Lists

A list is a collection of arbitrary objects, somewhat akin to an array in many other programming languages but more flexible. Lists are defined in Python by enclosing a comma-separated sequence of objects in square brackets ([]). List is a versatile datatype available in Python. Basically a python list is comma-separated values which are called items. List in python is written within square brackets. Interestingly it' s not necessary for items in a list to be of same types. A list is a collection which is ordered and changeable.

Features :

- Lists Are Ordered i.e. its elements can be accessed by using Index.
- Lists Can Contain Arbitrary Objects.
- List Elements Lists Can Be Nested.
- Lists Are Mutable.
- Lists Are Dynamic.

Syntax:

```
<list_name>=[value1,value2,value3,...,valueN]
```

Example:

Program

```
lst=["Amit","Sumit","Gopal","Harry"]
print(lst)
for n in lst:
    print(n)
```

Output

```
['Amit','Sumit','Gopal','Harry
']
Amit
Sumit
Gopal
Harry
```

Example:

Program

```
lst=[4,3,1,6,2]
s=0
for n in lst:
    s=s+n
print("Sum is ",s)
```

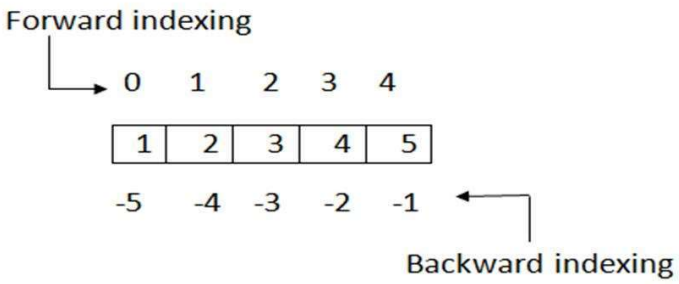
Output

```
Sum is  16
```

List Indexing:

Individual elements in a list can be accessed using an index in square brackets. This is exactly analogous to accessing individual characters in a string. List indexing is zero-based as it is with strings.

List indexing in Python is zero-based: the first item in the list has index 0, the next has index 1, and so on. The index of the last item will be the length of the list minus one.



We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4.

Trying to access an element other that this will raise an IndexError. The index must be an integer. We can't use float or other types, this will result into TypeError.

Syntax to get a value:

```
<list_name>[index]
```

Syntax to get a sublist:

```
<list_name>[indexstart: indexstop]
```

Example:

Program

```
Lst = [ 4, 3, 1, 6, 2, 9, 0 ]
print( lst[ 2 ] )
print( lst[ 0 ] )
print( lst[ -1 ] )
print( lst[ 2 : 5 ] )
```

Output

```
1
4
0
[1,6,2]
```

Lists Constructor:

The list() constructor returns a list in Python. The list() constructor takes a single argument an object that could be a sequence (string, tuples) or collection (set, dictionary) or any iterator object.

Syntax to get a value:

```
<list_name>=list(iterable)
```

Example:

Program

```
vowelString = 'AEIOU'
lst=list(vowelString)
print(lst)
```

Output

```
['A','E','I','O','U']
```

Lists Operator:

Python provides operator that work with lists.

operator	Description
+	operator to combine two lists.
*	repeats a list for the given number of times.
In	test if an item exists in a list or not

Example:

Program

```
lst1=['Amit','Gopal','Sumit']
lst2=['Harry','Raj']

print(lst1+lst2)

print(lst1*2)

print("Gopal" in lst1)
```

Output

```
['Amit','Gopal','Sumit','Harry','Raj'
]

['Amit','Gopal','Sumit','Amit','Gopal
','Sumit']

True
```

Lists Functions:

Python provides many functions that are built-in that work with lists.

Function	Description
`len(object)`	Returns the total number of items in a list
`str(object)`	Returns the list item as string.
`max(object)`	returns the largest item in an list.
`min(object)`	returns the smallest item in an list.

Example:

Program

```
lst=[23,12,45,68,47,12]
print(len(lst))
print(str(lst))
print(max(lst))
print(min(lst))
```

Output

```
6
[23, 12, 45, 68, 47, 12]
68
12
```

Lists method:

Python supplies several built-in methods that can be used to modify lists.

Function	Description
`lst.append(obj)`	The append() method adds an item to the end of the list.
`lst.extend(iterable)`	The extend() extends the list by adding all items of a list (passed as an argument) to the end
`lst.insert(index, obj)`	inserts object <obj> into list a at the specified <index>.
`lst.remove(obj)`	remove(obj) removes object <obj> from list. If obj isnt in a, an exception is raised
`lst.pop(index=-1)`	pop() simply removes the last item in the list
`lst.index(element)`	The index() method searches an element in the list and returns its index.
`list.count(element)`	The count() method returns the number of occurrences of an element in a list.
`lst.copy()`	The copy() method returns a shallow copy of the list.
`lst.clear()`	The clear() method removes all items from the list.
`lst.sort()`	The sort() method sorts the elements of a given list.
`lst.reverse()`	The reverse() method reverses the elements of a given list.

