mouse pointer will be changed to the type of the cursor specified, i.e., arrow, dot, etc.
6	font	The font type of the text written inside the widget.

7	fg	The foreground color of the text written inside the widget.
8	height	The height of the widget.
9	image	The image that is to be shown as the label.
10	justify	It is used to represent the orientation of the text if the text contains multiple lines. It can be set to LEFT for left justification, RIGHT for right justification, and CENTER for center justification.
11	padx	The horizontal padding of the text. The default value is 1.
12	pady	The vertical padding of the text. The default value is 1.
13	relief	The type of the border. The default value is FLAT.
14	text	This is set to the string variable which may contain one or more line of text.
15	textvariable	The text written inside the widget is set to the control variable StringVar so that it can be accessed and changed accordingly.
16	underline	We can display a line under the specified letter of the text. Set this option to the number of the letter under which the line will be displayed.
17	width	The width of the widget. It is specified as the number of characters.
18	wraplength	Instead of having only one line as the label text, we can break it to the number of lines where each line has the number of characters specified to this option.

Example 1

```
#!/usr/bin/python3
```

```
from tkinter import *
```

```
top = Tk()
```

```
top.geometry("400x250")
```

```

#creating label
uname = Label(top, text = "Username").place(x = 30,y = 50)

#creating label
password = Label(top, text = "Password").place(x = 30, y = 90)

sbmitbtn = Button(top, text = "Submit",activebackground = "pink", activeforeground = "blue").place(x = 30, y = 120)

e1 = Entry(top,width = 20).place(x = 100, y = 50)

e2 = Entry(top, width = 20).place(x = 100, y = 90)

top.mainloop()

```

Output:



Python Tkinter Listbox

The Listbox widget is used to display the list items to the user. We can place only text items in the Listbox and all text items contain the same font and color.

The user can choose one or more items from the list depending upon the configuration.

The syntax to use the Listbox is given below.

1. w = Listbox(parent, options)

A list of possible options is given below.

SN	Option	Description
1	bg	The background color of the widget.
2	bd	It represents the size of the border. Default value is 2 pixel.
3	cursor	The mouse pointer will look like the cursor type like dot, arrow, etc.
4	font	The font type of the Listbox items.
5	fg	The color of the text.
6	height	It represents the count of the lines shown in the Listbox. The default value is 10.
7	highlightcolor	The color of the Listbox items when the widget is under focus.
8	highlightthickness	The thickness of the highlight.
9	relief	The type of the border. The default is SUNKEN.
10	selectbackground	The background color that is used to display the selected text.
11	selectmode	It is used to determine the number of items that can be selected from the list. It can set to BROWSE, SINGLE, MULTIPLE, EXTENDED.
12	width	It represents the width of the widget in characters.
13	xscrollcommand	It is used to let the user scroll the Listbox horizontally.
14	yscrollcommand	It is used to let the user scroll the Listbox vertically.

Methods

There are the following methods associated with the Listbox.

SN	Method	Description
1	activate(index)	It is used to select the lines at the specified index.

2	<code>curselection()</code>	It returns a tuple containing the line numbers of the selected element or elements, counting from 0. If nothing is selected, returns an empty tuple.
3	<code>delete(first, last = None)</code>	It is used to delete the lines which exist in the given range.
4	<code>get(first, last = None)</code>	It is used to get the list items that exist in the given range.
5	<code>index(i)</code>	It is used to place the line with the specified index at the top of the widget.
6	<code>insert(index, *elements)</code>	It is used to insert the new lines with the specified number of elements before the specified index.
7	<code>nearest(y)</code>	It returns the index of the nearest line to the y coordinate of the Listbox widget.
8	<code>see(index)</code>	It is used to adjust the position of the listbox to make the lines specified by the index visible.
9	<code>size()</code>	It returns the number of lines that are present in the Listbox widget.
10	<code>xview()</code>	This is used to make the widget horizontally scrollable.
11	<code>xview_moveto(fraction)</code>	It is used to make the listbox horizontally scrollable by the fraction of width of the longest line present in the listbox.
12	<code>xview_scroll(number, what)</code>	It is used to make the listbox horizontally scrollable by the number of characters specified.
13	<code>yview()</code>	It allows the Listbox to be vertically scrollable.
14	<code>yview_moveto(fraction)</code>	It is used to make the listbox vertically scrollable by the fraction of width of the longest line present in the listbox.
15	<code>yview_scroll (number, what)</code>	It is used to make the listbox vertically scrollable by the number of characters specified.

Example 1

```
#!/usr/bin/python3

from tkinter import *

top = Tk()

top.geometry("200x250")

lbl = Label(top, text = "A list of favourite countries...")

listbox = Listbox(top)

listbox.insert(1,"India")

listbox.insert(2, "USA")

listbox.insert(3, "Japan")

listbox.insert(4, "Australia")

lbl.pack()

listbox.pack()

top.mainloop()
```

Output:



Example 2: Deleting the active items from the list

```
#!/usr/bin/python3

from tkinter import *

top = Tk()

top.geometry("200x250")

lbl = Label(top, text = "A list of favourite countries...")

listbox = Listbox(top)

listbox.insert(1,"India")

listbox.insert(2, "USA")

listbox.insert(3, "Japan")

listbox.insert(4, "Australia")

#this button will delete the selected item from the list

btn = Button(top, text = "delete", command = lambda listbox=listbox: listbox.delete
ANCHOR)

lbl.pack()

listbox.pack()

btn.pack()

top.mainloop()
```

Output:



After pressing the delete button.



Python Tkinter Menubutton

The Menubutton widget can be defined as the drop-down menu that is shown to the user all the time. It is used to provide the user a option to select the appropriate choice exist within the application.

The Menubutton is used to implement various types of menus in the python application. A Menu is associated with the Menubutton that can display the choices of the Menubutton when clicked by the user.

The syntax to use the python tkinter Menubutton is given below.

Syntax

1. w = Menubutton(Top, options)

A list of various options is given below.

SN	Option	Description
1	activebackground	The background color of the widget when the widget is under focus.
2	activeforeground	The font color of the widget text when the widget is under focus.
3	anchor	It specifies the exact position of the widget content when the widget is assigned more space than needed.
4	bg	It specifies the background color of the widget.
5	bitmap	It is set to the graphical content which is to be displayed to the widget.
6	bd	It represents the size of the border. The default value is 2 pixels.
7	cursor	The mouse pointer will be changed to the cursor type specified when the widget is under the focus. The possible value of the cursor type is arrow, or dot etc.
8	direction	Its direction can be specified so that menu can be displayed to the specified direction of the button. Use LEFT, RIGHT, or ABOVE to place the widget accordingly.
9	disabledforeground	The text color of the widget when the widget is disabled.
10	fg	The normal foreground color of the widget.
11	height	The vertical dimension of the Menubutton. It is specified as the number of lines.
12	highlightcolor	The highlight color shown to the widget under focus.
13	image	The image displayed on the widget.
14	justify	This specifies the exact position of the text under the widget when the text is unable to fill the width of the widget. We can use the LEFT for the left justification, RIGHT for the right justification, CENTER for the centre justification.

15	menu	It represents the menu specified with the Menubutton.
16	padx	The horizontal padding of the widget.
17	pady	The vertical padding of the widget.
18	relief	This option specifies the type of the border. The default value is RAISED.
19	state	The normal state of the Mousebutton is enabled. We can set it to DISABLED to make it unresponsive.
20	text	The text shown with the widget.
21	textvariable	We can set the control variable of string type to the text variable so that we can control the text of the widget at runtime.
22	underline	The text of the widget is not underlined by default but we can set this option to make the text of the widget underlined.
23	width	It represents the width of the widget in characters. The default value is 20.
24	wraplength	We can break the text of the widget in the number of lines so that the text contains the number of lines not greater than the specified value.

Example

```
#!/usr/bin/python3

from tkinter import *

top = Tk()

top.geometry("200x250")

menobutton = Menobutton(top, text = "Language", relief = FLAT)

menobutton.grid()
```

```
menubutton.menu = Menu(menubutton)

menubutton["menu"] = menubutton.menu

menubutton.menu.add_checkbutton(label = "Hindi", variable=IntVar())

menubutton.menu.add_checkbutton(label = "English", variable = IntVar())

menubutton.pack()

top.mainloop()
```

Output:



Python Tkinter Menu

The Menu widget is used to create various types of menus (top level, pull down, and pop up) in the python application.

