PYTHON PROGRAMMING
UNIT -III

List and Dictionaries: Lists, Defining Simple Functions, Dictionaries
Design with Function: Functions as Abstraction Mechanisms, Problem Solving with Top Down Design, Design with Recursive Functions, Case Study Gathering Information from a File System, Managing a Program's Namespace, Higher Order Function.
Modules: Modules, Standard Modules, Packages..

There are four collection data types in the Python programming language:
- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and un indexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

List:
A list is a sequence of values (similar to an array in other programming languages but more versatile). The values in a list are called items or sometimes elements.
The important properties of Python lists are as follows:
- Lists are ordered – Lists remember the order of items inserted.
- Accessed by index – Items in a list can be accessed using an index.
- Lists can contain any sort of object – It can be numbers, strings, tuples and even other lists.
- Lists are changeable (mutable) – You can change a list in-place, add new items, and delete or update existing items.

Create a List
There are several ways to create a new list; the simplest is to enclose the values in square brackets []
```
# A list of integers
L = [1, 2, 3]
# A list of strings
L = ['red', 'green', 'blue']
```
The items of a list don't have to be the same type. The following list contains an integer, a string, a float, a complex number, and a boolean.
```
# A list of mixed datatypes
L = [ 1, 'abc', 1.23, (3+4j), True]
```
A list containing zero items is called an empty list and you can create one with empty brackets []
```
# An empty list
L = []
```
There is one more way to create a list based on existing list, called **List comprehension.**
Replacing an element of the list can be done with the help of assignment statement

- List are mutable, meaning, their elements can be changed unlike string or tuple.
- We can use assignment operator (=) to change an item or a range of items.

```
# change the 1st item
>>> li = [10,20,30,40,50,60]
>>> li
[10, 20, 30, 40, 50, 60]
>>> li[0]=100
>>> li
[100, 20, 30, 40, 50, 60]
```

Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1,2,3] : print (x,end = ' ')	1 2 3	Iteration

Accessing Elements from a List(Searching)

Elements from the list can be accessed in various ways:

Index based: You can use the index operator to access the element from a list. In python, the indexing starts from 0 and ends at n-1, where n is the number of elements in the list. So, a list having 5 elements will have an index from 0 to 4.

Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError.

How to slice lists in Python?

We can access a range of items in a list by using the slicing operator :(colon).

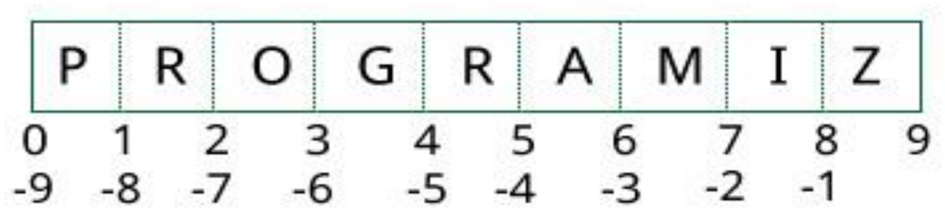
```
# List slicing in Python
my_list = ['p','r','o','g','r','a','m','i','z']
# elements 3rd to 5th
print(my_list[2:5])
# elements beginning to 4th
```

```
print(my_list[:-5])
# elements 6th to end
print(my_list[5:])
# elements beginning to end
print(my_list[:])
```

Output

```
['o', 'g', 'r']
['p', 'r', 'o', 'g']
['a', 'm', 'i', 'z']
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need two indices that will slice that portion from the list.



Element Slicing from a list in Python

IndexError: list index out of range

index[-1], index[-2] #negative indexing, here -1 means select first element from the last

(5, 4)

Loop through the List: You can loop through the list items by using a for loop.

Ex: thislist = ["apple", "banana", "cherry"]

```
for x in thislist:
 print(x)
```

Output: apple

banana

cherry

Check if item exists: To determine if a specified item is present in a list use the in keyword.

Ex: thislist = ["apple", "banana", "cherry"]

```
if "apple" in thislist:
print("Yes, 'apple' is in the fruits list")
```

Output: Yes, ‘apple’ is in the fruits list.

Length of the List: To determine how many items a list has, use the len() function. Print the number of items in the list.

Ex: thislist = ["apple", "banana", "cherry"]