Example:

Program

```
lst=[23,12,45,12]
print(lst)

lst.append(14)
print(lst)

lst.extend([23,45])
print(lst)

lst.insert(2,35)
print(lst)

lst.remove(12)
print(lst)

lst.pop()
print(lst)

print(lst.index(12))

print(lst.count(23))

data=lst.copy()
print(data)

lst.sort()
print(lst)

lst.reverse()
print(lst)

lst.clear()
print(lst)
```

Output

```
[23, 12, 45, 12]

[23, 12, 45, 12, 14]

[23, 12, 45, 12, 14, 23, 45]

[23, 12, 35, 45, 12, 14, 23,
45]

[23, 35, 45, 12, 14, 23, 45]

[23, 35, 45, 12, 14, 23]

3

2

[23, 35, 45, 12, 14, 23]

[12, 14, 23, 23, 35, 45]

[45, 35, 23, 23, 14, 12]

[]
```

Nested List :

A list can contain sublists, which in turn can contain sublists themselves, and so on to arbitrary depth. If required a list can contains items of type list as well as other collection types in it.

Syntax:

```
<list_name>=[[value1,value2...],[value1,...],valueN]

                OR

<list_name 1>=[value1,value2,valueN]

<list_name 2>=[value1,value2,valueN]

<list_name 1>=[list_name 1,list_name 2]
```

Example:

Program

```
nlst=[[1, 3, 5], [2, 4, 6, 3]]
s=0
for lst in nlst:
    for n in lst:
      s=s+n
print("Sum is ",s)
```

Output

```
Sum is 24
```

Example:

Program

```
slst=[["jan-2017","Amit",45000],
    ["jan-2017","Gopal",15000],
    ["feb-2017","Amit",25000]]
s=0
for lst in slst:
   s=s+ lst[2]
print("Total sales is ",s)
```

Output

```
Total sales is  85000
```

Tuple

A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas, in a list, elements can be changed. Tuple is a data structure which is used to store Collection of Items. if the element is itself a mutable datatype like list, its nested items can be changed.

Features :

- Tuple Are Ordered i.e Can Be Accessed by Index
- Tuples Are Immutable.

Syntax:

```
<tuple_name>=(value1,value2,value3,...,valueN)
```

Example:

Program

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

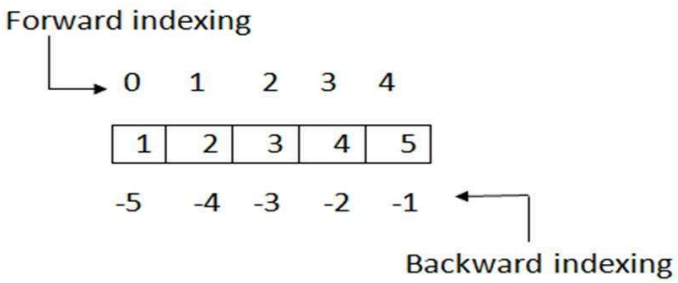
Output

```
('apple', 'banana', 'cherry')
```

Tuple Indexing:

Individual elements in a Tuple can be accessed using an index in square brackets. This is exactly analogous to accessing individual characters in a string. Tuple indexing is zero-based as it is with strings.

Tuple indexing in Python is zero-based: the first item in the list has index 0, the next has index 1, and so on. The index of the last item will be the length of the tuple minus one.



We can use the index operator [] to access an item in a Tuple. Index starts from 0. So, a Tuple having 5 elements will have index from 0 to 4.

Trying to access an element other that this will raise an IndexError. The index must be an integer. We can't use float or other types, this will result into TypeError.

Syntax to get a value:

```
<tuple_name>[index]
```

Syntax to get a sublist:

```
<tuple_name>[indexstart: indexstop]
```

Example:

Program

```
tp = ("apple","banana","cherry","mango")
print( tp[ 2 ] )
print( tp[ 0 ] )
print( tp[ -1 ] )
print( tp[ 1 : 3 ] )
```

Output

```
cherry
apple
mango
('banana', 'cherry')
```

Tuple Constructor:

In Python, a tuple is an immutable sequence type. One of the ways of creating tuple is by using the tuple() construct. an iterable (list, range, etc.) or an iterator object.

Syntax :

```
<tuple_name>=tuple(iterable)
```

Example:

Program

```
vowelString = 'AEIOU'
tp=tuple(vowelString)
print(tp)
```

Output

```
('A','E','I','O','U')
```

Tuple Operator:

Python provides operator that work with tuple.

operator	Description
+	operator to combine two tuple.
*	repeats a list for the given number of times.
In	test if an item exists in a list or not

Example:

Program

```
tpl1=('Amit','Gopal','Sumit')
tpl2=('Harry','Raj')

print(tpl1+tpl2)

print(tpl1*2)

print("Gopal" in tpl1)
```

Output

```
('Amit', 'Gopal', 'Sumit', 'Harry',
'Raj')

('Amit', 'Gopal', 'Sumit', 'Amit',
'Gopal', 'Sumit')

True
```

Tuple Functions:

Python provides many functions that are built-in that work with Tuple.

Function	Description
len(object)	Returns the total number of items in a tuple
str(object)	Returns the tuple item as string.
max(object)	returns the largest item in an tuple.
min(object)	returns the smallest item in an tuple.

Example:

Program

```
tpl=(23,12,45,68,47,12)
print(len(tpl))
print(str(tpl))
print(max(tpl))
print(min(tpl))
```

Output

```
6
(23, 12, 45, 68, 47, 12)
68
12
```

Tuple method:

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Function	Description
`tpl.count(x)`	Returns the number of items x
`tpl.index(x)`	Returns the index of the first item that is equal to x

Example:

Program

```
tpl=(23,12,45,68,47,12)

print(tpl.count(12))

print(tpl.index(12))
```

Output

```
2

1
```

Sets

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed). However, the set itself is mutable. We can add or remove items from it. Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

A Python set is similar to mathematical Set.The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table. There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.