The top-level menus are the one which is displayed just under the title bar of the parent window. We need to create a new instance of the Menu widget and add various commands to it by using the add() method.

The syntax to use the Menu widget is given below.

Syntax

1. w = Menu(top, options)

A list of possible options is given below.

SN	Option	Description
1	activebackground	The background color of the widget when the widget is under the focus.
2	activeborderwidth	The width of the border of the widget when it is under the mouse. The default is 1 pixel.
3	activeforeground	The font color of the widget when the widget has the focus.
4	bg	The background color of the widget.
5	bd	The border width of the widget.
6	cursor	The mouse pointer is changed to the cursor type when it hovers the widget. The cursor type can be set to arrow or dot.
7	disabledforeground	The font color of the widget when it is disabled.
8	font	The font type of the text of the widget.
9	fg	The foreground color of the widget.
10	postcommand	The postcommand can be set to any of the function which is called when the mouse hovers the menu.
11	relief	The type of the border of the widget. The default type is RAISED.
12	image	It is used to display an image on the menu.
13	selectcolor	The color used to display the checkbox or radiobutton when they are selected.
14	tearoff	By default, the choices in the menu start taking place from position 1. If we set the tearoff = 1, then it will start taking place from 0th position.
15	title	Set this option to the title of the window if you want to change the title of the window.

Methods

The Menu widget contains the following methods.

SN	Option	Description
1	add_command(options)	It is used to add the Menu items to the menu.
2	add_radiobutton(options)	This method adds the radiobutton to the menu.
3	add_checkbutton(options)	This method is used to add the checkbuttons to the menu.
4	add_cascade(options)	It is used to create a hierarchical menu to the parent menu by associating the given menu to the parent menu.
5	add_seperator()	It is used to add the seperator line to the menu.
6	add(type, options)	It is used to add the specific menu item to the menu.
7	delete(startindex, endindex)	It is used to delete the menu items exist in the specified range.
8	entryconfig(index, options)	It is used to configure a menu item identified by the given index.
9	index(item)	It is used to get the index of the specified menu item.
10	insert_seperator(index)	It is used to insert a seperator at the specified index.
11	invoke(index)	It is used to invoke the associated with the choice given at the specified index.
12	type(index)	It is used to get the type of the choice specified by the index.

Creating a top level menu

A top-level menu can be created by instantiating the Menu widget and adding the menu items to the menu.

Example 1

```
#!/usr/bin/python3

from tkinter import *

top = Tk()

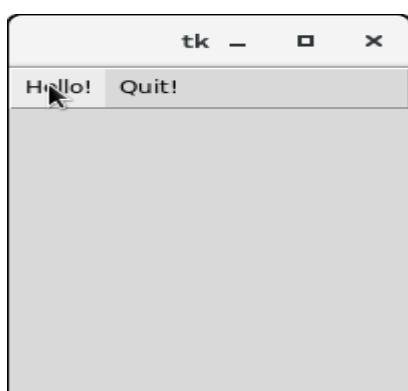
def hello():
    print("hello!")

# create a toplevel menu
menubar = Menu(root)
menubar.add_command(label="Hello!", command=hello)
menubar.add_command(label="Quit!", command=top.quit)

# display the menu
top.config(menu=menubar)

top.mainloop()
```

Output:



Clicking the hello Menubutton will print the hello on the console while clicking the Quit Menubutton will make an exit from the python application.

Example 2

```
from tkinter import Toplevel, Button, Tk, Menu
```

```
top = Tk()
menubar = Menu(top)
file = Menu(menubar, tearoff=0)
file.add_command(label="New")
file.add_command(label="Open")
file.add_command(label="Save")
file.add_command(label="Save as...")
file.add_command(label="Close")

file.add_separator()

file.add_command(label="Exit", command=top.quit)

menubar.add_cascade(label="File", menu=file)
edit = Menu(menubar, tearoff=0)
edit.add_command(label="Undo")

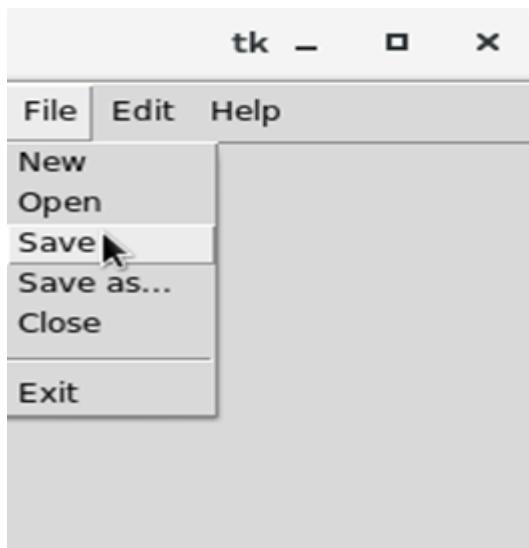
edit.add_separator()

edit.add_command(label="Cut")
edit.add_command(label="Copy")
edit.add_command(label="Paste")
edit.add_command(label="Delete")
edit.add_command(label="Select All")

menubar.add_cascade(label="Edit", menu=edit)
help = Menu(menubar, tearoff=0)
help.add_command(label="About")
menubar.add_cascade(label="Help", menu=help)

top.config(menu=menubar) top.mainloop()
```

Output



Python Tkinter Message

The Message widget is used to show the message to the user regarding the behaviour of the python application. The message widget shows the text messages to the user which can not be edited.

The message text contains more than one line. However, the message can only be shown in the single font.

The syntax to use the Message widget is given below.

Syntax

1. `w = Message(parent, options)`

A list of possible options is given below.

SN	Option	Description
1	anchor	It is used to decide the exact position of the text within the space provided to the widget if the widget contains more space than the need of the text. The default is CENTER.
2	bg	The background color of the widget.
3	bitmap	It is used to display the graphics on the widget. It can be set to any graphical or image object.

4	bd	It represents the size of the border in the pixel. The default size is 2 pixel.
5	cursor	The mouse pointer is changed to the specified cursor type. The cursor type can be an arrow, dot, etc.
6	font	The font type of the widget text.
7	fg	The font color of the widget text.
8	height	The vertical dimension of the message.
9	image	We can set this option to a static image to show that onto the widget.
10	justify	This option is used to specify the alignment of multiple line of code with respect to each other. The possible values can be LEFT (left alignment), CENTER (default), and RIGHT (right alignment).
11	padx	The horizontal padding of the widget.
12	pady	The vertical padding of the widget.
13	relief	It represents the type of the border. The default type is FLAT.
14	text	We can set this option to the string so that the widget can represent the specified text.
15	textvariable	This is used to control the text represented by the widget. The textvariable can be set to the text that is shown in the widget.
16	underline	The default value of this option is -1 that represents no underline. We can set this option to an existing number to specify that nth letter of the string will be underlined.
17	width	It specifies the horizontal dimension of the widget in the number of characters (not pixel).
18	wraplength	We can wrap the text to the number of lines by setting this option to the desired number so that each line contains only that number of characters.

Example

```
from tkinter import *
```

```

top = Tk()
top.geometry("100x100")
var = StringVar()
msg = Message( top, text = "Welcome to acytech")

msg.pack()
top.mainloop()

```

Output:



Python Tkinter Radiobutton

The Radiobutton widget is used to implement one-of-many selection in the python application. It shows multiple choices to the user out of which, the user can select only one out of them. We can associate different methods with each of the radiobutton.

We can display the multiple line text or images on the radiobuttons. To keep track the user's selection the radiobutton, it is associated with a single variable. Each button displays a single value for that particular variable.

The syntax to use the Radiobutton is given below.

Syntax

1. w = Radiobutton(top, options)

SN	Option	Description
1	activebackground	The background color of the widget when it has the focus.
2	activeforeground	The font color of the widget text when it has the focus.