```
print(len(thislist))
```

Output: 3

(Inserting)Add Items in the List:

1. To add an item to the end of the list, use the append() method.
   Using the append() method to append an item.

Ex: thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
Output: apple banana cherry orange

2. To add an item at the specified index, use the insert() method.

Insert an item as the second position.
Ex: thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
Output: apple orange banana cherry

Remove Item from the List:

There are several methods to remove items from a list.Some of them are:

a. The remove() method removes the specified item.

Ex: thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
Output: apple cherry

b. The pop() method removes the specified index, (or the last item if index is not specified).

Ex: thislist = ["apple", "banana", "cherry"]

thislist.pop()

print(thislist)

Output: apple banana

c. The del keyword removes the specified index.

Ex: thislist = ["apple", "banana", "cherry"]

del thislist[0]

print(thislist)

Output: banana cherry

d. The del keyword can also delete the list completely.

Ex: thislist = ["apple", "banana", "cherry"]

del thislist

Output:

e. The clear() method empties the list.

Ex: thislist = ["apple", "banana", "cherry"]

thislist.clear()
print(thislist)
Output: [ ]

Copy a List:

A list can be copied using the = operator. For example,
old_list = [1, 2, 3]
new_list = old_list

The problem with copying lists in this way is that if you modify new_list, old_list is also modified. It is because the new list is referencing or pointing to the same old_list object.
You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.
There are three ways to make a copy, one way is to use the built-in List method copy().

1. Make a copy of a list with the **copy( ) method.**

Ex: thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
Output: apple banana cherry

2. Another way to make a copy is to use the built-in method **list( ).** Make a copy of a list with the list( ) method.

Ex: thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
Output: apple banana cherry

3.Using slicing technique

This is the easiest and the fastest way to clone a list. This method is considered when we want to modify a list and also keep a copy of the original. In this we make a copy of the list itself, along with the reference. This process is also called **cloning.**

```
# shallow copy using the slicing syntax
# mixed list
list = ['cat', 0, 6.7]
# copying a list using slicing
new_list = list[:]
# Adding an element to the new list
new_list.append('dog')
# Printing new and old list
print('Old List:', list)
print('New List:', new_list)
```

Output

