Unit-III

Lists :

A list can be defined as a collection of values or items. The items in the list are separated with the comma (,) and enclosed with the square brackets. **Lists might contain items of different types.**

Ex 1:

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Ex2:

```
>>> emp = ["pavan", 35, "India"]
>>> emp
['pavan', 35, 'India']
```

Accessing Values in Lists:

To access the values in the lists, we can use indexing and slicing operations using index([ ]) operator and slicing [:] operator. list indices starts with 0. python provides us the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (right most) of the list has the index -1 , its adjacent left element is present at the index -2 and so on until the left most element is encountered.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[0]
1
>>> squares[-3]
9
```

Note 1:

Index values must be with in the range. Trying to access the list of elements out of the range raise IndexError. Also index value must be integer value. We cannot use float or other type of value. Using other type of value(except integer)for index raise TypeError.

Ex:

```
>>> squares = [1, 4, 9, 16, 25]

>>> squares
[1, 4, 9, 16, 25]

>>> squares[25]       # index value out of range
Traceback (most recent call last):
  File "<pyshell#47>", line 1, in <module>
    squares[25]
IndexError: list index out of range

>>> squares[1.2]      #index value must be integer
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    squares[1.2]
TypeError: list indices must be integers or slices, not float
```

Note2:

All slice operations return a new copy of the list containing the requested elements.

Ex1:

```
>>> squares[0:2]
[1, 4]
```

Ex2:

```
>>> a=squares[2:4]
>>> a
[9, 16]
```

lists are mutable type, i.e. it is possible to change their content. You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator.

Ex:

```
>>>squares = [1, 4, 9, 16, 25]
>>> squares
```

[1, 4, 9, 16, 25]

```
>>> squares[2:3]=[99,90]
>>> squares
[1, 4, 99, 90, 16, 25]
```

To remove a list element, you can use either the del statement if you know exactly which element(s).
Ex:

```
>>> squares
[1, 4, 9, 16, 25]
>>> del squares[3]
>>> squares
[1, 4, 9, 25]
```

List methods:

Method name	Description	example

append(x)	This method takes a single item x and adds it to the end of the list.
	The item can be numbers, strings, another list, dictionary etc.
		>>> list1=[1,2,3,4]
		>>> list1.append(5)
		>>> list1
		[1, 2, 3, 4, 5]

extend(*iterable*)	Extend the list by appending all the items from the iterable.	>>> list1=[1,2,3,4]
		>>> list1
		[1, 2, 3, 4]
		>>> t1=(5,6,7)
		>>> list1.extend(t1)
		>>> list1
		[1, 2, 3, 4, 5, 6, 7]
		>>> list1.extend("python")
		>>> list1
		[1, 2, 3, 4, 5, 6, 7, 'p', 'y', 't', 'h', 'o', 'n']

insert(*i*, *x*)	inserts the element x at the given index i, shifting elements to the right. The first argument is the index at which insert has to be made and second argument is the element to be inserted	>>> list1=[1,2,3,4,5]
		>>> list1.insert(2,100)
		>>> list1
		[1, 2, 100, 3, 4, 5]

remove(*x*)	Remove the first item from the list whose value is equal to *x*. It raises a ValueError if there is no such item.	>>> list1
		[1, 2, 100, 3, 4, 5]
		>>> list1.remove(100)
		>>> list1
		[1, 2, 3, 4, 5]

pop([*i*])	Remove the item at the given position in the list, and return it. If no index is specified, pop() removes and returns the last item in the list.	>>> list1
		[1, 2, 3, 4, 5]
		>>> list1.pop(3)
		4
		>>> list1
		[1, 2, 3, 5]
		>>> list1.pop()
		5
		>>> list1
		[1, 2, 3]

clear()	Remove all items from the list	>>> list1
		[1, 2, 3]
		>>> list1.clear()
		>>> list1
		[ ]

index(*x*[, *start* [, *end*]])	Return zero-based index in the list of the first item whose value is equal to *x*. Raises a ValueError if there is no such item.
	
	The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.
		>>> list1=[1,2,3,4,5]
		>>> list1.index(4)
		3
		>>> list1.index(4,2,5)
		3

count(*x*)	Return the number of times *x* appears in the list.	>>> vowels = ['a', 'e', 'i', 'o', 'i', 'u']
		>>> print('The count of i is:', vowels.count('i'))
		The count of i is: 2

sort(***key=None, reverse=False***)	The sort() method sorts the elements of a given list in a specific order - Ascending or Descending.
	
	It has two optional parameters:
	reverse - If true, the sorted list is reversed (or sorted in Descending order)
	key - function that serves as a key for the sort comparison
		>>> list1=[1,0,2,3,-1]
		>>> list1.sort()
		>>> list1
		[-1, 0, 1, 2, 3]
		>>>
		list1.sort(reverse=True)
		>>> list1
		[3, 2, 1, 0, -1]

reverse()	Reverse the elements of the list in place.	>>> list1=['a','e','i','o','u']
		>>> list1
		['a', 'e', 'i', 'o', 'u']
		>>> list1.reverse()
		>>> list1
		['u', 'o', 'i', 'e', 'a']

copy()	Return a shallow copy of the list.	>>> list1
		['a', 'e', 'i', 'o', 'u']
		>>> list2=list1.copy()
		>>> list2
		['a', 'e', 'i', 'o', 'u']

Basic List operations:

Consider the following two lists L1=[1,2,3,4] and L2=[5,6,7,8]

Operator	Description	Example
Repetition	The repetition operator * enables the list elements to be repeated multiple times.	L1*2 = [1, 2, 3, 4, 1, 2, 3, 4]
Concatenation	The concatenation operator + is used to combine two lists	L1+L2 = [1, 2, 3, 4, 5, 6, 7, 8]
Membership	We can test if an item exists in a list or not, using the keyword in. It returns True if a particular item exists in a particular list otherwise False.	print(2 in L1) prints True.
Iteration	The for loop is used to iterate over the list elements.	for i in L1: print(i) **Output** 1 2 3 4
Length	It is used to get the length of the list	len(L1) = 4

Mutator Methods and the Value None:

Mutable objects (such as lists) have some methods devoted entirely to modifying the internal state of the object. Such methods are called **mutators**.
Examples are the list methods insert, append, extend, pop, and sort.

Mutator methods are called to change the state of an object. These methods usually return the value None(except pop method). This value is automatically returned by any function or method that does not have a return statement.

Python automatically returns the special value None even when a method does not explicitly return a value

Ex1:

```
>>> list1=[20,2,1,-2,0]

>>> print(list1)
[20, 2, 1, -2, 0]

>>> list2=list1.sort()   #sort() method is mutator method. It returns None

>>> print(list1)
[-2, 0, 1, 2, 20]

>>> print(list2)
None
```

Ex2:

```
>>> list1=[20,2,1,-2,0]

>>> print(list1)
[20, 2, 1, -2, 0]

>>> list1=list1.sort()     #sort() method is mutator method. It returns None

>>> print(list1)           #observe carefully
None
```

Dictionaries:

A dictionary is the unordered collection of key-value pair of items where values can be of any data type and can repeat but keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

Creating a dictionary:

For each item, each key is separated from its value by a colon (:) and the items are separated by commas enclosed in curly braces{ }.

Ex:1

```
>>>phonebook={'pavan':8374518088,'kumar':9000009999}

>>>print(phonebook)

{'kumar': 9000009999, 'pavan': 8374518088}
```

Ex2:

```
>>>Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGL
E"}

>>>print(Employee)

{'salary': 25000, 'Company': 'GOOGLE', 'Age': 29, 'Name': 'John'}
```

Ex3:

```
>>>d={ }     #creates an empty dictionary

>>>print(d)

{ }
```

The dict() constructor builds dictionaries directly from sequences of key-value pairs

Ex4:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])

{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Ex5:

```
>>>dict( )        #creates an empty dictionary
```

Note:

You cannot have duplicate keys in a dictionary. When you assign a value to an existing key, the new value replaces the existing value.

Accessing elements of a dictionary:

Dictionaries are indexed by keys. So in the dictionary, the values of the elements can be accessed by using the keys by writing the keys inside square brackets [ ].If the key is found, the corresponding value is retrieved else, If the key is not found, this procedure returns a KeyError.

Ex:1

```
>>>phonebook={'pavan':8374518088,'kumar':9000009999}
>>> print(phonebook['kumar'])
9000009999
```

Ex2:

```
>>>d = {'name':'Jack', 'age': 26}
>>> print(d['name'])
 Jack
```

Note:

We can also use get( ) method to access the values of the dictionary using key

Updating dictionary elements:

Dictionary is mutable. We can add new items or change the value of existing items using assignment operator.

If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

Ex:

```
>>>d = {'name':'Jack', 'age': 26}
>>>print(d)
{'name': 'Jack', 'age': 26}
>>>d['age'] = 27                          # update value of the age
>>>print(d)
{'name': 'Jack', 'age': 27}

>>>d['phoneno']=8374518088          #updating with new key :value pair
>>>print(d)
{'name': 'Jack', 'age': 27, 'phoneno': 8374518088}
```

Note:

The update( ) method also adds element(s) to the dictionary if the key is not in the dictionary. If the key is in the dictionary, it updates the key with the new value.
The update( ) method takes either a dictionary or an iterable object of key/value pairs (generally tuples).

Ex1:

```
>>>d = {1: "one", 2: "two"}
>>>print(d)
{1: 'one', 2: 'two'}
>>>d1 = {3: "three"}
>>>d.update(d1)                 # adds element with key 3
>>>print(d)
{1: 'one', 2: 'two', 3: 'three'}
```

Ex2:

```
>>>d = {'x': 2}
>>>print(d)
{'x': 2}
>>>d.update(y = 3, z = 0)
>>>print(d)
{'x': 2, 'y': 3, 'z': 0}
```

deleting or removing elements from a dictionary:

We can remove a particular item in a dictionary by using the method pop(). This method removes as item with the provided key and returns the value.

The method, popitem() can be used to remove and return an arbitrary item (key, value) form the dictionary. All the items can be removed at once using the clear() method.

We can also use the del keyword to remove individual items or the entire dictionary itself.

Ex:

```
>>>squares = {1:1, 2:4, 3:9, 4:16, 5:25}      # create a dictionary
>>> print(squares.pop(4))                      # remove a particular item
16
>>> print(squares)
{1: 1, 2: 4, 3: 9, 5: 25}
>>> print(squares.popitem())                   # remove an arbitrary item
 (1, 1)
>>> print(squares)
{2: 4, 3: 9, 5: 25}
>>> del squares[5]                             # delete a particular item
>>>print(squares)
{2: 4, 3: 9}
>>> squares.clear()                           # remove all items
>>> print(squares)
{}
>>> del squares                               # delete the dictionary itself
>>>print(squares)                             # Throws Error
```

Dictionary methods:

1.**copy( ) method:**

This method returns a shallow copy of the dictionary. It doesn't modify the original dictionary. When copy( ) method is used, a new dictionary is created which is filled with a copy of the references from the original dictionary.

Ex:

```
>>>original = {1:'one', 2:'two'}
>>>new = original.copy()
```

```
>>>print('Orignal: ', original)
Orignal:  {1: 'one', 2: 'two'}
>>>print('New: ', new)
New:  {1: 'one', 2: 'two'}
```

Note:
When = operator is used to copy the dictionary, a new reference to the original dictionary is created.

Using copy() method	Using = operator to copy the dictionary

original = {1:'one', 2:'two'}
new = original.copy()
 	
# removing all elements from the list 	
new.clear()
 	
print('new: ', new)
print('original: ', original)
	original = {1:'one', 2:'two'}
	new = original
	
	# removing all elements from the list
	new.clear()
	
	print('new: ', new)
	print('original: ', original)

Output: new: {} original: {1: 'one', 2: 'two'}	Output: new: {} original: {}

2. **get(key[, default]) method:**

The get( ) method returns the value for the specified key if key is in dictionary. If the key is not found and default is specified this method returns d. If the key is not found and default is not specified this method returns None.

Ex:

```
>>>person = {'name': 'Pavan', 'age': 22}
>>>print('Name: ', person.get('name'))
Name: Pavan

>>>print('Age: ', person.get('age'))
```

Age: 22

>>>print('Salary: ', person.get('salary')) # Key is not found default value is not provided
Salary: None

>>>print('Salary: ', person.get('salary', 0.0)) # key is not found and default value is provided
Salary: 0.0

3. items( ) method:

The items() method returns a view object that displays a list of dictionary's (key, value) tuple pairs.

Ex:

```
>>>person = {'name': 'Pavan', 'age': 22}
>>>print(person.items())
dict_items([('name', 'Pavan'), ('age', 22)])
```

Note:

The view objects provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