3	anchor	It represents the exact position of the text within the widget if the widget contains more space than the requirement of the text. The default value is CENTER.
4	bg	The background color of the widget.
5	bitmap	It is used to display the graphics on the widget. It can be set to any graphical or image object.
6	borderwidth	It represents the size of the border.
7	command	This option is set to the procedure which must be called every-time when the state of the radiobutton is changed.
8	cursor	The mouse pointer is changed to the specified cursor type. It can be set to the arrow, dot, etc.
9	font	It represents the font type of the widget text.
10	fg	The normal foreground color of the widget text.
11	height	The vertical dimension of the widget. It is specified as the number of lines (not pixel).
12	highlightcolor	It represents the color of the focus highlight when the widget has the focus.
13	highlightbackground	The color of the focus highlight when the widget is not having the focus.
14	image	It can be set to an image object if we want to display an image on the radiobutton instead the text.
15	justify	It represents the justification of the multi-line text. It can be set to CENTER(default), LEFT, or RIGHT.
16	padx	The horizontal padding of the widget.
17	pady	The vertical padding of the widget.
18	relief	The type of the border. The default value is FLAT.
19	selectcolor	The color of the radio button when it is selected.
20	selectimage	The image to be displayed on the radiobutton when it is selected.

21	state	It represents the state of the radio button. The default state of the Radiobutton is NORMAL. However, we can set this to DISABLED to make the radiobutton unresponsive.
22	text	The text to be displayed on the radiobutton.
23	textvariable	It is of String type that represents the text displayed by the widget.
24	underline	The default value of this option is -1, however, we can set this option to the number of character which is to be underlined.
25	value	The value of each radiobutton is assigned to the control variable when it is turned on by the user.
26	variable	It is the control variable which is used to keep track of the user's choices. It is shared among all the radiobuttons.
27	width	The horizontal dimension of the widget. It is represented as the number of characters.
28	wraplength	We can wrap the text to the number of lines by setting this option to the desired number so that each line contains only that number of characters.

Methods

The radiobutton widget provides the following methods.

Play Video SN	Method	Description
1	deselect()	It is used to turn off the radiobutton.
2	flash()	It is used to flash the radiobutton between its active and normal colors few times.
3	invoke()	It is used to call any procedure associated when the state of a Radiobutton is changed.
4	select()	It is used to select the radiobutton.

Example

```

from tkinter import *

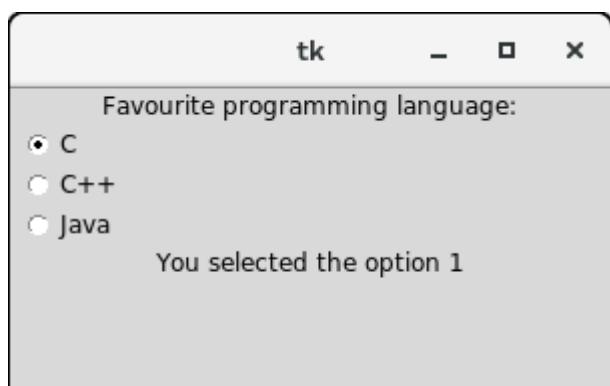
def selection():
    selection = "You selected the option " + str(radio.get())
    label.config(text = selection)

top = Tk()
top.geometry("300x150")
radio = IntVar()
lbl = Label(text = "Favourite programming language:")
lbl.pack()
R1 = Radiobutton(top, text="C", variable=radio, value=1,
                  command=selection)
R1.pack( anchor = W )
R2 = Radiobutton(top, text="C++", variable=radio, value=2,
                  command=selection)
R2.pack( anchor = W )
R3 = Radiobutton(top, text="Java", variable=radio, value=3,
                  command=selection)
R3.pack( anchor = W )

label = Label(top)
label.pack()
top.mainloop()

```

Output:



Python Tkinter Scale

The Scale widget is used to implement the graphical slider to the python application so that the user can slide through the range of values shown on the slider and select the one among them.

We can control the minimum and maximum values along with the resolution of the scale. It provides an alternative to the Entry widget when the user is forced to select only one value from the given range of values.

The syntax to use the Scale widget is given below.

Syntax

1. w = Scale(top, options)

A list of possible options is given below.

SN	Option	Description
1	activebackground	The background color of the widget when it has the focus.
2	bg	The background color of the widget.
3	bd	The border size of the widget. The default is 2 pixel.
4	command	It is set to the procedure which is called each time when we move the slider. If the slider is moved rapidly, the callback is done when it settles.
5	cursor	The mouse pointer is changed to the cursor type assigned to this option. It can be an arrow, dot, etc.
6	digits	If the control variable used to control the scale data is of string type, this option is used to specify the number of digits when the numeric scale is converted to a string.
7	font	The font type of the widget text.
8	fg	The foreground color of the text.

9	from_	It is used to represent one end of the widget range.
10	highlightbackground	The highlight color when the widget doesn't have the focus.
11	highlighcolor	The highlight color when the widget has the focus.
12	label	This can be set to some text which can be shown as a label with the scale. It is shown in the top left corner if the scale is horizontal or the top right corner if the scale is vertical.
13	length	It represents the length of the widget. It represents the X dimension if the scale is horizontal or y dimension if the scale is vertical.
14	orient	It can be set to horizontal or vertical depending upon the type of the scale.
15	relief	It represents the type of the border. The default is FLAT.
16	repeatdelay	This option tells the duration up to which the button is to be pressed before the slider starts moving in that direction repeatedly. The default is 300 ms.
17	resolution	It is set to the smallest change which is to be made to the scale value.
18	showvalue	The value of the scale is shown in the text form by default. We can set this option to 0 to suppress the label.
19	sliderlength	It represents the length of the slider window along the length of the scale. The default is 30 pixels. However, we can change it to the appropriate value.
20	state	The scale widget is active by default. We can set this to DISABLED to make it unresponsive.
21	takefocus	The focus cycles through the scale widgets by default. We can set this option to 0 if we don't want this to happen.
22	tickinterval	The scale values are displayed on the multiple of the specified tick interval. The default value of the tickinterval is 0.

23	to	It represents a float or integer value that specifies the other end of the range represented by the scale.
24	troughcolor	It represents the color of the through.
25	variable	It represents the control variable for the scale.
26	width	It represents the width of the through part of the widget.

Methods

SN	Method	Description
1	get()	It is used to get the current value of the scale.
2	set(value)	It is used to set the value of the scale.

Example

```

from tkinter import *

def select():
    sel = "Value = " + str(v.get())
    label.config(text = sel)

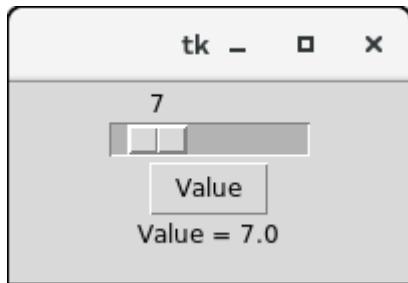
top = Tk()
top.geometry("200x100")
v = DoubleVar()
scale = Scale( top, variable = v, from_ = 1, to = 50, orient = HORIZONTAL)
scale.pack(anchor=CENTER)

btn = Button(top, text="Value", command=select)
btn.pack(anchor=CENTER)

label = Label(top)
label.pack()

top.mainloop() Output:

```



Python Tkinter Scrollbar

The scrollbar widget is used to scroll down the content of the other widgets like listbox, text, and canvas. However, we can also create the horizontal scrollbars to the Entry widget.

The syntax to use the Scrollbar widget is given below.

Syntax

1. `w = Scrollbar(top, options)`

A list of possible options is given below.

SN	Option	Description
1	activebackground	The background color of the widget when it has the focus.
2	bg	The background color of the widget.
3	bd	The border width of the widget.
4	command	It can be set to the procedure associated with the list which can be called each time when the scrollbar is moved.
5	cursor	The mouse pointer is changed to the cursor type set to this option which can be an arrow, dot, etc.
6	elementborderwidth	It represents the border width around the arrow heads and slider. The default value is -1.
7	highlightbackground	The focus highlightcolor when the widget doesn't have the focus.

8	highlightcolor	The focus highlightcolor when the widget has the focus.
9	highlightthickness	It represents the thickness of the focus highlight.
10	jump	It is used to control the behavior of the scroll jump. If it set to 1, then the callback is called when the user releases the mouse button.
11	orient	It can be set to HORIZONTAL or VERTICAL depending upon the orientation of the scrollbar.
12	repeatdelay	This option tells the duration up to which the button is to be pressed before the slider starts moving in that direction repeatedly. The default is 300 ms.
13	repeatinterval	The default value of the repeat interval is 100.
14	takefocus	We can tab the focus through this widget by default. We can set this option to 0 if we don't want this behavior.
15	troughcolor	It represents the color of the trough.
16	width	It represents the width of the scrollbar.

Methods

The widget provides the following methods.