Features :

- Sets are unordered.
- Set elements are unique. Duplicate elements are not allowed.
- A set itself may be modified, but the elements contained in the set must be of an immutable type.

Syntax:

```
<set_name>={value1,value2,value3,...,valueN}
```

Example:

Program

```
thissets = {"apple", "banana", "cherry"}
print(thissets)
```

Output

```
{'apple','banana','cherry'}
```

set Constructor:

The set() builtin creates a Python set from the given iterable. set() takes a single optional parameter iterable (optional) a sequence (string, tuple, etc.) or collection (set, dictionary, etc.) or an iterator object to be converted into a set. set() returns an empty set if no parameters are passed and a set constructed from the given iterable parameter.

Syntax:

```
<set_name>=set(iterable)
```

Example:

Program

```
lst=["Mon","Tue","Wed","Thu","Fri","Sat","Sun
"]
Days=set(lst)
print(Days)
```

Output

```
{"Mon","Tue","Wed","Thu","Fri"
,"Sat","Sun"}
```

Set Operator:

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

operator	Description
|	Union of A and B is a set of all elements from both sets.
&	Intersection of A and B is a set of elements that are common in both sets.
-	Difference of A and B (A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of element in B but not in A.
^	Symmetric Difference of A and B is a set of elements in both A and B except those that are common in both.

Example:

Program

```
daysA={"Mon","Tue","Wed","Sat","Sun"}
daysB={"Thu","Fri","Sat","Sun"}

print(daysA | daysB)

print(daysA & daysB)

print(daysA - daysB)

print(daysA ^ daysB)
```

Output

```
{'Fri', 'Sat', 'Thu', 'Sun',
'Wed', 'Mon', 'Tue'}

{'Sun', 'Sat'}

{'Mon', 'Wed', 'Tue'}

{'Fri', 'Thu', 'Wed', 'Mon',
'Tue'}
```

set Functions:

Python provides many functions that are built-in that work with set.

Function	Description
`len(object)`	Returns the total number of items in a set
`str(object)`	Returns the set item as string.
`max(object)`	returns the largest item in an set.
`min(object)`	returns the smallest item in an set.
`sum(object)`	The sum() function adds the items of an iterable and returns the sum.
`sorted(iterable, reverse=False)`	The sorted() function returns a sorted list from the items in an iterable. iterable - A sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any other iterator.reverse (Optional) - If True, the sorted list is reversed (or sorted in descending order). Defaults to False if not provided.

Example:

Program

```
num={23,12,45,56,35,31,21}

print(len(num))

print(str(num))

print(max(num))

print(min(num))

print(sum(num))

print(sorted(num))

print(sorted(num,reverse=True))
```

Output

```
7

{35, 12, 45, 21, 23, 56, 31}

56

12

223

[12, 21, 23, 31, 35, 45, 56]

[56, 45, 35, 31, 23, 21, 12]
```

Set method:

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with set objects.

Function	Description
set.add(x)	The set add() method adds a given element to a set. If the element is already present, it doesn't add any element.
set.remove(x)	The remove() method searches for the given element in the set and removes it.
set.discard(x)	The discard() method removes a specified element from the set (if present).
set.update(iterable)	The Python set update() method updates the set, adding items from other iterables.
set.pop()	The pop() method removes an arbitrary element from the set and returns the element removed.
set.copy()	The copy() method returns a shallow copy of the set.
set.clear()	The clear() method removes all elements from the set.
set.difference(setB)	The difference() method returns the set difference of two sets.
set.intersection(setB)	The intersection() method returns a new set with elements that are common to all sets.
set.symmetric_difference (setB)	The Python symmetric_difference() method returns the symmetric difference of two sets.
set.union(setB)	The Python set union() method returns a new set with distinct elements from all the sets.
set.isdisjoint(setB)	The isdisjoint() method returns True if two sets are disjoint sets. If not, it returns False
set.issubset(setB)	The issubset() method returns True if all elements of a set are present in another set (passed as an argument). If not, it returns False.

Example:

Program

```
numA={23,12,45,56,35,31,21}
numB={23,12,89,78,35,87}

numA.add(26)
print(numA)

numA.remove(26)
print(numA)

numA.update([26,32])
print(numA)

numA.discard(26)
print(numA)

numA.pop()
print(numA)

print(numA.difference(numB))

print(numA.intersection(numB))

print(numA.symmetric_difference(numB))

print(numA.union(numB))

print(numA.isdisjoint(numB))

print(numA.issubset(numB))

numC=numA.copy()
print(numC)

numA.clear()
print(numA)
```

Output

```
{35, 12, 45, 21, 23, 56, 26, 31}

{35, 12, 45, 21, 23, 56, 31}

{32, 35, 12, 45, 21, 23, 56, 26, 31}

{32, 35, 12, 45, 21, 23, 56, 31}

{35, 12, 45, 21, 23, 56, 31}

{56, 21, 45, 31}

{35, 12, 23}

{45, 78, 21, 87, 56, 89, 31}

{35, 12, 45, 78, 21, 87, 23, 56, 89, 31}

False

False

{35, 21, 23, 56, 12, 45, 31}

set()
```

Frozen sets

Python provides another built-in type called a frozenset, which is in all respects exactly like a set, except that a frozenset is immutable. You can perform non-modifying operations on a frozensetFrozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets.

Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary. Frozen set is just an immutable version of a Python set object. While elements of a set can be modified at any time, elements of frozen set remains the same after creation.

Frozensets can be created using the function frozenset().The frozenset() method returns an immutable frozenset object initialized with elements from the given iterable. The frozenset() method optionally takes a single parameter

iterable (Optional) - the iterable which contains elements to initialize the frozenset with. Iterable can be set, dictionary, tuple, etc.

Syntax:

```
<set_name>=frozenset(iterable)
```

Example:

Program

```
days=frozenset(["Mon","Tue","Wed","Thu","Fri"
,"Sat","Sun"])

print(days)
```

Output

```
frozenset({'Wed', 'Fri',
'Mon', 'Tue', 'Sun', 'Sat',
'Thu'})
```

Frozensets Functions:

Python provides many functions that are built-in that work with Frozenset.

Function	Description
`len(object)`	Returns the total number of items in a frozen set
`str(object)`	Returns the frozen set item as string.
`max(object)`	returns the largest item in an frozen set.
`min(object)`	returns the smallest item in an frozen set.
`sum(object)`	The sum() function adds the items of an iterable and returns the sum.
`sorted(iterable, reverse=False)`	The sorted() function returns a sorted list from the items in an iterable. iterable - A sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any other iterator.reverse (Optional) - If True, the sorted list is reversed (or sorted in descending order). Defaults to False if not provided.

Example:

Program

```
num= frozenset([12,5,35,1,4,14])

print(len(num))

print(str(num))

print(max(num))

print(min(num))

print(sum(num))

print(sorted(num))

print(sorted(num,reverse=True))
```

Output

```
6

frozenset({1, 35, 4, 5, 12, 14})

35

1

71

[1, 4, 5, 12, 14, 35]

[35, 14, 12, 5, 4, 1]
```

Frozenset method:

There are many frozenset methods, some of which we have already used above. Here is a list of all the methods that are available with frozenset objects.

Function	Description
set.copy()	The copy() method returns a shallow copy of the frozenset.
set.difference(setB)	The difference() method returns the set difference of two sets.
set.intersection(setB)	The intersection() method returns a new set with elements that are common to all frozensets.
set.symmetric_difference (setB)	The Python symmetric_difference() method returns the symmetric difference of two frozensets.
set.union(setB)	The Python set union() method returns a new set with distinct elements from all the frozensets.
set.isdisjoint(setB)	The isdisjoint() method returns True if two frozensets are disjoint frozensets. If not, it returns False
set.issubset(setB)	The issubset() method returns True if all elements of a frozensets are present in another frozensets (passed as an argument). If not, it returns False.

Example:

Program

```
numA=frozenset({23,12,45,56,35,31,21})
numB=frozenset({23,12,89,78,35,87})

print(numA.difference(numB))

print(numA.intersection(numB))

print(numA.symmetric_difference(numB))

print(numA.union(numB))

print(numA.isdisjoint(numB))

print(numA.issubset(numB))

numC=numA.copy()

print(numC)
```

Output

```
frozenset({56, 45, 21, 31})

frozenset({35, 12, 23})

frozenset({45, 78, 21, 87, 56, 89, 31})

frozenset({35, 12, 45, 78, 21, 87, 23, 56, 89, 31})

False

False

frozenset({35, 21, 23, 56, 12, 45, 31})
```

Dictionary

Dictionaries are Python' s implementation of a data structure that is more generally known as an associative array. A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.

You can define a dictionary by enclosing a comma-separated list of key-value pairs in curly braces ({}). A colon (:) separates each key from its associated value. Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.Dictionaries are optimized to retrieve values when the key is known.

Creating a dictionary is as simple as placing items inside curly braces {} separated by comma.An item has a key and the corresponding value expressed as a pair, key: value.While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

Features :

- It is mutable. i.e. its elements can be changed.
- It is Dynamic. i.e. elements can be add / removed at runtime.
- It is Ordered . i.e. Its elements can be accessed by using a key.
- It can be nested i.e.A dictionary can contain another collection.

Syntax:

```
<dictionary_name>={key1:value1,key2:value2,……,keyN:valueN}
```

- The key must be unique.
- Value is accessed by key.
- Value can be updated while key cannot be changed.

Example:

Program

```
data={'first_name':"Amit",'last_name':"Jain",
'age':24}
print(data)
```

Output

```
{'first_name': 'Amit',
'last_name': 'Jain', 'age':
24}
```

Accessing Dictionary Values:

Dictionary values can be accessed by using keys .A value is retrieved from a dictionary by specifying its corresponding key in square brackets ([ ]). While indexing is used with other container types to access values, dictionary uses keys.

Syntax:

```
<dictionary_name>[key]
```

Example:

Program

```
my_dict = {'name': 'John', 1: [2, 4, 3]}
print(my_dict["name"])

print(my_dict[1])
```

Output

```
John

[2, 4, 3]
```

Dictionary Constructor:

The dict( ) constructor creates a dictionary in Python. A keyword argument is an argument preceded by an identifier (eg. name=). Hence, the keyword argument of the form kwarg=value is passed to the dict( ) constructor to create dictionaries. The dict( ) doesn't return any value (returns None).

Syntax:

```
<dict_name>=dict(**kwarg)
```

Example:

Program

```
data=dict(first_name="Amit",last_name="Jain",
age=31)
print(data)
```

Output

```
{"Mon","Tue","Wed","Thu","Fri"
,"Sat","Sun"}
```

Dictionary Functions:

Python provides many functions that are built-in that work with Dictionary.