```
>>>person = {'name': 'Pavan', 'age': 22}
>>>d=person.items()
>>>print('Original items:', d)
Original items: dict_items([('name', 'Pavan'), ('age', 22)])
>>>del person['age']
>>>print('Updated items:', d)
Updated items: dict_items([('name', 'Pavan')])
```

4. keys( ) method:

The keys( ) method returns a view object that displays a list of all the keys in the dictionary. When the dictionary is changed, the view object also reflect these changes.

Ex:

```
>>>person = {'name': 'Pavan', 'age': 22}
>>>print(person.keys())
```

dict_keys(['age', 'name'])

5. **values( ) method:**
The values( ) method returns a view object that displays a list of all the values in the dictionary. When the dictionary is changed, the view object also reflect these changes.

Ex:
```
>>>person = {'name': 'Pavan', 'age': 22}
>>>print(person.values())
dict_values(['Pavan', 22])
```

6. **fromkeys(*iterable*[, *value*]) method**:
Create a new dictionary with keys from *iterable* and values set to *value.value* defaults to None(i,e., If the value argument is not given, value set to None)

Ex:
```
>>>keys = {'name','age'}
>>>d = dict.fromkeys(keys)
>>>print(d)
{'name': None, 'age': None}
```

Ex:
```
>>>keys = {'name','age'}
>>>values=('pavan',22)
>>>d = dict.fromkeys(keys,values)
>>>print(d)
{'name': ('pavan', 22), 'age': ('pavan', 22)}
```

7**. setdefault(key,[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.

The setdefault() takes maximum of two parameters:

- **key** - key to be searched in the dictionary
- value (optional) - *key* with a value is inserted to the dictionary if key is not in the dictionary. If not provided, it defaults to None.

Ex:

```
>>>person = {'name': 'Pavan'}
>>>print(person)
{'name': 'Pavan'}
>>>person.setdefault('salary')    # key is not in the dictionary and value not
provided
>>>print(person)
{'name': 'Pavan', 'salary': None}
>>>person.setdefault('age', 22)     # key is not in the dictionary and value  provided
>>>print(person)
{'age': 22, 'name': 'Pavan', 'salary': None}
```

Built-in Functions with Dictionary:

Built-in functions like all(), any(), len(), sorted() etc. are commonly used with dictionary to perform different tasks.

Function	Description
all()	Return True if all keys of the dictionary are true (or if the dictionary is empty).
any()	Return True if any key of the dictionary is true. If the dictionary is empty, return False.
len()	Return the length (the number of items) in the dictionary.
sorted()	Return a new sorted list of keys in the dictionary.

Comprehensions:

Comprehensions in Python provide a short and concise way to construct new sequences using sequences which have been already defined. Python supports the following types of comprehensions:

- List Comprehension
- Dictionary Comprehension
- Set Comprehension

List Comprehension:

List comprehension is an elegant and concise way to create new list from an existing list in Python.

List comprehension consists of an expression followed by for statement followed by additional for or if statements inside square brackets.

```
list_variable = [x for x in iterable]
```

Ex1:
Consider the following python code containing list comprehension

```
pow2 = [2 ** x for x in range(10)]
print(pow2)
```

the output of the above python code is as follows

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

The above list comprehension makes a list with each item being increasing power of 2.

The above code is equivalent to the following code:

```
pow2 = [ ]
for x in range(10):
   pow2.append(2 ** x)
print pow2
```

Ex2:

```
shark_letters = [letter for letter in 'shark']
print(shark_letters)
```

output:

['s', 'h', 'a', 'r', 'k']

The above code is equivalent to the following code.

```
shark_letters = []

for letter in 'shark':
    shark_letters.append(letter)

print(shark_letters)
```

A list comprehension can optionally contain more for or if statements. An optional if statement can filter out items for the new list

Ex1:

```
>>> pow2 = [2 ** x for x in range(10) if x > 5]
>>> pow2
[64, 128, 256, 512]
```

Ex2:

```
>>> odd = [x for x in range(20) if x % 2 == 1]
>>> odd
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Ex3:

```
fish_tuple = ('blowfish', 'clownfish', 'catfish', 'octopus')
fish_list = [fish for fish in fish_tuple if fish != 'octopus']
print(fish_list)
```

output:

['blowfish', 'clownfish', 'catfish']

we can also replicate nested if statements with a list comprehension:

Ex:

```
number_list = [x for x in range(100) if x % 3 == 0 if x % 5 == 0]
print(number_list)
```

Here, the list comprehension will first check to see if the number x is divisible by 3, and then check to see if x is divisible by 5. If x satisfies both requirements it will print, and the output is:

[0, 15, 30, 45, 60, 75, 90]

We can also perform **nested iteration** inside a list comprehension.

```
matrix = [[1, 2], [3,4], [5,6], [7,8]]
transpose = [[row[i] for row in matrix] for i in range(2)]
print (transpose)
```

When we run the above code, the output will be:
[[1, 3, 5, 7], [2, 4, 6, 8]]

Dictionary comprehensions:

Dictionary comprehensions can be used to create dictionaries from arbitrary key and value expressions.

Dictionary comprehension consists of an expression pair (key: value) followed by for statement inside curly braces {}.

Ex1:
Consider the following code containing dictionary comprehension

```
squares = {x: x*x for x in range(6)}
print(squares)
```

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

The above code is equivalent to the following code

```
squares = {}
for x in range(6):
        squares[x] = x*x
```

A dictionary comprehension can optionally contain more for or if statements.
An optional if statement can filter out items to form the new dictionary.

```
odd_squares = {x: x*x for x in range(11) if x%2 == 1}
print(odd_squares)
```

Output:

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

Set comprehensions:

Ex1:

```
>>>s1={x*x for x in [1,2,3]}
>>>print(s1)
{1,4,9}
```

Ex2:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

Sequences:

A sequence is a positionally ordered collection of other objects. Sequences maintain a left to right order among the items they contain. The items of a sequence are stored and fetched by their relative positions.

The basic sequence types are

1.strings

2. lists

3. tuples

4. range objects

Common Sequence Operations:

Operation	Result
x in s	True if an item of *s* is equal to *x*, else False
x not in s	False if an item of *s* is equal to *x*, else True
s + t	the concatenation of *s* and *t*
s * n or n * s	equivalent to adding *s* to itself *n* times
s[i]	*i*th item of *s*, origin 0
s[i:j]	slice of *s* from *i* to *j*
s[i:j:k]	slice of *s* from *i* to *j* with step *k*
len(s)	length of *s*
min(s)	smallest item of *s*
max(s)	largest item of *s*
s.index(x[, i[, j]])	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)
s.count(x)	total number of occurrences of *x* in *s*

Defining Simple Functions:

A function is block of re-usable code to perform specific task.