SN	Method	Description
1	get()	It returns the two numbers a and b which represents the current position of the scrollbar.
2	set(first, last)	It is used to connect the scrollbar to the other widget w. The yscrollcommand or xscrollcommand of the other widget to this method.

Example

```
from tkinter import *
```

```
top = Tk()
```

```

sb = Scrollbar(top)
sb.pack(side = RIGHT, fill = Y)

mylist = Listbox(top, yscrollcommand = sb.set )

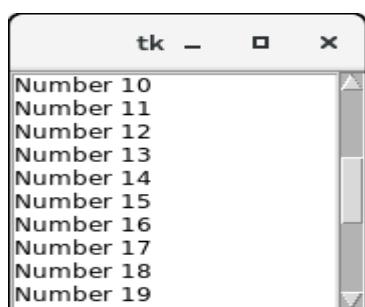
for line in range(30):
    mylist.insert(END, "Number " + str(line))

mylist.pack( side = LEFT )
sb.config( command = mylist.yview )

mainloop()

```

Output:



Python Tkinter Text

The Text widget is used to show the text data on the Python application. However, Tkinter provides us the Entry widget which is used to implement the single line text box.

The Text widget is used to display the multi-line formatted text with various styles and attributes. The Text widget is mostly used to provide the text editor to the user.

The Text widget also facilitates us to use the marks and tabs to locate the specific sections of the Text. We can also use the windows and images with the Text as it can also be used to display the formatted text.

The syntax to use the Text widget is given below.

Syntax

1. w = Text(top, options)

A list of possible options that can be used with the Text widget is given below.

SN	Option	Description
1	bg	The background color of the widget.
2	bd	It represents the border width of the widget.
3	cursor	The mouse pointer is changed to the specified cursor type, i.e. arrow, dot, etc.
4	exportselection	The selected text is exported to the selection in the window manager. We can set this to 0 if we don't want the text to be exported.
5	font	The font type of the text.
6	fg	The text color of the widget.
7	height	The vertical dimension of the widget in lines.
8	highlightbackground	The highlightcolor when the widget doesn't has the focus.
9	highlightthickness	The thickness of the focus highlight. The default value is 1.
10	highlighcolor	The color of the focus highlight when the widget has the focus.
11	insertbackground	It represents the color of the insertion cursor.
12	insertborderwidth	It represents the width of the border around the cursor. The default is 0.
13	insertofftime	The time amount in Milliseconds during which the insertion cursor is off in the blink cycle.
14	insertontime	The time amount in Milliseconds during which the insertion cursor is on in the blink cycle.
15	insertwidth	It represents the width of the insertion cursor.
16	padx	The horizontal padding of the widget.

17	pady	The vertical padding of the widget.
18	relief	The type of the border. The default is SUNKEN.
19	selectbackground	The background color of the selected text.
20	selectborderwidth	The width of the border around the selected text.
21	spacing1	It specifies the amount of vertical space given above each line of the text. The default is 0.
22	spacing2	This option specifies how much extra vertical space to add between displayed lines of text when a logical line wraps. The default is 0.
23	spacing3	It specifies the amount of vertical space to insert below each line of the text.
24	state	If the state is set to DISABLED, the widget becomes unresponsive to the mouse and keyboard unresponsive.
25	tabs	This option controls how the tab character is used to position the text.
26	width	It represents the width of the widget in characters.
27	wrap	This option is used to wrap the wider lines into multiple lines. Set this option to the WORD to wrap the lines after the word that fit into the available space. The default value is CHAR which breaks the line which gets too wider at any character.
28	xscrollcommand	To make the Text widget horizontally scrollable, we can set this option to the set() method of Scrollbar widget.
29	yscrollcommand	To make the Text widget vertically scrollable, we can set this option to the set() method of Scrollbar widget.

Methods

We can use the following methods with the Text widget.

SN	Method	Description

1	delete(startindex, endindex)	This method is used to delete the characters of the specified range.
2	get(startindex, endindex)	It returns the characters present in the specified range.
3	index(index)	It is used to get the absolute index of the specified index.
4	insert(index, string)	It is used to insert the specified string at the given index.
5	see(index)	It returns a boolean value true or false depending upon whether the text at the specified index is visible or not.

Mark handling methods

Marks are used to bookmark the specified position between the characters of the associated text.

SN	Method	Description
1	index(mark)	It is used to get the index of the specified mark.
2	mark_gravity(mark, gravity)	It is used to get the gravity of the given mark.
3	mark_names()	It is used to get all the marks present in the Text widget.
4	mark_set(mark, index)	It is used to inform a new position of the given mark.
5	mark_unset(mark)	It is used to remove the given mark from the text.

Tag handling methods

The tags are the names given to the separate areas of the text. The tags are used to configure the different areas of the text separately. The list of tag-handling methods along with the description is given below.

SN	Method	Description

1	tag_add(tagname, startindex, endindex)	This method is used to tag the string present in the specified range.
2	tag_config	This method is used to configure the tag properties.
3	tag_delete(tagname)	This method is used to delete a given tag.
4	tag_remove(tagname, startindex, endindex)	This method is used to remove a tag from the specified range.

Example

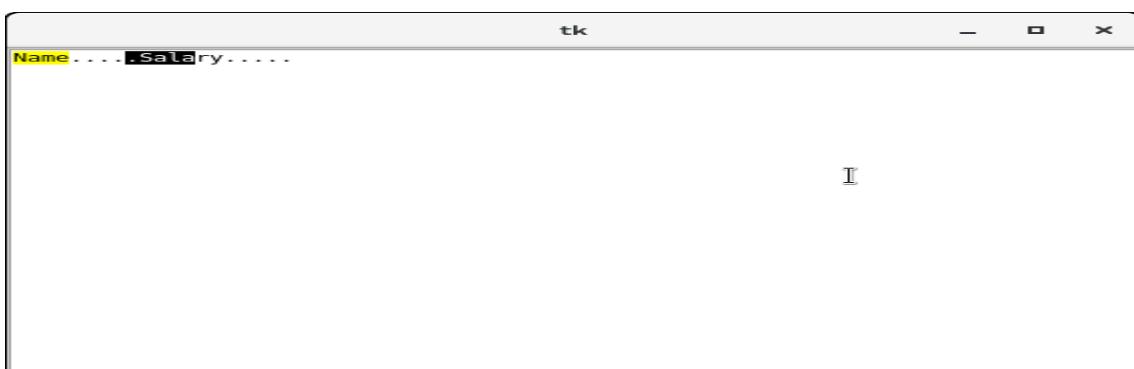
```
from tkinter import *

top = Tk()
text = Text(top)
text.insert(INSERT, "Name.....")
text.insert(END, "Salary.....")
text.pack()

text.tag_add("Write Here", "1.0", "1.4")
text.tag_add("Click Here", "1.8", "1.13")

text.tag_config("Write Here", background="yellow", foreground="black")
text.tag_config("Click Here", background="black", foreground="white")
```

top.mainloop() **Output:**



Python Tkinter Spinbox

The Spinbox widget is an alternative to the Entry widget. It provides the range of values to the user, out of which, the user can select the one.

It is used in the case where a user is given some fixed number of values to choose from.

We can use various options with the Spinbox to decorate the widget. The syntax to use the Spinbox is given below.

Syntax

1. w = Spinbox(top, options)

A list of possible options is given below.

Play VideoSN	Option	Description
1	activebackground	The background color of the widget when it has the focus.
2	bg	The background color of the widget.
3	bd	The border width of the widget.
4	command	The associated callback with the widget which is called each time the state of the widget is called.
5	cursor	The mouse pointer is changed to the cursor type assigned to this option.
6	disabledbackground	The background color of the widget when it is disabled.
7	disabledforeground	The foreground color of the widget when it is disabled.
8	fg	The normal foreground color of the widget.
9	font	The font type of the widget content.
10	format	This option is used for the format string. It has no default value.
11	from_	It is used to show the starting range of the widget.

12	justify	It is used to specify the justification of the multi-line widget content. The default is LEFT.
13	relief	It is used to specify the type of the border. The default is SUNKEN.
14	repeatdelay	This option is used to control the button auto repeat. The value is given in milliseconds.
15	repeatinterval	It is similar to repeatdelay. The value is given in milliseconds.
16	state	It represents the state of the widget. The default is NORMAL. The possible values are NORMAL, DISABLED, or "readonly".
17	textvariable	It is like a control variable which is used to control the behaviour of the widget text.
18	to	It specifies the maximum limit of the widget value. The other is specified by the from_ option.
19	validate	This option controls how the widget value is validated.
20	validatecommand	It is associated to the function callback which is used for the validation of the widget content.
21	values	It represents the tuple containing the values for this widget.
22	vcmd	It is same as validation command.
23	width	It represents the width of the widget.
24	wrap	This option wraps up the up and down button the Spinbox.
25	xscrollcommand	This option is set to the set() method of scrollbar to make this widget horizontally scrollable.