```
Old List: ['cat', 0, 6.7]
```

New List: ['cat', 0, 6.7, 'dog']

Join Two LISTS: There are several ways to join, or concatenate, two or more lists in Python.
1. One of the easiest ways are by using **the + operator**.
Ex: list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)
Output: ‘a’ ‘b’ ‘c’ 1 2 3
2. Another way to join two lists are **by appending** all the items from list2 into list1, one by one.
Ex: Append list2 into list1.
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
for x in list2:
list1.append(x)
print(list1)
Output: [‘a’ ‘b’ ‘c’ 1 2 3]
3. We can use **the extend()** method, which purpose is to add elements from one list to another list. Use
the extend() method to add list2 at the end of list1.
Ex: list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
list1.extend(list2)
print(list1)
Output: [‘a’ ‘b’ ‘c’ 1 2 3]

- The **List() Constructor**: It is also possible to use the list() constructor to make a new list. Using the list() constructor to make a List.

Ex: thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)
Output: [‘apple’ ‘banana’ ‘cherry’]

Python Nested List

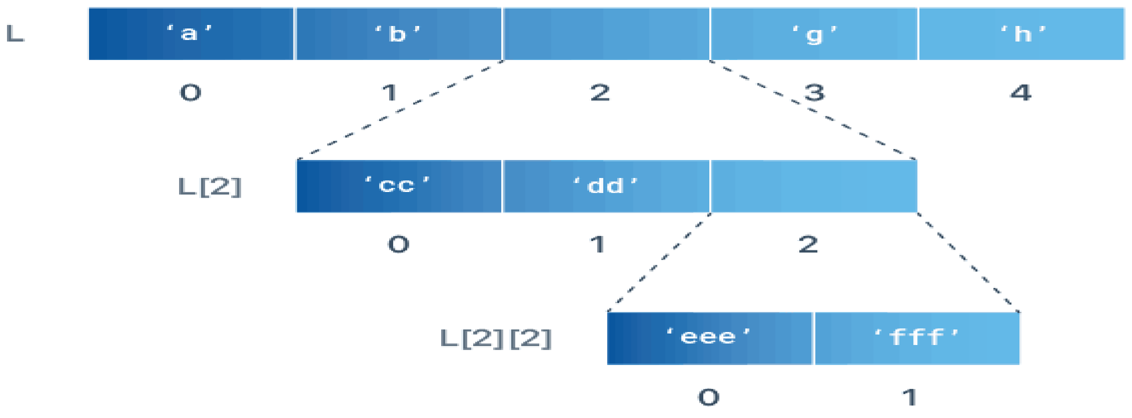
A list can contain any sort object, even another list (sublist), which in turn can contain sublists themselves, and so on. This is known as nested list.
You can use them to arrange data into hierarchical structures.

Create a Nested List

A nested list is created by placing a comma-separated sequence of sublists.
L = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', 'h']

Access Nested List Items by Index

You can access individual items in a nested list using multiple indexes.
The indexes for the items in a nested list are illustrated as below:



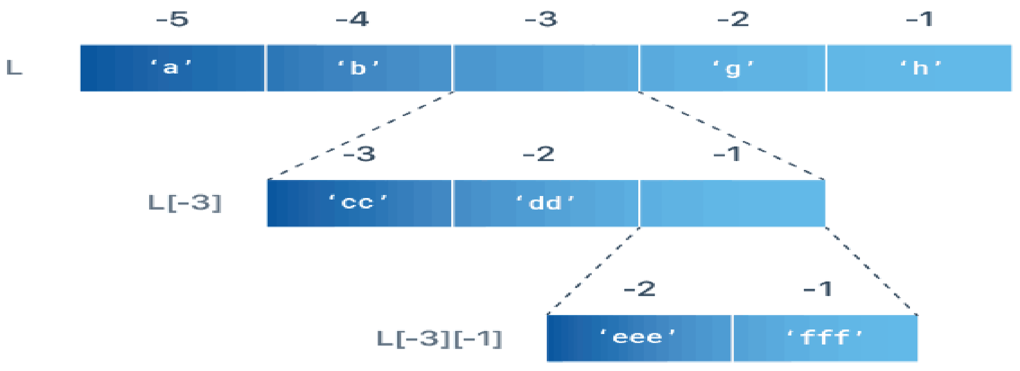
```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']

print(L[2])
# Prints ['cc', 'dd', ['eee', 'fff']]

print(L[2][2])
# Prints ['eee', 'fff']

print(L[2][2][0])
# Prints eee
```

Negative List Indexing In a Nested List

You can access a nested list by negative indexing as well.

Negative indexes count backward from the end of the list. So, L[-1] refers to the last item, L[-2] is the second-last, and so on.

The negative indexes for the items in a nested list are illustrated as below:



```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
print(L[-3])
# Prints ['cc', 'dd', ['eee', 'fff']]
```

```
print(L[-3][-1])
#Prints ['eee', 'fff']
print(L[-3][-1][-2])
# Prints eee
```

Python List Methods

Methods that are available with list objects in Python programming are tabulated below. They are accessed as list.method(). Some of the methods have already been used above.

Python List Methods

append() - Add an element to the end of the list

extend() - Add all elements of a list to the another list

insert() - Insert an item at the defined index

remove() - Removes an item from the list

pop() - Removes and returns an element at the given index

clear() - Removes all items from the list

index() - Returns the index of the first matched item

count() - Returns the count of the number of items passed as an argument

sort() - Sort items in a list in ascending order

reverse() - Reverse the order of items in the list

copy() - Returns a shallow copy of the list

Some examples of Python list methods:

```
# Python list methods
my_list = [3, 8, 1, 6, 0, 8, 4]
# Output: 1
print(my_list.index(8))

# Output: 2
print(my_list.count(8))

my_list.sort()
```

```
# Output: [0, 1, 3, 4, 6, 8, 8]
print(my_list)
my_list.reverse()

# Output: [8, 8, 6, 4, 3, 1, 0]
print(my_list)
```

Output

```
1
2
[0, 1, 3, 4, 6, 8, 8]
[8, 8, 6, 4, 3, 1, 0]
```

List comprehensions

A Python list comprehension is a method for constructing a list from a list, a range, or other iterable structure. List comprehensions are frequently used to extract data from other lists or to generate new data from an existing data list.

List comprehension is an elegant and concise way to create a new list from an existing list in Python. A list comprehension consists of an expression followed by for statement inside square brackets.

Syntax of List Comprehension

list_variable = [expression for item in collection]

ex1: Here is an example to make a list with each item being increasing power of 2.

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

Output

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

This code is equivalent to:

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

The first expression generates elements in the list followed by a for loop over some collection of data which would evaluate the expression for every item in the collection.