Function	Description
`len(object)`	Returns the total number of items in a dictionary
`str(object)`	Returns the dictionary item as string.
`max(object)`	returns the largest key in an dictionary.
`min(object)`	returns the smallest key in an dictionary.
`sorted(iterable, reverse=False)`	The sorted() function returns a sorted list from the items in an iterable. iterable - A sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any other iterator.reverse (Optional) - If True, the sorted list is reversed (or sorted in descending order). Defaults to False if not provided.

Example:

Program

```
data={2010:"Amit",3021:"Sumit",4215:"Gopal",2
531:"Harry"}

print(len(data))

print(max(data))

print(min(data))

print(sum(data))

print(sorted(data))

print(sorted(data,reverse=True))
```

Output

```
4

4215

2010

11777

[2010, 2531, 3021, 4215]

[4215, 3021, 2531, 2010]
```

Dictionary method:

Methods that are available with dictionary are tabulated below.

Function	Description
dict.get(key)	The get() method returns the value for the specified key if key is in dictionary.
dict.keys()	The keys() method returns a view object that displays a list of all the keys in the dictionary.
dicti.items()	The items() method returns a view object that displays a list of dictionary's (key, value) tuple pairs.
dict.popitem()	The popitem() returns and removes an arbitrary element (key, value) pair from the dictionary.
dict.setdefault(key, default_value)	The setdefault() method returns the value of a key (if the key is in dictionary). If not, it inserts key with a value to the dictionary.
dict.pop(key, default_value)	The pop() method removes and returns an element from a dictionary having the given key.
dict.values()	The values() method returns a view object that displays a list of all the values in the dictionary.
dict.update(dict)	The update() method updates the dictionary with the elements from the another dictionary object or from an iterable of key/value pairs.
dict.copy()	They copy() method returns a shallow copy of the dictionary.
dict.clear()	The clear() method removes all items from the dictionary.
dict.fromkeys(sequence, value)	The fromkeys() method creates a new dictionary from the given sequence of elements with a value provided by the user.

Example:

Program

```
data={2010:"Amit",3021:"Sumit",4215:"Gopal",2
531:"Harry"}
print(data.get(2010))

print(data.keys())

print(data.items())

print(data.popitem())
print(data)

print(data.setdefault(2531,"Harry"))
print(data)

print(data.pop(2532,None))

print(data.values())

data.update({4251:"Raj"})
print(data)

data2=data.copy()
print(data2)

data.clear()
print(data)
```

Output

```
Amit

dict_keys([2010, 3021, 4215,
2531])

dict_items([(2010, 'Amit'),
(3021, 'Sumit'), (4215,
'Gopal'), (2531, 'Harry')])

(2531, 'Harry')
{2010: 'Amit', 3021: 'Sumit',
4215: 'Gopal'}

Harry
{2010: 'Amit', 3021: 'Sumit',
4215: 'Gopal', 2531: 'Harry'}


None

dict_values(['Amit', 'Sumit',
'Gopal', 'Harry'])

{2010: 'Amit', 3021: 'Sumit',
4215: 'Gopal', 2531: 'Harry',
4251: 'Raj'}

{2010: 'Amit', 3021: 'Sumit',
4215: 'Gopal', 2531: 'Harry',
4251: 'Raj'}


{}
```

Processing Dictionary:

Items of dictionary can be processed by using for loop.If for loop is applied for dictionary object it return key of each item.

Example:

Program

```
dit = {'Name': 'Amit Jain', 'Age':
7,'City':'Amravati'};

for key in dit:
    print(key);
```

Output

```
Name
Age
City
```

Example:

Program

```
dit = {'Name': 'Amit Jain', 'Age':
7,'City':'Amravati'};

for key,val in dic.items():
    print(key, " : " ,val);
```

Output

```
Name  : Amit Jain
Age : 7
City : Amravati
```

Map Function

The map() function applies a given function to each item of an iterable (list, tuple etc.) and returns a list of the results. It applies a given function to each item of an iterable (i.e Collection for ex: list, tuple etc.).The map() function applies a given to function to each item of an iterable and returns a list of the results. The returned value from map() (map object) then can be passed to functions like list() (to create a list), set() (to create a set) and so on.

Note: You can pass more than one iterable to the map() function.

Syntax:

```
map(function, iterable, ...)
```

Parameter:

- function - map() passes each item of the iterable to this function.
- Iterable - iterable which is to be mapped.

Example:

Program

```
def myfunc(nm):
  return nm.upper()

names=['amit','raj', 'mohan' ]
itr = map( myfunc, names )
lst=list(itr)
print(lst)
```

Output

```
['AMIT', 'RAJ', 'MOHAN']
```

Example:

Program

```
def myfunc(a, b):
  return a + " "+ b

itr = map(myfunc, ('amit', 'gopal', 'raj‘
,’mona’), ('jain', 'pandey', 'joshi'))
lst=list(itr)
print(lst)
```

Output

```
['amit jain', 'gopal pandey',
'raj joshi']
```

Example:

Program

```
print("Enter nos.")
nos=input().split()
print(nos)

itr=map(int,nos)
lst=list(itr)
print(lst)
```

Output

```
Enter nos.
5 10 20 10 5 2
['5', '10', '20', '5', '2']
[5, 10, 20, 10, 5, 2]
```

Example:

Program

```
fnames=('amit', 'gopal', 'raj')
lnames=('joshi', 'patil', 'pandey')

itr = map(lambda fn,ln : fn+" "+ln , fnames ,
lnames)
lst=list(itr)
print(lst)
```

Output

```
['amit joshi', 'gopal patil',
'raj pandey']
```

Array

This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character.