Methods

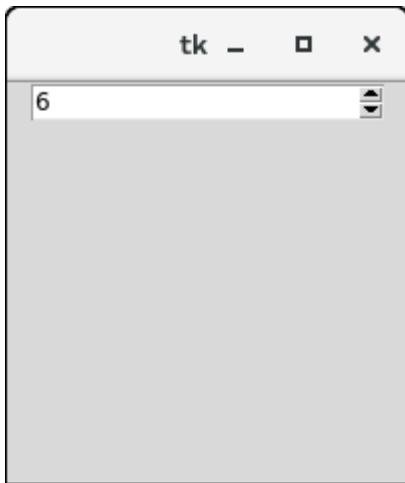
There are the following methods associated with the widget.

SN	Option	Description
1	delete(startindex, endindex)	This method is used to delete the characters present at the specified range.
2	get(startindex, endindex)	It is used to get the characters present in the specified range.
3	identify(x, y)	It is used to identify the widget's element within the specified range.
4	index(index)	It is used to get the absolute value of the given index.
5	insert(index, string)	This method is used to insert the string at the specified index.
6	invoke(element)	It is used to invoke the callback associated with the widget.

Example

```
from tkinter import *
top = Tk()
top.geometry("200x200")
spin = Spinbox(top, from_= 0, to = 25)
spin.pack()
top.mainloop()
```

Output:



Tkinter PanedWindow

The PanedWindow widget acts like a Container widget which contains one or more child widgets (panes) arranged horizontally or vertically. The child panes can be resized by the user, by moving the separator lines known as sashes by using the mouse.

Each pane contains only one widget. The PanedWindow is used to implement the different layouts in the python applications.

The syntax to use the PanedWindow is given below.

Syntax

1. w= PanedWindow(master, options)

A list of possible options is given below.

SN	Option	Description
1	bg	It represents the background color of the widget when it doesn't have the focus.
2	bd	It represents the 3D border size of the widget. The default option specifies that the trough contains no border whereas the arrowheads and slider contain the 2-pixel border size.

3	borderwidth	It represents the border width of the widget. The default is 2 pixel.
4	cursor	The mouse pointer is changed to the specified cursor type when it is over the window.
5	handlepad	This option represents the distance between the handle and the end of the sash. For the horizontal orientation, it is the distance between the top of the sash and the handle. The default is 8 pixels.
6	handlesize	It represents the size of the handle. The default size is 8 pixels. However, the handle will always be a square.
7	height	It represents the height of the widget. If we do not specify the height, it will be calculated by the height of the child window.
8 orient		The orient will be set to HORIZONTAL if we want to place the child windows side by side. It can be set to VERTICAL if we want to place the child windows from top to bottom.
9	relief	It represents the type of the border. The default is FLAT.
10	sashpad	It represents the padding to be done around each sash. The default is 0.
11	sashrelief	It represents the type of the border around each of the sash. The default is FLAT.

12	sashwidth	It represents the width of the sash. The default is 2 pixels.
13	showhandle	It is set to True to display the handles. The default value is false.
14	Width	It represents the width of the widget. If we don't specify the width of the widget, it will be calculated by the size of the child widgets.

Methods

There are the following methods that are associated with the PanedWindow.

SN	Method	Description
1	add(child, options)	It is used to add a window to the parent window.
2	get(startindex, endindex)	This method is used to get the text present at the specified range.
3	config(options)	It is used to configure the widget with the specified options.

Example

```
#!/usr/bin/python3
from tkinter import *

def add():
    a = int(e1.get())
    b = int(e2.get())
    leftdata = str(a+b)
    left.insert(1,leftdata)

w1 = PanedWindow()
w1.pack(fill = BOTH, expand = 1)
```

```

left = Entry(w1, bd = 5)
w1.add(left)

w2 = PanedWindow(w1, orient = VERTICAL)
w1.add(w2)

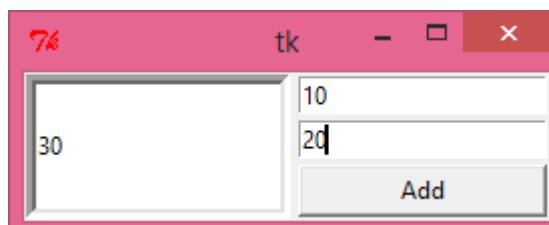
e1 = Entry(w2)
e2 = Entry(w2)

w2.add(e1)
w2.add(e2)

bottom = Button(w2, text = "Add", command = add)
w2.add(bottom)

```

`mainloop()` **Output:**



Tkinter LabelFrame

The `LabelFrame` widget is used to draw a border around its child widgets. We can also display the title for the `LabelFrame` widget. It acts like a container which can be used to group the number of interrelated widgets such as `Radiobuttons`.

This widget is a variant of the `Frame` widget which has all the features of a frame. It also can display a label.

The syntax to use the `LabelFrame` widget is given below.

Syntax

1. `w = LabelFrame(top, options)`

A list of options is given below.

SN	Option	Description
1	bg	The background color of the widget.
2	bd	It represents the size of the border shown around the indicator. The default is 2 pixels.
3	Class	The default value of the class is LabelFrame.
4	colormap	This option is used to specify which colormap is to be used for this widget. By colormap, we mean the 256 colors that are used to form the graphics. With this option, we can reuse the colormap of another window on this widget.
5	container	If this is set to true, the LabelFrame becomes the container widget. The default value is false.
6	cursor	It can be set to a cursor type, i.e. arrow, dot, etc. the mouse pointer is changed to the cursor type when it is over the widget.
7	fg	It represents the foreground color of the widget.
8	font	It represents the font type of the widget text.
9	height	It represents the height of the widget.
10	labelAnchor	It represents the exact position of the text within the widget. The default is NW(north-west)
11	labelwidget	It represents the widget to be used for the label. The frame uses the text for the label if no value specified.
12	highlightbackground	The color of the focus highlight border when the widget doesn't have the focus.
13	highlightcolor	The color of the focus highlight when the widget has the focus.
14	highlightthickness	The width of the focus highlight border.

15	padx	The horizontal padding of the widget.
16	pady	The vertical padding of the widget.
17	relief	It represents the border style. The default value is GROOVE.
18	text	It represents the string containing the label text.
19	width	It represents the width of the frame.

Example

```
#!/usr/bin/python3
from tkinter import *

top = Tk()
top.geometry("300x200")

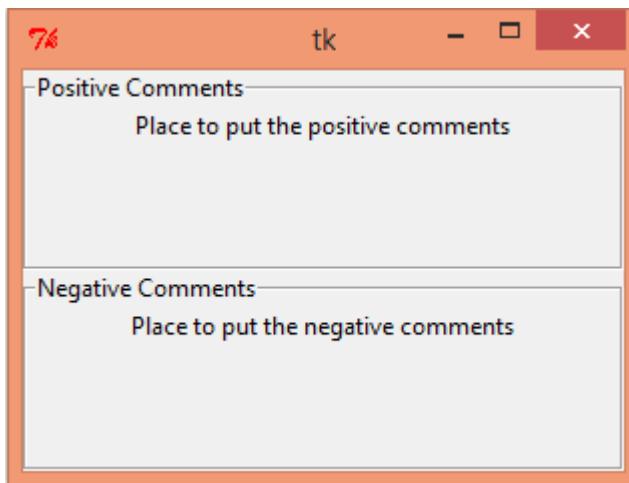
labelframe1 = LabelFrame(top, text="Positive Comments")
labelframe1.pack(fill="both", expand="yes")

toplable = Label(labelframe1, text="Place to put the positive comments")
toplable.pack()

labelframe2 = LabelFrame(top, text = "Negative Comments")
labelframe2.pack(fill="both", expand = "yes")

bottomlabel = Label(labelframe2, text = "Place to put the negative comments")
bottomlabel.pack()

top.mainloop() Output:
```



Tkinter messagebox

The messagebox module is used to display the message boxes in the python applications. There are the various functions which are used to display the relevant messages depending upon the application requirements.