```
V = [2**i for i in range(13)]
V
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
```

Dictionaries

- 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.
- Python dictionary is an **unordered collection of items**. Each item of a dictionary has a key/value pair.
- Dictionaries are optimized to retrieve values when the key is known.

Creating Python Dictionary

- Creating a dictionary is as simple as placing items inside curly braces {} separated by commas. An item has a key and a corresponding value that is expressed as a pair (key: value).
- While the 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.**

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:

```
d = {    <key>: <value>,
    <key>: <value>,     .
      .
      .
    <key>: <value>
}
```

# empty dictionary
```
my_dict = {}
```

# dictionary with integer keys
```
my_dict = {1: 'apple', 2: 'ball'}
```

# dictionary with mixed keys
```
my_dict = {'name': 'John', 1: [2, 4, 3]}
```

# using dict()
```
my_dict = dict({1:'apple', 2:'ball'})
```

# from sequence having each item as a pair
```
my_dict = dict([(1,'apple'), (2,'ball')])
```
As you can see from above, we can also create a dictionary using the built-in dict() function.

Accessing Elements from Dictionary

While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside **square brackets []** or with the **get()** method.

If we use the square brackets [], **KeyError** is raised in case a key is not found in the dictionary.

On the other hand, the get() method returns **None** if the key is not found.

Example:

```
my_dict = {'name': 'Jack', 'age': 26}

print(my_dict['name'])

print(my_dict.get('age'))

# Trying to access keys which doesn't exist throws error None

print(my_dict.get('address'))

print(my_dict['address'])
```

Output

```
Jack

26

None

Traceback (most recent call last):

  File "<string>", line 15, in <module>

    print(my_dict['address'])

KeyError: 'address'
```

Changing and Adding Dictionary elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.

If the key is already present, then the existing value gets updated. In case the key is not present, a new (key: value) pair is added to the dictionary.

```
# Changing and adding Dictionary Elements
my_dict = {'name': 'Jack', 'age': 26}
# update value
my_dict['age'] = 27
print(my_dict)
# add item
my_dict['address'] = 'Downtown'

print(my_dict)
```

Output

```
{'name': 'Jack', 'age': 27}

{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

Removing elements from Dictionary

- We can remove a particular item in a dictionary by using the pop() method. This method removes the item with the specified key name and returns the value.
- The popitem() method can be used to remove and return an arbitrary (key, value) item pair from the dictionary. This popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):
- All the items can be removed at once, using the clear() method.
- The del keyword removes the item with the specified key name:
- The del keyword can also delete the dictionary completely:

Example:

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

Output:

```
16
{1: 1, 2: 4, 3: 9, 5: 25}
(5, 25)
{1: 1, 2: 4, 3: 9}
{}
Traceback (most recent call last):
  File "<string>", line 30, in <module>
    print(squares)
NameError: name 'squares' is not defined
```

Adding and removing keys

To add a new item into a dictionary, we can use the subscript[ ] operator. The syntax to add an item to a dictionary is Dictionary_Name[key] = value

Example:

```
>>> phonebook["Amol"]="1212121211"
>>> phonebook
{'Amit': '9948911111', 'Arun': '8899776655', 'Amol': '1212121211'}
```

To delete any entry from a dictionary. The del operator is used to remove a key and its associated value. If a key is in a dictionary then it is removed otherwise Python raises an error. The syntax used to remove an element from a dictionary is
del dictionary_name[key] Example:

```
>>> phonebook
{'Amit': '9948911111', 'Arun': '8899776655', 'Amol': '1212121211'}
>>> del phonebook["Amol"]
>>> phonebook
{'Amit': '9948911111', 'Arun': '8899776655'}
```

Accessing and replacing values

Accessing of value associated key from the dictionary can be done using the subscript [ ] operator. Dictionary_Name[key]

```
>>> phonebook
{'Amit': '9948911111', 'Arun': '8899776655'}
>>>
```

Iterating Through a Dictionary or Traversing Dictionaries

The for loop is used to traverse all the keys and values of a dictionary. A variable of the for loop is bound to each key in an unspecified order. It means it retrieves the key and its value in any order.