Types of functions:

In python, functions are classified into two types. They are

a) Built-in functions
b) User defined functions

a) Built-in functions:

The Python interpreter has a number of functions built into it that are always available. These functions are called Built-in-functions.The built-in-functions available in python3 are given below.

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

b) user defined functions:

The functions that are defined by the users are called as user-defined functions.

Defining a function:

The syntax of a function definition is given below.

```
def functionname( parameter list ):
        """docstring"""
        statement1
        statement2
        statement3
        ------------
        return [expression]
```

A function definition starts with the keyword def followed by the function name and the parenthesized list of formal parameters ending with a colon(:)

The statements that form the body of the function start at the next line, and must be indented. The first statement of the function body can optionally be a string literal which is the function's documentation string, or *docstring* that is used to describe what function does. A function can also have an optional return statement that returns the value evaluated by the expression which defaults to None.

Ex:

```
def greet(name):
        """This function greets to
        the person passed in as
        parameter"""
        print("Hello, " + name + ". Good morning!")
```

calling a 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 parameters.

Ex1:

```
def greet(name):                    #function definition
        """This function greets to
```

```
        the person passed in as
        parameter"""
        print("Hello, " + name + ". welcome to python programming")


greet("pavan")                  #function call
```

output:

Hello, pavan. welcome to python programming

Ex2:

```
def add(x, y): # function definition
    z = x + y
    return z

result = add(3, 4) # function call
print (result)
```

output:
7

Defining a main Function:

In scripts that include the definitions of several cooperating functions, it is often useful to define a special function named main that serves as the entry point for the script. This function usually expects no arguments and returns no value. Its purpose might be to take inputs, process them by calling other functions, and print the results.

Ex:

```
"""
Filename: computesquare.py
Illustrates the definition of a main function.
"""

def main():
    number = float(input("Enter a number: "))
```

```
    result = square(number)
    print("The square of", number, "is", result)


def square(x):
 """Returns the square of x."""
   return x * x

# The entry point for program execution
if __name__ == "__main__":
   main()
```

```
Output:

Enter a number: 5
The square of 5.0 is 25.0
```

The above script can be run from IDLE, run from a terminal command prompt, or imported as a module.

When the **script is launched from IDLE or a terminal prompt**, the value of the module variable __name __ will be "__main__". In that case, the main function is called and the script runs as a standalone program.

When the **script is imported as a module**, the value of the module variable __name__ will be the name of the module, "computeSquare". In that case, the main function is not called, but the script's functions become available to be called by other code.

Function arguments:

A function can be called by using the following types of arguments −

- Keyword arguments
- Positional arguments
- Default arguments
- Variable-length arguments(arbitrary arguments)

Keyword arguments:

Keyword arguments allows you to pass each arguments using name value pairs like

argumentname=value.

While calling the function, the values passed through keyword arguments are assigned to parameters by their name irrespective of its position. All the keyword arguments must match one of the arguments accepted by the function.

Ex:

```
def named_args(name,greeting):
        print(greeting+" " +name)

named_args(name="pavan",greeting="welcome to python programming")
named_args(greeting='welcome to python programming',name='kumar')
```

output:

```
welcome to python programming pavan
welcome to python programming kumar
```

positional arguments(or required arguments):

Positional arguments are mandatory arguments. The values passed through positional arguments are assigned to parameters in order, by their position. These arguments need to be passed during the function call and in precisely the right order, we cannot skip passing them.

Ex:

```
def add(x, y, z):
   s = x + y + z
   return s

result=add(3, 4, 5) # positional arguments
print(result)
```

Output:

```
12
```

Note1:

Having a positional argument after keyword arguments will result into errors.
(first positional arguments then keyword arguments must come)

Ex:

```
def add(x, y, z):
    s = x + y + z
    return s

print(add(2, z=4, y=5))     # z and y are keyword arguments and the result is 11
print(add(2, z=5, 10))      # results in SyntaxError since positional argument
                            # follows keyword argument
print(add(2, 5, y=10)       #TypeError: add() got multiple values for argument 'y'
```

Default arguments:

Default arguments are those that take a default value if no argument value is passed during the function call. To specify default values of argument, you just need to assign a value using assignment operator in function definition.

Ex:

```
def add(x, y=0, z=0):          # function definition with default arguments
    s = x + y + z
    return s

print(add(4))          # y, z takes default value 0

print(add(4, 5))       # y's default value replaced by the actual argument value 5
                       #  and z takes default value 0

print(add(4, z=5))   # y takes default value 0 and z's default value replaced by the
                       #actual argument, 5
```

Note2:

non-default arguments must not follow default arguments
(first nondefault arguments must come and followed by default arguments)

Ex:

```
def add(x=0, y, z=0):
    r = x + y + z
    return r

print(add(4, 5))   #results in SyntaxError since non-default argument follows
                   #default argument
```

variable-length arguments:

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 arbitrary number of arguments.

To denote this kind of argument, use an asterisk (*) before the parameter name in the function definition. The asterisk (*) before the parameter name allows to pass variable number of non keyword arguments to a function. The arguments are passed as a tuple and these passed arguments make tuple inside the function with same name as the parameter excluding asterisk *

Ex:

```
def greet(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
    for name in names:
        print("Hello",name)

greet("pavan","kumar","phani","bharat")
```

output:

```
Hello pavan
Hello kumar
Hello phani
Hello bharat
```

Here, we have called the function with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a for loop to retrieve all the arguments back.

Python also allows to pass variable length of keyword arguments to the function by using double asterisk(**) before the parameter name in the function definition. The arguments are passed as a dictionary and these arguments make a dictionary inside function with name same as the parameter excluding double asterisk **.
Ex:

```
def intro(**data):
    for key, value in data.items():
        print("{} is {}".format(key,value))

print("output1:")
intro(Firstname="pavan", Lastname="kumar", Age=22, Phone=1234567890)
print("output2:")
intro(Firstname="phani", Lastname="kanth", Email="phani@gmail.com",
                        Country="India", Age=25, Phone=9876543210)
```

```
output1:
Firstname is pavan
Lastname is kumar
Age is 22
Phone is 1234567890
output2:
Firstname is phani
Lastname is kanth
Email is phani@gmail.com
Country is India
Age is 25
Phone is 9876543210
```

Design with Functions:

Functions as Abstraction Mechanisms:

To design software systems, the functions are very much useful. Effective designers must invent useful abstractions to control complexity. An abstraction hides detail and thus allows a person to view many things as just one thing.