The syntax to use the messagebox is given below.

Syntax

1. `messagebox.function_name(title, message [, options])`

Parameters

- **function_name:** It represents an appropriate message box function.
- **title:** It is a string which is shown as a title of a message box.
- **message:** It is the string to be displayed as a message on the message box.
- **options:** There are various options which can be used to configure the message dialog box.

The two options that can be used are default and parent.

1. default

The default option is used to mention the types of the default button, i.e. ABORT, RETRY, or IGNORE in the message box.

2. parent

The parent option specifies the parent window on top of which, the message box is to be displayed.

There is one of the following functions used to show the appropriate message boxes. All the functions are used with the same syntax but have the specific functionalities.

1. showinfo()

The showinfo() messagebox is used where we need to show some relevant information to the user.

Example

```
# !/usr/bin/python3

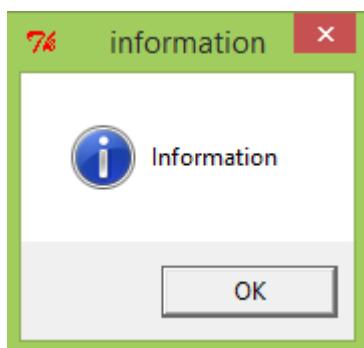
from tkinter import *
from tkinter import messagebox
top = Tk()

top.geometry("100x100")

messagebox.showinfo("information","Information")

top.mainloop()
```

Output:



2. showwarning()

This method is used to display the warning to the user. Consider the following example.

Example

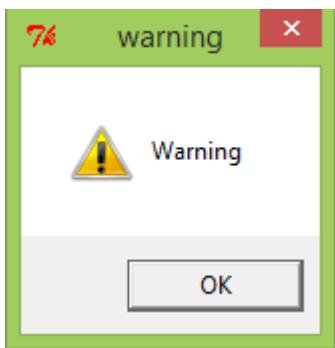
```
#!/usr/bin/python3
from tkinter import *

from tkinter import messagebox

top = Tk()
top.geometry("100x100")
messagebox.showwarning("warning","Warning")

top.mainloop()
```

Output:



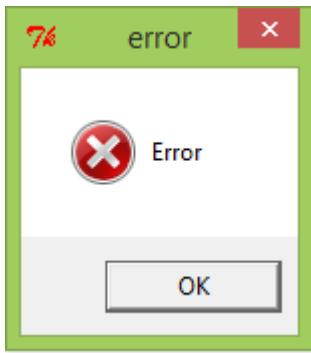
3. showerror()

This method is used to display the error message to the user. Consider the following example.

Example

```
#!/usr/bin/python3
from tkinter import *
from tkinter import messagebox

top = Tk()
top.geometry("100x100")
messagebox.showerror("error","Error")
top.mainloop() Output:
```



4. askquestion()

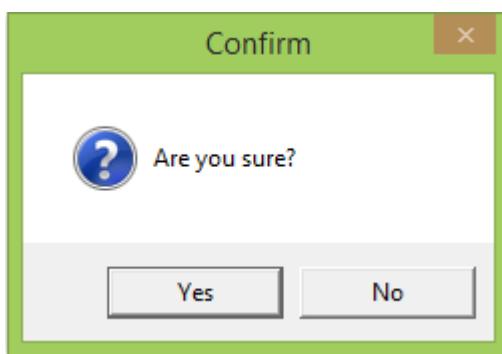
This method is used to ask some question to the user which can be answered in yes or no. Consider the following example.

Example

```
#!/usr/bin/python3
from tkinter import *
from tkinter import messagebox

top = Tk()
top.geometry("100x100")
messagebox.askquestion("Confirm","Are you sure?")
top.mainloop()
```

Output:



5. askokcancel()

This method is used to confirm the user's action regarding some application activity. Consider the following example.

Example

```
#!/usr/bin/python3
from tkinter import *
from tkinter import messagebox

top = Tk()
top.geometry("100x100")
messagebox.askokcancel("Redirect","Redirecting you to www.acytech.com")
top.mainloop()
```

Output:

6. askyesno()

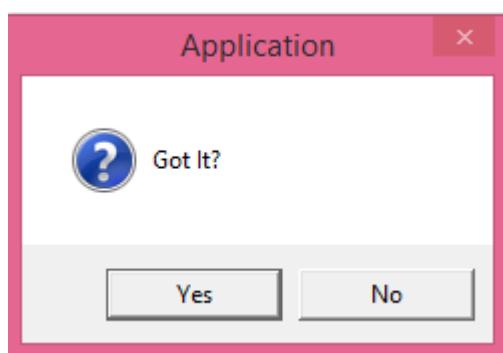
This method is used to ask the user about some action to which, the user can answer in yes or no. Consider the following example.

Example

```
#!/usr/bin/python3
from tkinter import *
from tkinter import messagebox

top = Tk()
top.geometry("100x100")
messagebox.askyesno("Application","Got It?")
top.mainloop()
```

Output:



7. askretrycancel()

This method is used to ask the user about doing a particular task again or not. Consider the following example.

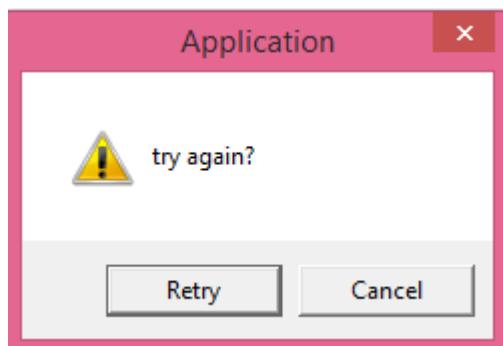
Example

```
#!/usr/bin/python3
from tkinter import *
from tkinter import messagebox

top = Tk()
top.geometry("100x100")
messagebox.askretrycancel("Application","try again?")

top.mainloop()
```

Output:



Python CGI Programming

In the Python CGI programming, we will learn how we can run the Python Script on the web; we will learn how Python file can be executed as CGI Script and discuss the its configuration of web-browser to Python script run as a CGI. Also, we will learn the following topics:

What is CGI?

The word CGI is the acronyms of the "**Common Gateway Interface**", which is used to define how to exchange information between the web server and a custom script. The NCSA officially manages the CGI scripts.

The Common Gateway Interface is a standard for external gateway programs to interface with the server, such as HTTP Servers.

In simple words, it is the collection of methods used to set up a dynamic interaction between the server, and the client application. When a client sends a request to the webserver, the CGI programs execute that particular request and send back the result to the webserver.

The users may submit the information in web browser by using HTML <form> or <isindex> element. There is a server's special directory called cgi-bin, where cgi script is generally stored.

When a client makes a request, the server adds the additional information to request.

This additional information can be the hostname of the client, the query string, the requested URL, and so on. The webserver executes and sends the output to the web browser (or other client application).

Sometimes the server passes the data by a query string that is the part of the URL. A query string may not be conventionally a hierarchical path structure such as the following link.

<https://www.google.com/search?q=wikipedia&oq=wiki&aqs=chrome.1.69i57j0l6j5.3046j0j7&sourceid=chrome&ie=UTF-8>

Python provides the CGI module, which helps to debug the script and also support for the uploading files through a HTML form.

So here the question arises what does a Python CGI script output look like? The HTTP server gives back the output as two sections separated by a blank line. The first section grasps the number of headers, notifying the client what kind of data is following.

Let's understand the following example of generate the minimal header section in the Python CGI programming.

Example -

```
# HTML is following
print("Content-Type: text/html")
# blank line, end of headers
print()
```

In the above example, the first statement says that, the server that html code follows; the second blank line indicates the header is ended here. Let's understand the following example of generating the minimal header section in the Python CGI programming.

Example -

```
print("<title> This is a CGI script output</title>")
print("<h1>This is my first CGI script</h1>")
print("Hello, Welcome!")
```

Web Browsing

Before understanding the CGI concepts, we should know the internal process of web page or URL when we click on the given link.

- o The client (web browser) communicates with the HTTP server and asks for the URL i.e.,

- filename.
- If the web browser finds that requested file, then it sends back to the client (web browser),
- otherwise it sends the error message to the client as error file.
- Here the responsibility of the web browser to display either the received file or an error message.