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

Syntax:

```
array(typecode, initializer)
```

A new array whose items are restricted by typecode, and initialized from the optional initializer value, which must be a list, a bytes-like object, or iterable over elements of the appropriate type.

Type code	C Type	Python Type	Minimum size in bytes
'b'	signed char	int	1
'B'	unsigned char	int	1
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Array objects support the ordinary sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, TypeError is raised.

Accessing Array Values:

Accessing elements is the same as accessing Strings in Python. You pass the index values and hence can obtain the values as needed.

Syntax:

```
<array_name>[index]
```

Example:

Program

```
import array
arr=array.array('i',[1,2,3,4,5,6,7])
print(arr[5])
print(arr[2:7])
print(arr[:3])
```

Output

```
6
array('i', [3, 4, 5, 6, 7])
array('i', [1, 2, 3])
```

Array method:

Methods that are available with array are tabulated below.

Function	Description
array.append(x)	Append a new item with value x to the end of the array.
array.count(x)	Return the number of occurrences of x in the array.
array.extend(iterable)	Append items from iterable to the end of the array. If iterable is another array, it must have exactly the same type code; if not, TypeError will be raised.
array.index(x)	Return the smallest i such that i is the index of the first occurrence of x in the array.
array.insert(i, x)	Insert a new item with value x in the array before position i. Negative values are treated as being relative to the end of the array.
array.pop([i])	Removes the item with the index i from the array and returns it.
array.remove(x)	Remove the first occurrence of x from the array.
array.reverse()	Reverse the order of the items in the array.

array.tolist()	Convert the array to an ordinary list with the same items.

Example:

Program

```
import array


arr=array.array('i',[1,2,3,4,5,6])
print(arr)
arr.append(3)
print(arr)
print(arr.count(3))
arr.extend([2,5])
print(arr)
print(arr.index(2))
arr.insert(2,5)
print(arr)
print(arr.pop())
arr.remove(3)
print(arr)
arr.reverse()
print(arr)
print(arr.tolist())
```

Output

```
array('i', [1, 2, 3, 4, 5, 6])
array('i', [1, 2, 3, 4, 5, 6, 3])
2
array('i', [1, 2, 3, 4, 5, 6, 3, 2, 5])


1
array('i', [1, 2, 5, 3, 4, 5, 6, 3, 2,
5])
5
array('i', [1, 2, 5, 4, 5, 6, 3, 2])
array('i', [2, 3, 6, 5, 4, 5, 2, 1])
[2, 3, 6, 5, 4, 5, 2, 1]
```

Modules

Modules

A module is simply a Python file, where classes, functions and variables are defined. These modules can be imported in our project whenever required. Modules refer to a file containing Python statements and definitions.

A file containing Python code, for e.g.: example.py, is called a module and its module name would be example. We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code. We can define our most used functions in a module and import it, instead of copying their definitions into different programs. Let us create a module. Type the following and save it as example.py.

There are 2 types of modules:

- Built-in Modules
- User-define Modules

Built-in Modules

Some popular modules in Python are.

1. math : for mathematical functions
2. threading : for multithreaded applications.
3. collections : for additional collections classes.
4. os : operating system related functions
5. re : for text processing
6. random : for random number generation
7. pickle : is used for serializing and de-serializing Python objects.
8. nltk : natural language processing.
9. datetime : is used to perform operations on date and time data.
10. tkinter : for GUI interface..
11. sockets : for network programming.
12. Etc

Import Statement

It is used to import a module. We can import the definitions inside a module to another module or the interactive interpreter in Python. We use the import keyword to do this. We can import a module using import statement and access the definitions inside it using the dot operator.

Syntax:

```
import <module_name>
```

Example:

Program

```
import math
print("Pi value: ",math.pi)
```

Output

```
Pi value 3.141592653589793
```

from import Statement

It is used to import particular attribute from a module. We can import specific names from a module without importing the module as a whole. We can import all names (definitions) from a module

Syntax:

```
from <modulename> import <attributename>,..
from <modulename> import *
```

Example:

Program

```
from math import pi
print("Pi value: ",pi)
```

Output

```
Pi value 3.141592653589793
```

Renaming a module:

We can create an alias when we import a module. This makes it possible to place names directly into the local symbol table but avoid conflicts with previously existing names

Syntax:

```
import <modulename> as <aliasname>

from <module_name> import <name> as <alt_name>
```

Example:

Program

```
import math as m
print("Pi value: ",m.pi)
```

Output

```
Pi value 3.141592653589793
```

User-defined Modules

It is module that is created by the user so that module can be use by himself or by other users in other programs.

Step to create a module

- Create a file having same name as modulename.
- In that file we can define any type of components such as functions , varialbes, classes , objects etc.