A function serves as an abstraction mechanism by allowing us to view many things as one thing. The various ways in which functions serve as abstraction mechanisms in a program are given below.

a) Functions Eliminate Redundancy:

The first way that functions serve as abstraction mechanisms is by eliminating redundant, or repetitious, code. A function eliminates redundant patterns of code by specifying a single place where the pattern is defined.

Instead of multiple instances of redundant code, the programmer can write only a single function in just one place—say, in a library module. Any other module or program can then import the function for its use. Once imported, the function can be called as many times as necessary. When the programmer needs to debug, repair, or improve the function, then it needs to edit and test only the single function definition. There is no need to edit the parts of the program that call the function.

Ex:

```
def summation(lower, upper):
  """Arguments: A lower bound and an upper bound
  Returns: the sum of the numbers from lower through
  upper
  """
  result = 0
  while lower <= upper:
      result += lower
      lower += 1
  return result

print(summation(1,4))
print(summation(50,100))
```

Output:

10
3825

In a program that must calculate multiple summations, the same code would appear multiple times. In other words, redundant code would be included in the program.

If the summation function didn't exist, the programmer would have to write the entire algorithm every time a summation is computed.

By using the single function definition, the programmer can eliminate redundant code. Any other module or program can then import the function for its use. Once imported, the function can be called as many times as necessary.

When the programmer needs to debug, repair, or improve the function, she needs to edit and test only the single function definition. There is no need to edit the parts of the program that call the function.

b) Functions Hide Complexity:

Another way that functions serve as abstraction mechanisms is by hiding complicated details. A function hides a complex chunk of code in a single named entity.

c) Functions Support General Methods with Systematic Variations:

A function allows a general method to be applied in varying situations. The variations are specified by the function's arguments.

d) Functions Support the Division of Labor:

Functions support the division of labor when a complex task is divided into simpler subtasks. Each of the subtask required by a system can be assigned to a function,

Design Strategies:

Problem Solving with Top-Down Design:

One popular design strategy for programs of any significant size and complexity is called top-down design.

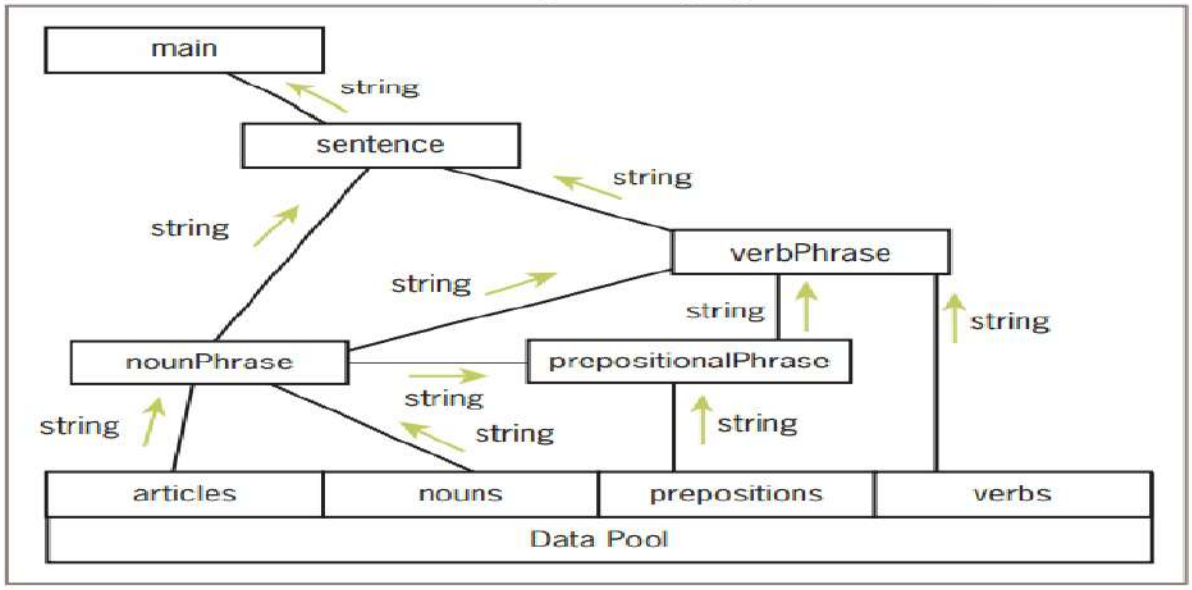
Top-down design is a strategy that decomposes a complex problem into simpler sub problems. This process is called **problem decomposition**. Functions are developed to solve each sub problem.

Problem decomposition may continue down to lower levels, because a sub problem might in turn contain two or more lower-level problems to solve. As functions are developed to solve each sub problem, the solution to the overall problem is gradually filled out in detail. This process is called **stepwise refinement.**

The relationships among the functions in this design are expressed in the structure chart.

A structure chart is a diagram that shows the relationships among a program's functions and the passage of data between them. Cooperating functions communicate information by passing arguments and receiving return values. They also can receive information directly from common pools of data.

Ex: A structure chart for the sentence-generator program



Each box in the structure chart is labeled with a function name. The main function at the top is where the design begins, and decomposition leads us to the lower-level functions on which main depends. The lines connecting the boxes are labeled with data type names, and arrows indicate the flow of data between them.

Design with Recursive Functions:

Recursive design is a special case of top-down design, in which a complex problem is decomposed into smaller problems of the same form. Thus, the original problem is solved by a single recursive function.

A recursive function is a function that calls itself. A recursive function consists of at least two parts:

- a base case that ends the recursive process and
- a recursive step that continues it.

These two parts are structured as alternative cases in a selection statement.

Ex:

```
def fib(n):
    """Returns the nth Fibonacci number."""
    if n == 1:
        return 0
    elif n==2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


n=int(input("enter the n value:"))
print(n,"th term of the sequence is", fib(n))
```

Output:

```
enter the n value:6
6 th term of the sequence is 5
```

Infinite Recursion:

A situation where the function can (theoretically) continue executing forever is known as infinite recursion. Infinite recursion arises when the programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process.