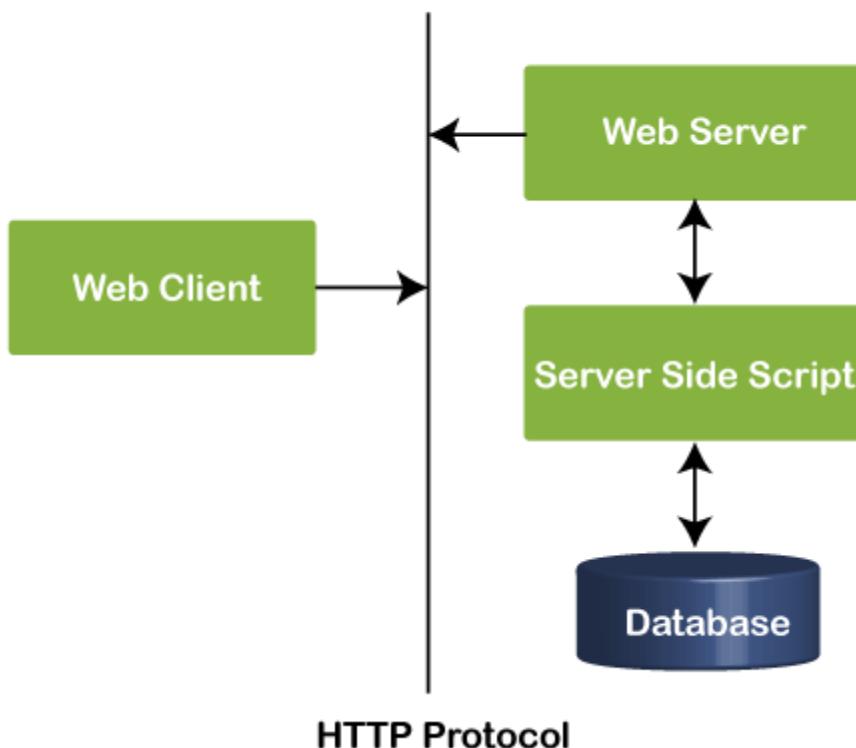
However, we can set a HTTP server so that whenever user requests in a particular dictionary, then it should be sent to the client; instead, it executed as a program and whatever the result is sent back for the client to display. This process is called the Common Gateway Interface or CGI and the programs are called CGI scripts. We can write the CGI programs as Python Script, PERL Script, Shell Script, C or C++, programs, etc.

Configure Apache Web server for CGI

We need to configure the Apache Web server in order to run the CGI script on the server.

CGI Architecture

CGI Architecture Diagram



Using the cgi module

Python provides the **cgi** module, which consists of many useful built-in functions. We can use them by importing the **cgi** module.

```
import cgi
```

Now, we can write further script.

```
import cgi  
cgitb.enable()
```

The above script will stimulate an exception handler to show the detailed report in the web browser of occurred errors. We can also save the report by using the following script.

```
import cgitb  
cgitb.enable(display=0, logdir="/path/to/logdir")
```

The above feature of the cgi module is helpful during the script development. These reports help us to debug the script effectively. When we get the expected output, we can remove this.

Previously, we have discussed the users save information using the form. So how can we get that information? Python provides the **FieldStorage** class. We can apply the encoding keyword parameter to the document if the form contains the non-ASCII character. We will find the content <META> tag in the <HEAD> section in our HTML document.

The FieldStorage class reads the form information from the standard input or the environment.

A **FieldStorage** instance is the same as the Python dictionary. We can use the **len()** and all dictionary function in the FieldStorage instance. It overlooks fields with empty string values. We can also consider the empty values using the optional keyword parameter **keep_blank_values** by setting **True**.

Example -

```
form = cgi.FieldStorage()
if ("name" not in form or "addr" not in form):
    print("<H1>Error</H1>")
    print("Please enter the information in the name and address fields.")
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
#Next lines of code will execute here...
```

In the above example, we have used the form **["name"]**, here name is **key**. This is used to extract the value which is entered by the user.

We can use the **getvalue()** method to fetch the string value directly. This function also takes an optional second argument as a default. If the key is not present, it returns the default value.

If the submitted form data have more than one field with the same name, we should use the **form.getlist()** function. It returns the list of strings. Look at the following code, we add the any number of username field, which is separated by commas.

```
value1 = form.getlist("username")
usernames1 = ",".join(value)
```

If the field is uploaded file, then it can be accessed by **value** attribute or the **getvalue()** method and read that uploaded file in bytes. Let's understand the following code if user upload the file.

Example -

```
file_item = form["userfile"]
if (fileitem.file):
    # It represent the uploaded file; counting lines
    count_line = 0
    while(True):
        line = fileitem.file.readline()
        if not line: break
```

```
count_line = count_line + 1
```

Sometimes an error can interrupt the program while reading the content of the uploaded file (When a user clicks on Cancel Button or Back Button). FieldStorage class provides the **done** attribute to set to the value -1.

If we submit the form in the "**old**" format, the item will be instances of the class **MiniFieldStorage**. Generally, the form is submitted via POST and contains a query string with both **FieldStorage** and **MiniStorage** items.

Here, we are defining the **FieldStorage** attribute in the following table.

Attributes	Description
Name	It represents the field name.
Filename	It represents Client side filename.
File	It is a file(-like) object from which we can read data as bytes.
Value	It is a string type value. Use for file uploads, reads the file and returns bytes.
Type	It is used to display the content-type.
Header	It is a dictionary type object which contains all headers.

The **FieldStorage** instance uses the many built-in methods to manipulate the users' data. Below are a few FieldStorage's methods.

FieldStorage Methods:

Methods	Description
---------	-------------

getfirst()	It returns the first value received.
getlist()	It returns the list of the received values.
getvalue()	It is a dictionary get() method.
keys()	It is dictionary keys() method
make_file()	It returns a readable and writable file.

Running First Python File as CGI

In this section, we will discuss how can run the CGI program over the server. Before doing this, we must ensure that our system has the following configuration-

- Apache Server
- Python

If you already have the XAMPP server in your system then you can skip this part.

Installing the XAMPP server

XAMPP stands for cross-platform, Apache, MySQL, PHP, and Perl, and it provides the localhost server to test or deploy websites. Generally, it gives the two essential components for its installation, first is - Apache that creates a local server and MySQL, which we can use a database. Follow the below steps to install xampp.

Step - 1: Visit its official website (<https://www.apachefriends.org/download.html>) and download the latest version.

It will take away on the webserver and start installing all packages and files.

).

First CGI Program

We have created a new folder called **example** in xampp's htdocs folder. Then, we write a Python script, which includes the HTML tags. Let's see the following directory structure and the demo.py file.