Example:

Program

```
mylib.py
-----------------------------------------
class Rectangle:

      def setdimension(self,x,y):
            self.length=x
            self.breadth=y

      def area(self):
            a=self.length*self.breadth
            print("Area is ",a)

class Circle:

      def setradius(this,n):
            this.r=n

      def area(this):
            a=3.14*this.r**2
            print("Area is ",a)

def display(n):
      for i in range(1,n+1):
            print(i)

demo.py
-----------------------------------------
import mylib

a=mylib.Circle()

b=mylib.Rectangle()

a.setradius(5)

b.setdimension(5,10)

a.area()

b.area()

mylib.display(5)
```

Output

```
Area is  78.5

Area is  50

1

2

3

4

5
```

Note : If import statement is used to import an module then components of that module can be accessed by using syntax: **moduleName.ComponentName**

Example:

Program

```
mylib.py
-----------------------------------------

class Rectangle:

      def setdimension(self,x,y):
            self.length=x
            self.breadth=y

      def area(self):
            a=self.length*self.breadth
            print("Area is ",a)

class Circle:

      def setradius(this,n):
            this.r=n

      def area(this):
            a=3.14*this.r**2
            print("Area is ",a)

def display(n):
      for i in range(1,n+1):
            print(i)

demo.py
-----------------------------------------

from mylib import Circle


a=Circle()

a.setradius(5)

a.area()
```

Output

```
Area is  78.5
```

Note : If from import statement is used to import an Component of a module then components of that module can be accessed by using **syntax: ComponentName**

We can use * to import all components from a module.

Syntax: from <module> import *

Example:

Program

```
mylib.py
-----------------------------------------

class Rectangle:

      def setdimension(self,x,y):
            self.length=x
            self.breadth=y

      def area(self):
            a=self.length*self.breadth
            print("Area is ",a)

class Circle:

      def setradius(this,n):
            this.r=n

      def area(this):
            a=3.14*this.r**2
            print("Area is ",a)

def display(n):
      for i in range(1,n+1):
            print(i)

demo.py
-----------------------------------------

import mylib as lib


a=lib.Circle()

a.setradius(5)

a.area()
```

Output

```
Area is  78.5
```

Note : If modules or components have lengthy names then we can use some short names for them

syntax:

AliasModuleName.ComponentName

AliasCompName

Python Module Search Path

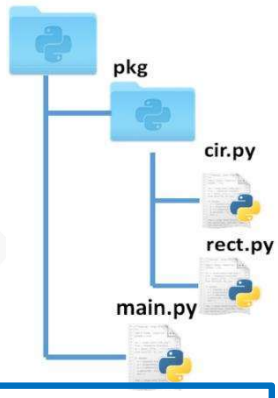
While importing a module, Python looks at several places. Interpreter first looks for a built-in module then (if not found) into a list of directories defined in sys.path. The search is in this order.

- The current directory.
- PYTHONPATH (an environment variable with a list of directory).
- The installation-dependent default directory.

Python Packages

A package is a collection of Python modules. Packages allow for a hierarchical structuring of the module namespace using dot notation. In the same way that modules help avoid collisions between global variable names, packages help avoid collisions between module names.

Creating a package is quite straightforward, since it makes use of the operating system' s inherent hierarchical file structure.



Example:

Program

```
cir.py
-------------------------------------
class Circle:

    def setradius(this,n):
        this.r=n

    def area(this):
        a=3.14*this.r**2
        print("Area is ",a)
rect.py
-------------------------------------
class Rectangle:

    def setdimension(self,x,y):
        self.length=x
        self.breadth=y

    def area(self):
        a=self.length*self.breadth
        print("Area is ",a)
```

```
main.py
-------------------------------------
from pkg import Circle,Rectangle

a=cir.Circle()
a.setradius(5)
a.area()

b=rect.Rectangle()
b.setdimension(5,2)
b.area()
```

Output

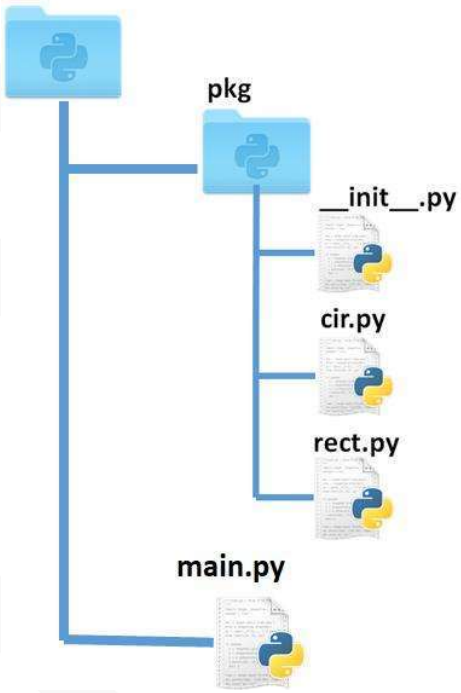
```
Area is  78.5

Area is  10
```

Packages __init__.py

A package is a directory of Python modules containing an additional __init__.py file. The __init__.py file indicate that it is a python package directory. If a file named __init__.py is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.