In fact, the Python virtual machine eventually runs out of memory resources to manage the process, so it halts execution with a message indicating a stack overflow error. **An infinite recursion eventually halts execution with an error message.**

```
def runForever(n):
    if n > 0:
        runForever(n)
    else:
        runForever(n - 1)

runForever(6)          #leads to infinite recursion which eventually halts with error
                       #message
```

Case Study: Gathering Information from a File System

Write a program that allows the user to obtain information about the file system. The program should display the path of the CWD, a menu of command options, and a prompt for a command, as shown in following Figure.

```
/Users/KenLaptop/Book/Chapter6
1   List the current directory
2   Move up
3   Move down
4   Number of files in the directory
5   Size of the directory in bytes
6   Search for a filename
7   Quit the program
Enter a number:
```

When the user enters a command number, the program runs the command, which may display further information, and the program displays the CWD and command menu again. An unrecognized command produces an error message, and command number 7 quits the program.
The following table summarizes what each command do.

Command	What It Does
List the current working directory	Prints the names of the files and directories in the current working directory (CWD).
Move up	If the CWD is not the root, move to the parent directory and make it the CWD.
Move down	Prompts the user for a directory name. If the name is not in the CWD, print an error message; otherwise, move to this directory and make it the CWD.
Number of files in the directory	Prints the number of files in the CWD and all of its subdirectories.
Size of the directory in bytes	Prints the total number of bytes used by the files in the CWD and all of its subdirectories.
Search for a filename	Prompts the user for a search string. Prints a list of all the filenames (with their paths) that contain the search string, or "String not found."
Quit the program	Prints a signoff message and exits the program.

Sol:

```
import os, os.path
QUIT = '7'
COMMANDS = ('1', '2', '3', '4', '5', '6', '7')
MENU = """1 List the current directory
2 Move up
3 Move down
4 Number of files in the directory
5 Size of the directory in bytes
6 Search for a filename
7 Quit the program"""

def main():
    while True:
        print(os.getcwd())
        print(MENU)
        command = acceptCommand()
        runCommand(command)
```

```
        if command == QUIT:
            print("Have a nice day!")
            break

def acceptCommand():
    """Inputs and returns a legitimate command number."""
    command = input("Enter a number: ")
    if command in COMMANDS:
        return command
    else:
        print("Error: command not recognized")
        return acceptCommand()

def runCommand(command):
    """Selects and runs a command."""
    if command == '1':
        listCurrentDir(os.getcwd())
    elif command == '2':
        moveUp()
    elif command == '3':
        moveDown(os.getcwd())
    elif command == '4':
        print("The total number of files is", \
        countFiles(os.getcwd()))
    elif command == '5':
        print("The total number of bytes is", \
        countBytes(os.getcwd()))
    elif command == '6':
        target = input("Enter the search string: ")
        fileList = findFiles(target, os.getcwd())
        if not fileList:
            print("String not found")
        else:
            for f in fileList:
                print(f)

def listCurrentDir(dirName):
    """Prints a list of the cwd's contents."""
    lyst = os.listdir(dirName)
    print(lyst)
```

```
    #for element in lyst:
    #    print(element)

def moveUp():
    """Moves up to the parent directory."""
    os.chdir("..")

def moveDown(currentDir):
    """Moves down to the named subdirectory if it exists."""
    newDir = input("Enter the directory name: ")
    if os.path.exists(currentDir + os.sep + newDir) and \
       os.path.isdir(newDir):
        os.chdir(newDir)
    else:
        print("ERROR: no such name")

def countFiles(path):
    """Returns the number of files in the cwd and
   all its subdirectories."""
    count = 0
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            count += 1
        else:
            os.chdir(element)
            count += countFiles(os.getcwd())
            os.chdir("..")
    return count
def countBytes(path):
    """Returns the number of bytes in the cwd and
    all its subdirectories."""
    count = 0
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            count += os.path.getsize(element)
        else:
            os.chdir(element)
            count += countBytes(os.getcwd())
```

```
            os.chdir("..")
    return count

def findFiles(target, path):
    """Returns a list of the filenames that contain
    the target string in the cwd and all its subdirectories."""
    files = []
    lyst = os.listdir(path)
    for element in lyst:
        if os.path.isfile(element):
            if target in element:
                files.append(path + os.sep + element)
            else:
                os.chdir(element)
                files.extend(findFiles(target, os.getcwd()))
                os.chdir("..")
    return files

if __name__ == "__main__":
    main()
```

Namespace and Scope :

A *namespace* is a mapping from names to objects. Such a "name-to-object" mapping allows us to access an object by a name that we've assigned to it.

For example, when we do the assignment a = 2, we created a reference to the integer value object 2 and henceforth we can access it through its name a.

Namespaces are created at different moments and have different lifetimes. Different namespaces can co-exist at a given time but are completely isolated. **This isolation ensures that there are no name collisions**. **Namespaces enable programs to avoid nameclashes by associating each name with the namespace from which it is derived**. Most namespaces are currently implemented as Python dictionaries.

Examples of namespaces are:

1) **Built-in namespace** :

The namespace containing the set of built-in names which is created when we start the Python interpreter and exists as long we don't exit.

2)**global namespace**:

Each module creates its own global namespace. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits.

3**) local namespace:**

A local namespace is created when a function is called, which has all the names defined in it. The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. Similar, is the case with class methods

Note:
there is absolutely no relation between names in different namespaces

Ex:

Consider the following python script mod1.py

```
def display():
        print("hello mod1")
        print("module name=",__name__)
```

Consider the other following python script mod2.py

```
def display():
        print("hello mod2")
        print("module name=",__name__)
```

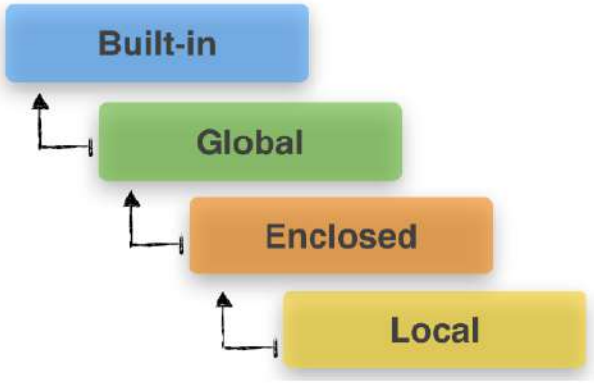
Consider the other following python script sample.py

```
import mod1
import mod2
mod1.display( )
mod2.display( )
```

scope:

A *scope* is a textual region of a Python program where a namespace is directly accessible to find the name in the namespace. So the scope decides the part of the program from where the name can be accessible. So, based on where the name is referenced in the program, the name is searched in the namespaces in a specific order as per LEGB (Local -> Enclosed -> Global -> Built-in) rule.