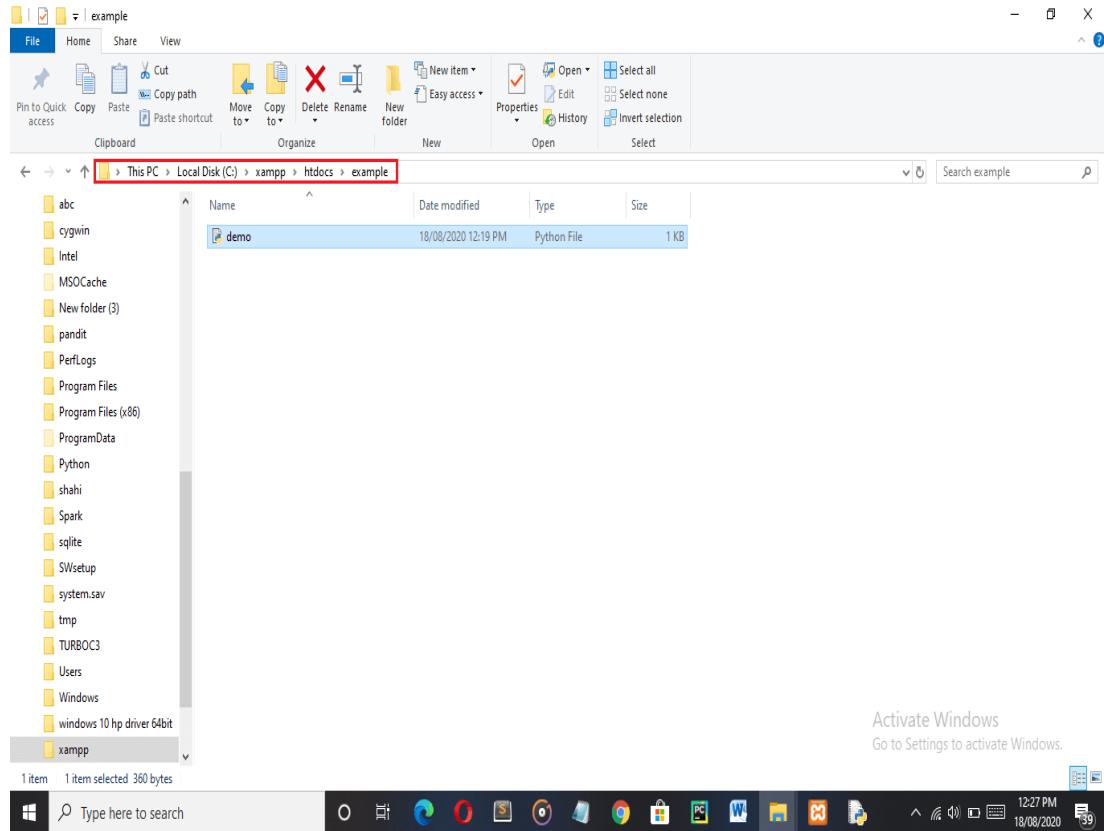


demo.py

```
#C:\Users\DEVANSH SHARMA\AppData\Local\Programs\Python\Python37

print ("Content-Type : text/html\r\n\r\n")
# then come the rest hyper-text documents
print ("<html>")
print ("<head>")
print ("<title>My First CGI-Program </title>")
print ("<head>")
print ("<body>")
print (""
<h1>This is my CGI script </h1>
")
print ("</body>")
print ("</html>")
```

And, its directory structure as follow.



Type the localhost/example/demo.py into the web browser. It will display the following output.

This is my first CGI program.

Note - We need to start the Apache server then execute the CGI script. Our script demo.py will run on host 127.0.0.1 by default.

Let's understand another example of CGI script.

Example - 2:

```
#!/usr/bin/python3
# Importing the 'cgi' module
import cgi
```

```

print("Content-type: text/html\n\n")
print("<html><body>")
print(" <h1> Hello Program! </h1> ")
# Using the inbuilt methods
form = cgi.FieldStorage()
if form.getvalue("name")
    name = form.getvalue("name")
    print(" <h1>Hello" +name+"! Thanks for using my script!</h1> ")
if form.getvalue("happy")
    print(" Yayy! I'm happy too! ")
if form.getvalue("sad")
#!/usr/bin/python3
# Importing the 'cgi' module
import cgi
print("Content-type: text/html\n\n")
print("<html><body>")
print(" <h1> Hello Program! </h1> ")
# Using the inbuilt methods
form = cgi.FieldStorage()
if form.getvalue("name")
    name = form.getvalue("name")
    print(" <h1>Hello" +name+"! Thanks for using my script!</h1> ")
if form.getvalue("happy")
    print(" Yayy! I'm happy too! ")
if form.getvalue("sad")
    print(" Oh no! Why are you sad? ")
# Using HTML input and forms method
print(" <form method='post' action='demo2.py'> ")
print(" Name: <input type='text' name='name' /> ")
print("<input type='checkbox' name='happy' /> Happy")
print("<input type='checkbox' name='sad' /> Sad")
print("<input type='submit' value='Submit' />")
print("</form>")
print("</body></html>")

```

Structure of a Python CGI Program

Let's understand the following structure of the program.

- The CGI script must contain two sections which separated by a blank line.
- The header must be in the first section, and the second section will contain the kind of data that will be used during the execution of the script.

Legend of Syntaxes

When scripting a CGI program in Python, take note of the following commonly used syntaxes.

HTML Header

In the above program, the line **Content-type:text/html\r\n\r\n** is a portion of the HTTP, which we will use in our CGI programs.

HTTP Field Name: Field Content

For Example

Content-type: text/html\r\n\r\n

Sr.	Header	Description
1.	Content-type	It is a MIME string that is used to define the format of the file being returned.
2.	Expires: Date	It displays the valid date information.
3.	Location: URL	The URL that is returned by the server.
4.	Last-modified: Date	It displays the date of the last modification of the resource.
5.	Content-length: N	This information is used to report the estimated download time for a file.
6.	Set-Cookies: String	It is used to set the cookies by using string.

CGI Environment Variables

We should remember the following CGI environment variable along with the HTML syntax. Let's understand the commonly used CGI environment variables.

- **CONTENT_TYPE** - It describes the data and type of content.
- **CONTENT_LENGTH** - It defines the length of a query or information.
- **HTTP_COOKIE** - It is used to return the cookie, which is set by the user in the current scenario.
- **HTTP_USER_AGENT** - This variable is used to display the type of browser that the user is currently using.
- **REMOTE_HOST** - It is used to describe the path of the CGI scripts.
- **PATH_INFO** - This variable is used to define the path of the CGI script.
- **REMOTE_ADDR** - We can define the IP address of the visitor by using it.
- **REQUEST_METHOD** - It is used to make a request either via POST or GET.

Functions of Python CGI Programming

The CGI module provides the many functions to work with the cgi. We are defining a few important functions as follows.

- **parse(fp = None, environ = os.environ, keep_blanks_values = False, strict_parsing = False)** - It is used to parse a query in the environment. We can also parse it using a file, the default for which is **sys.stdin**.
- **parse_qs(qs, keep_blank_values = False, strict_parsing = False)** - While this is denigrated, Python uses it for `urllib.parse.parse_qs()` instead.
- **parse_qsl(qs, keep_blank_value = False, strict_parsing = False)** - This is also denigrated, and maintains of for backward-compatibility.
- **parse_multipart(fb, pdict)** - It is used to parse input of type multipart/form-data for file

uploads. The first argument is the **input file**, and the second argument is a dictionary holding in the other parameters in the content-type header.

- **parse_header(string)** - It is used to parse the header. It permits the MIME header into the main value and a dictionary of parameters.
- **test()** - It is used to test a CGI script, and we can use it in our program. It will generally write minimal HTTP headers.
- **print_form(form)** - It formats a form in HTML.
- **print_directory()** - It formats the current directory in HTML.
- **escape(s, quote = False)** - The **escape()** function is used to convert characters '<', '>', and '&' in the string's to HTML safe sequence.

Debugging CGI Scripts

First, we need to check the trivial installation error. Most of the time, the error occurs during the installation of the CGI script. In the beginning, we must follow the installation instructions and try installing a copy of this module file **cgi.py** as a CGI script.

Next, we can use **test()** function from the script. Type the following code with a single statement.

```
cgi.test()
```

Advantages of CGI Programming

There are various advantages of using CGI programming. Below are some of its advantages.

- They are language independent. We can use CGI programs with any programming languages.
- The CGI programs can work on almost any web server and the portable.
- They are portable.
- The CGI programs can perform both simple and complex tasks; means they are fairly scalable.

- The CGIs can increase the dynamic communication in the web applications.
- The CGIs can also be profitable, if we use them in development; they reduce the development costs and maintenance costs.
- The CGIs takes less time to process the requests.

Disadvantages of CGI

Consider the following disadvantages of CGI.

- CGI programs are too complex and hard to debug.
- When we initiate the program, the interpreter has to evaluate a CGI script in each initiation.
- The result is, creates a lot of traffic because are many requests from the side of the client-server.
- CGI programs are quite vulnerable, as most of them are free and easily available without the server security.
- CGI uses a lot of process time.
- During the page load, the data doesn't store in the cache memory.
- There are huge extensive codebases, most of it Perl.

Common Problems and Solutions

We can face problems during the implement the CGI script on the server. We have listed below the few common problems and their solutions.

- First of all, check the installation instructions. Most of the problems occur during the installation of server. Follow the installation guide properly.
- Check the HTTP server's log file. The **tail -f logfile** in a separated window may be valuable.
- In the CGI, it is possible to display the progress report on the client's screen of running

requests. Most HTTP servers save the output from the CGI script until the script is finished.

- Before executing the file, check the syntax error in your script, following as

python script.py.

- If the script does not have any syntax error then import the library such as ***import cgitb;*** ***cgitb.enable()*** to the top of the script.
- The absolute path must be included when importing the external program. The path is usually not set to a very useful value in a CGI script.
- When reading or writing external files, make sure that they can be read or written by the user under which your CGI script will be running. This is authorized user id where the script file in which the web server is running or some specified userid for a web server.
- It should be remembered that the CGI script must not set in ***set-uid*** It won't work on most systems, and also a security liability.