Example:



Program

```
cir.py
-------------------------------------
class Circle:

    def setradius(this,n):
        this.r=n

    def area(this):
        a=3.14*this.r**2
        print("Area is ",a)

rect.py
-------------------------------------
class Rectangle:

    def setdimension(self,x,y):
        self.length=x
        self.breadth=y

    def area(self):
        a=self.length*self.breadth
        print("Area is ",a)

__init__.py
-------------------------------------
import pkg.Circle,pkg.Rectangle

file=["Circle","Rectangle"]
```

```
main.py
-----------------------------------
import pkg

print(pkg.file)

a=pkg.cir.circle()
a.setradius(5)
a.area()

b=pkg.rect.Rectangle()
b.setdimension(5,2)
b.area()
```

Output

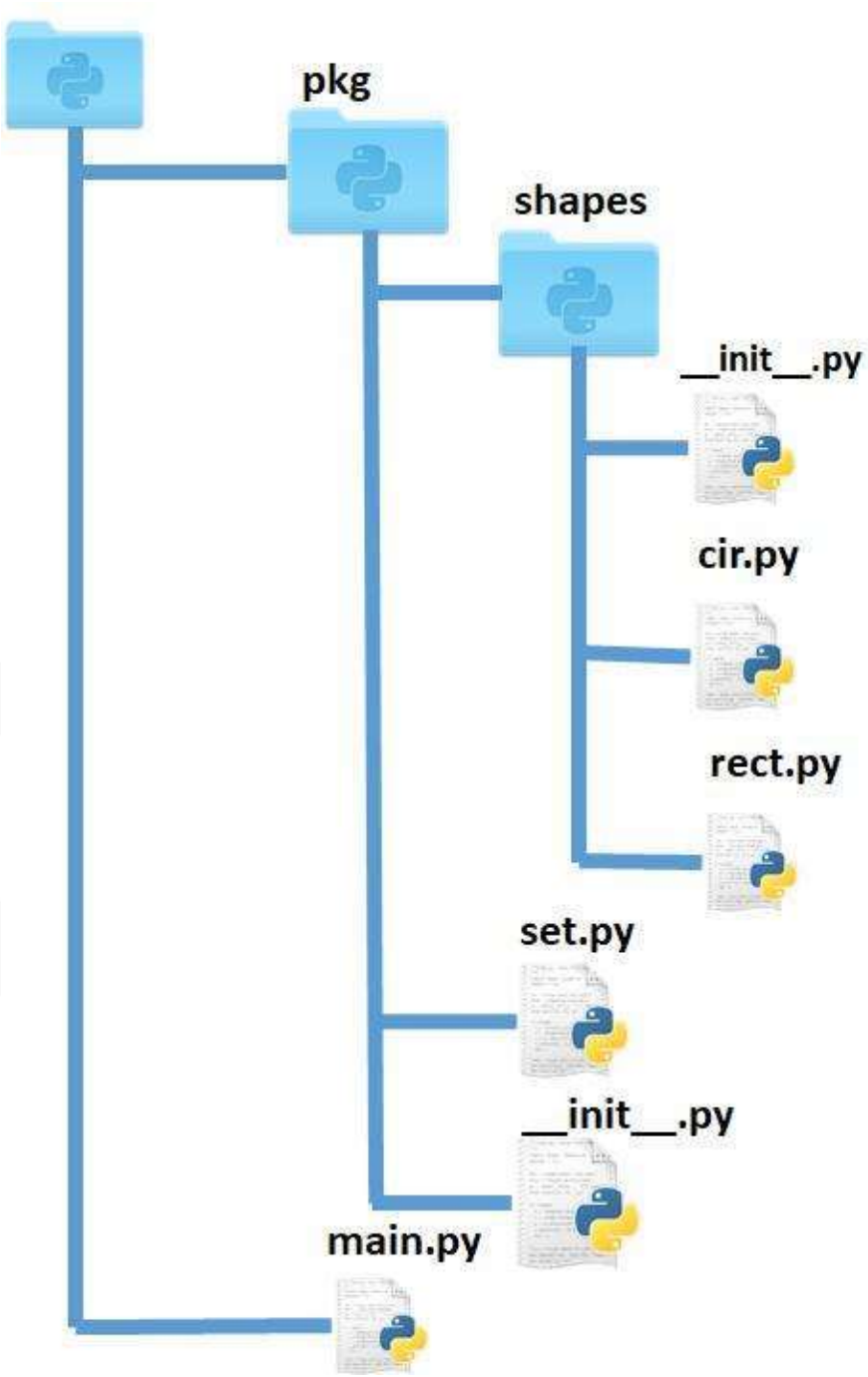
```
["Circle","Rectangle"]
Area is  78.5
Area is  10
```

Packages Subpackages

Packages can contain nested subpackages to arbitrary depth. when you import a package, only variables / functions/ classes of that package are directly visible, not sub-packages .

Example:

Program



Program

```
pkg/shapes/cir.py
-----------------------------------
class Circle:

    def setradius(this,n):
        this.r=n

    def area(this):
        a=3.14*this.r**2
        print("Area is ",a)

pkg/shapes/rect.py
-----------------------------------
class Rectangle:

    def setdimension(self,x,y):
        self.length=x
        self.breadth=y

    def area(self):
        a=self.length*self.breadth
        print("Area is ",a)

pkg/shapes/__init__.py
-----------------------------------
import pkg.shapes.cir
import pkg.shapes.rect

pkg/__init__.py
-----------------------------------
import pkg.set
```

```
pkg/set.py
-----------------------------------
class Set:

      def __init__(self,a,b,c):
            self.N1=a
            self.N2=b
            self.N3=c

      def sum(self):
            s=self.N1+self.N2+self.N3
            print("Sum is ",s)

main.py
-----------------------------------
from pkg import *
from pkg.shapes import *

b=set.Set(2,3,5)
b.sum()

c=cir.Circle(5)
c.area()

a=rect.Rectangle(5,2)
a.area()
```

Output

```
Sum is  10
Area is  78.5
Area is  10
```