As per LEGB rule, when the python interpreter sees the name, first it searches local namespace. If not present in the local namespace then the namespace of enclosing function is searched. If the search in the enclosed scope is also unsuccessful then Python moves on to the global namespace, and eventually, it will search the built-in namespace. if a name cannot found in any of the namespaces, a *NameError* will is raised. This is illustrated in the following figure.



Local scope:

Variables that are defined inside a function have a local scope and are called local variables. Local variables can be accessed only inside the function in which they are defined.

total=0
 	
def sum(a, b):
 	
  total = a + b # Here total is local variable.
 	
  print ("Inside the function local total : ", total)
 	
  return total
 	
sum(10, 20)
 	
print ("Outside the function total : ", total)
	Output:
	
	Inside the function local total : 30
	
	Outside the function total : 0

def sum(a, b):
 	
  total = a + b # Here total is local variable.
 	
  print ("Inside the function local total : ", total)
 	
  return total
 	
sum(10, 20)
 	
print ("Outside the function total : ", total)
	Output:
	
	Inside the function local total : 30
	
	Traceback (most recent call last):
	
	File "D:/Python36-32/pav112.py", line 7, in <module>
	
	  print ("Outside the function total : ", total)
	
	NameError: name 'total' is not defined

Global scope:

Variables that are defined outside the function have a global scope and are called global variables. global variable can be accessed inside or outside of the function but its value cannot changed inside the function.

x = 10 def fun(): print("x inside :", x) #accessing global variable fun( ) print("x outside:", x)	Output: x inside : 10 x outside: 10

If we assign another value to a globally declared variable inside the function, a new local variable is created in the function's namespace. This assignment will not alter the value of the global variable

x = 10
 	
def fun( ):
  x=20 #local variable is created
  print("x inside :", x) #accessing local variable
 	
fun( )
print("x outside:", x)
	Output:
	
	x inside : 20
	x outside: 10

So if you need to access and change the value of the global variable from within a function, then the variable must be declared as **global.**

x = 10
 	
def fun():
  global x #accessing global variable
  x=20 #changing global variable value
  print("x inside :", x) #accessing global variable
 	
fun()
print("x outside:", x)
	Output:
	
	x inside : 20
	x outside: 20

Lifetime:

A variable's lifetime is the period of time during program execution when the variable has memory storage associated with it. When a variable comes into existence, storage is allocated for it; when it goes out of existence, storage is reclaimed.

Module variables exist for the lifetime of the program that uses them.

All the Parameters and temporary variables come into existence when they are bound to values during a function call and go out of existence when the function call terminates.

Higher order functions:

In Python, functions can be treated as first-class data objects. This means that they can be assigned to variables (as they are when they are defined), passed as arguments to other functions, returned as the values of other functions, and stored in data structures such as lists and dictionaries.

Ex:

```
def sum(a,b):
   res=a+b
```

```
    return res

add=sum    #assigning sum function to a variable. Here add is an alias for sum

print(add(5,6))
```

Ex:

```
def greet():
    print ("Welcome Python!")

def fun(f):
    print ('*'*20)
    f()
    print ('*'*20)

fun(greet)      #passing greet() function as an argument to another function fun()
```

Higher-order functions are the functions which can expect other functions as arguments and/or return functions as values.

Ex:

Mapping:

A mapping function expects two arguments - a function and a list of values(an iterable). It applies the function to each value in the list and a map object containing the results is returned. Python includes a map function for this purpose.

Ex1:

```
a,b,c=map(eval,input("enter three integers separated by comma:").split(','))
print(a,type(a), sep=' ')
print(b,type(b), sep=' ')
print(c,type(c), sep=' ')
```

Output:

```
enter three integers separated by comma:3,4.5,4+6j
3 <class 'int'>
4.5 <class 'float'>
(4+6j) <class 'complex'>
```

Ex2:

```
words=["231", "20", "-45", "99"]
print(words)
words=list(map(int,words))
print(words)
```

Output:

```
['231', '20', '-45', '99']
[231, 20, -45, 99]
```

Filtering:

The second type of higher order function is called Filtering. It expects two arguments – a function (called as predicate) and a list of values (an iterable). It returns an filter object containing all the elements of the list for which the function is true. Python includes filter function for this purpose.

Ex:

```
def odd(n):
    if n%2==0:
        return n

list(filter(odd,range(10)))
```

Output:

```
[2, 4, 6, 8]
```

Reducing:

Another type of higher order function is reducing. It expects two arguments – a function and a list of values (or a sequence). It applies a function of two arguments cumulatively to the items of a list, from left to right, so as to reduce the list to a single value. The Python functools module includes a reduce function for this purpose.

Ex:

```
from functools import reduce
list1=[5,10,15,20]
def add(x,y):
   return x+y
print(reduce(add,list1))
```

Output:

50

Anonymous functions:

An anonymous function is a function that is defined without a name. While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.

Hence, anonymous functions are also called **lambda functions**.

Syntax:

lambda <argument1,argument-2,……argument-n>: <expression>

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Ex1:

```
# Program to show the use of lambda functions
double = lambda x: x * 2
print(double(5))
```

output:

```
10
```

In the above program, lambda x: x * 2 is the lambda function. Here x is the argument and x * 2 is the expression that gets evaluated and returned.
This function has no name. It returns a function object which is assigned to the identifier double. We can now call it as a normal function.

Ex2:

```
sum = lambda a,b: (a+b)
print (sum(1,2))
print (sum(3,5))
```

Output:

```
3
8
```

Ex3:

```
L = [word for word in ['this', 'is', 'a', 'test'] if len(word)>2]

Print(L)
```

Output:

```
['this', 'test']
```

Ex4:

```
from functools import reduce
def fact(n):
    return reduce(lambda x,y:x*y, range(1,n+1))

print(fact(5))
```

Output:

120

Modules:

A module is a file with .py extension containing Python definitions and statements. so any Python file can be referenced as a module. A module can define functions, classes and variables that can then be utilized in other Python programs.

Modules allow breaking down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

For example, test.py is called a module and its name would be test

Importing modules in python:

To use the functionality present in any module, you have to import it into your current program. For this use the import keyword along with the desired module name as given below

import modulename

When interpreter comes across an import statement, it imports the module to your current program. the variables, functions inside a module can be used by a dot(.) operator along with the module name as given below

modulename.variablename

Ex:

Consider the following module test.py containing two functions add( ) and sub( ).

```
def add(x,y):
   return (x+y)
def sub(x,y):
   return (x-y)
```

now to import this module test.py in another program, just write a statement

import test

and the functions inside the module can be accessed in the following way

test.add(4,3)

test.sub(4,3)

there are more ways to import modules. Some of them are given below

from .. import statement:

The from .. import statement allows you to import specific functions/variables from a module instead of importing everything.

Syntax:

from modulename import name1[,name2[,.....nameN]]

Ex:

Consider the following module test.py containing two functions add( ) and sub( ).

```
def add(x,y):
   return (x+y)
def sub(x,y):
   return (x-y)
```

now suppose if add() function is only needed from the test.py module in the code, then write

```
from test import add
add(4,5)
```

In above example, only the add() function is imported and used. also it can be accessed directly without using the module name

from ..import* statement:

This imports all names except those beginning with an underscore (_).the syntax is given below

```
from modulename import *
```

Ex:

Consider the following module test.py containing two functions add( ) and sub( ).

```
def add(x,y):
   return (x+y)
def sub(x,y):
   return (x-y)
```

now we can import the functions add() and sub( ) as shown below

```
from test import *
add(4,3)
sub(4,5)
```

Renaming the imported module:

While importing the module, we can give the different name(more meaningful name or alias) to the module. The as keyword is used to create an alias.
For ex, consider the following

```
import numpy as np
radians = np.pi
sineValue = np.sin(radians)
print("Sine of %f radians:%f"%(radians, sineValue))
```

In the example above, we create an alias, np, when importing the numpy module, and now we can refer to the numpy module by using np instead of numpy

Module search path:

A **search path** is a list of directories that the interpreter **searches** before importing a **module**. When a module named spam is imported, the interpreter first searches for a built-in module with that name. If not found,it searches for a file named spam.py in a list of directories given by the variable sys.path.
sys.path is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- The list of directories contained in the PYTHONPATH environment variable, if it is set.
- The installation-dependent ist of directories configured at the time Python is installed.

Ex:

```
>>>import sys
>>> sys.path
['','C:\\python36\\Lib\\idlelib', 'C:\\python36\\python36.zip', 'C:\\python36\\DLLs',
'C:\\python36\\lib', 'C:\\python36',
'C:\\Users\\acer\\AppData\\Roaming\\Python\\Python36\\site-packages',
'C:\\python36\\lib\\site-packages']
```

If the required module is not present in any of the directories above, the message ModuleNotFoundError is thrown.

The dir() function:

The dir() function is used to find out all the names(or attributes) defined in a module. It returns a sorted list of strings containing the names defined in a module.

Ex:

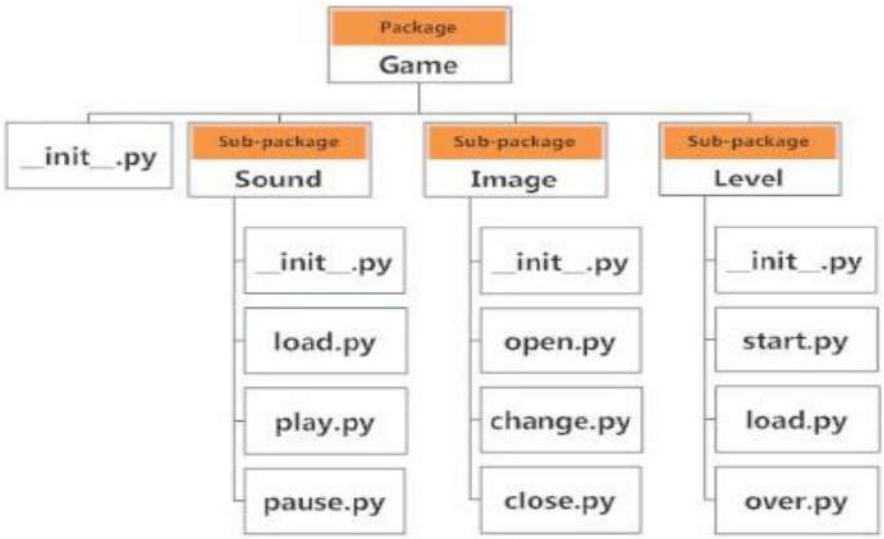
```
>>>import test
>>>dir("test")
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'add', 'sub']
```

Note: The __name__ attribute returns the name of the module and the __doc__ attribute denotes the documentation string (docstring) line written in a module code. __file__ attribute gives you the name and location of the module

Packages:

A package is a hierarchical file directory structure that has modules and other packages with in it. A directory must contain a file named __init__.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

For example, Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.



Importing module from a package:

We can import modules from packages using the dot (.) operator. For example, if want to import the start module in the above example, it is done as follows.

```
import Game.Level.start
```

Now if this module contains a function named select_difficulty(), we must use the full name to reference it.

```
Game.Level.start.select_difficulty(2)
```

The above can also be done using the following code

```
from Game.Level import start
start.select_difficulty(2)
```

PyPI(Python Package Index):

The Python Package Index (PyPI) is a repository of software for the Python programming language.
PyPI helps you find and install software developed and shared by the Python community. **PIP is a package management system and can be used to install packages from the Python Package Index(PyPI)**. Pip is a recursive acronym which stands for pip installs packages.

Installing pip:
Python 2.7.9 and later (python2 series), and Python 3.4 and later (python 3 series) already comes with pip. If you do not have PIP installed, you can download and install it from this page: https://pypi.org/project/pip/

Once pip is installed you can check it by navigating command line to the location of Python's script directory, and type the following

```
C:\Python36-32\Scripts>pip --version
```

Installing packages:

To install the package, navigate command line to the location of Python's script directory and type the following command

pip install packagename

Ex:

Suppose you want to install a package called requests (which is used to make HTTP requests). You need to issue the following command.

```
pip install requests                    # this will install latest request package
```

To install the specific version of a package you can issue the command

pip install packagename==versionnumber

Ex:

```
pip install requests==2.6.0      # this will install requests 2.6.0 package and not
                                 #the latest package
```

To install the specified minimum version you can issue the command

pip install packagename>=versionnumber

if the specified minimum version is not available it will install the latest version

Ex:

```
pip install requests>=2.6.0      # specify a minimum version if it's not available
```

Uninstalling packages

To uninstall the package use the command below.

pip uninstall package_name

Ex: pip uninstall requests

Listing installed packages

To list all the installed packages on the computer system use the following command

pip list

upgrade package:

To upgrade the installed package use the following command

pip install --upgrade packagename

Ex:

pip install --upgrade requests