Example:

```
Grades={"Ramesh":"A","Viren":"B","Kumar":"C"}
for key in Grades:
    print(key,":",str(Grades[key]))
```

Output:

```
Ramesh : A
Viren : B
Kumar : C
```

If you use a dictionary in a for loop, it traverses the keys of the dictionary by default.

```
D = {'name': 'Bob',     'age': 25,     'job': 'Dev'}
for x in D:
   print(x)
```

output:

```
name age job
```

Get All Keys, Values and Key:Value Pairs:

There are three dictionary methods that return all of the dictionary's keys, values and key-value pairs: keys(), values(), and items(). These methods are useful in loops that need to step through dictionary entries one by one.

All the three methods return **iterable object**. If you want a true list from these methods, wrap them in a list() function.

```
D = {'name': 'Bob',     'age': 25,     'job': 'Dev'}

# get all keys

print(D.keys())

# get all values

print(list(D.values()))

# get all pairs

print(list(D.items()))
```

output:

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

['Bob', 25, 'Dev']

[('name', 'Bob'), ('age', 25), ('job', 'Dev')]

# Prints False
```

Python Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python.

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

Here is an example to make a dictionary with each item being a pair of a number and its square.

```
# 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}
```

This code is equivalent to

```
squares = {}
```

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

Output

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

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

Nested Dictionary

In Python, a nested dictionary is a dictionary inside a dictionary. It's a collection of dictionaries into one single dictionary.

```
nested_dict = { 'dictA': {'key_1': 'value_1'},
                'dictB': {'key_2': 'value_2'}}
```

Here, the nested_dict is a nested dictionary with the dictionary dictA and dictB. They are two dictionary each having own key and value.

Create a Nested Dictionary

A nested dictionary is created the same way a normal dictionary is created. The only difference is that each value is another dictionary.

```
D = {'emp1': {'name': 'Bob', 'job': 'Mgr'},
     'emp2': {'name': 'Kim', 'job': 'Dev'},
     'emp3': {'name': 'Sam', 'job': 'Dev'}}
```

The dict() Constructor

There are several ways to create a nested dictionary using a type constructor called dict().
To create a nested dictionary, simply pass dictionary key:value pair as keyword arguments to dict() Constructor.

```
D = dict(emp1 = {'name': 'Bob', 'job': 'Mgr'},
         emp2 = {'name': 'Kim', 'job': 'Dev'},
         emp3 = {'name': 'Sam', 'job': 'Dev'})
print(D)
```

output:

```
{'emp1': {'name': 'Bob', 'job': 'Mgr'},   'emp2': {'name': 'Kim', 'job': 'Dev'},
 'emp3': {'name': 'Sam', 'job': 'Dev'}}
```

You can use dict() function along with the zip() function, to combine separate lists of keys and values obtained dynamically at runtime.

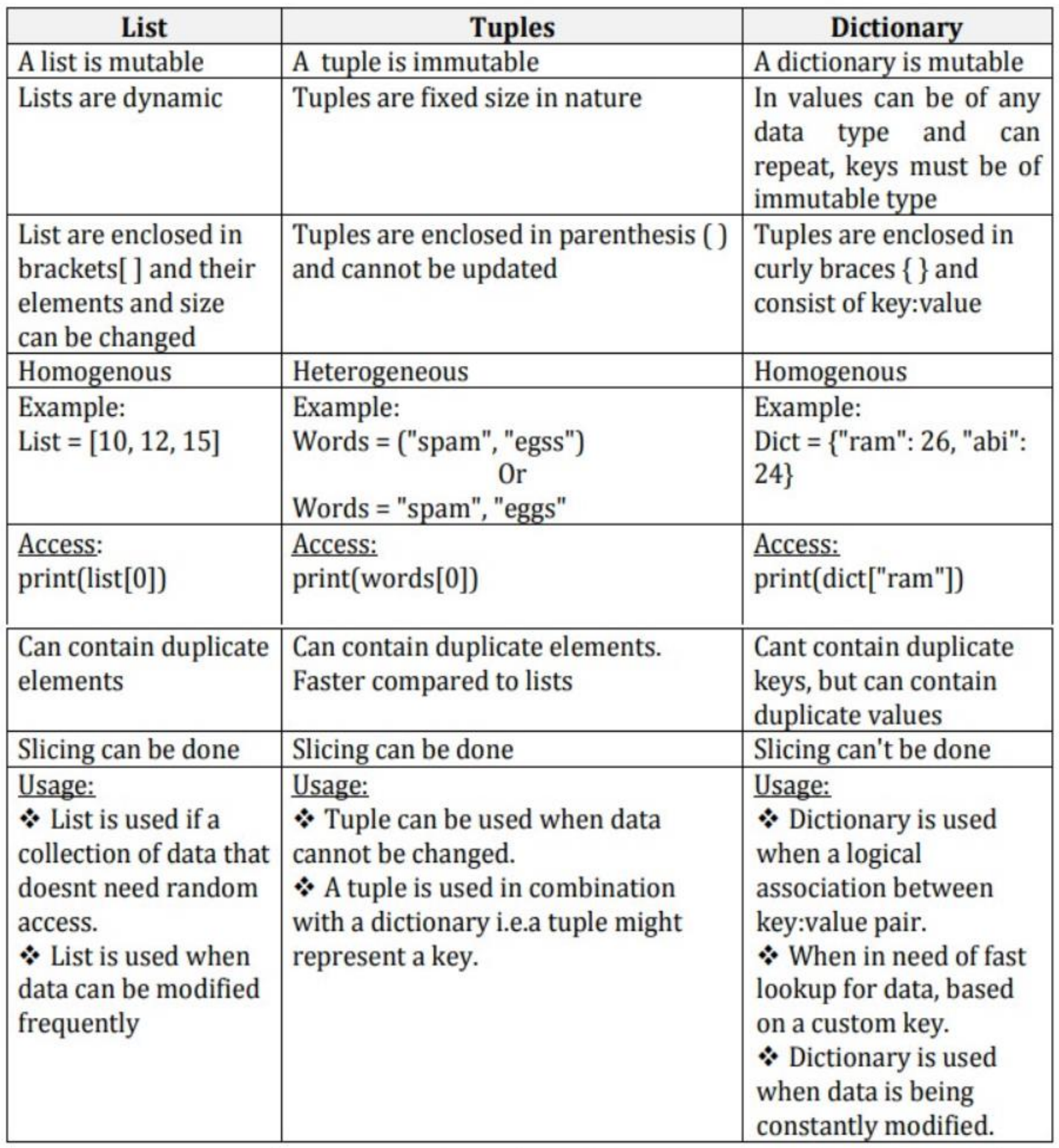
```
IDs = ['emp1','emp2','emp3']
EmpInfo = [{'name': 'Bob', 'job': 'Mgr'},
        {'name': 'Kim', 'job': 'Dev'},
        {'name': 'Sam', 'job': 'Dev'}]
D = dict(zip(IDs, EmpInfo))

print(D)
```

output:

```
{'emp1': {'name': 'Bob', 'job': 'Mgr'},  'emp2': {'name': 'Kim', 'job': 'Dev'},
 'emp3': {'name': 'Sam', 'job': 'Dev'}}
```

Difference between List and Dictionary:

LIST	DICTIONARY
List is a collection of index values pairs as that of array in c++.	Dictionary is a hashed structure of **key and value** pairs.
List is created by placing elements in **[ ]** seperated by commas “, “	Dictionary is created by placing elements in **{ }** as “key”:”value”, each key value pair is seperated by commas “, ”
The indices of list are integers starting from 0.	The keys of dictionary can be of any data type.
The elements are accessed via indices.	The elements are accessed via key-values.
The order of the elements entered are maintained.	There is no guarantee for maintaining order.



List	Tuples	Dictionary
A list is mutable	A tuple is immutable	A dictionary is mutable
Lists are dynamic	Tuples are fixed size in nature	In values can be of any data type and can repeat, keys must be of immutable type
List are enclosed in brackets[ ] and their elements and size can be changed	Tuples are enclosed in parenthesis ( ) and cannot be updated	Tuples are enclosed in curly braces { } and consist of key:value
Homogenous	Heterogeneous	Homogenous

Example:
List = [10, 12, 15]
	Example:
	Words = ("spam", "egss")
	Or
	Words = "spam", "eggs"
		Example:
		Dict = {"ram": 26, "abi": 24}

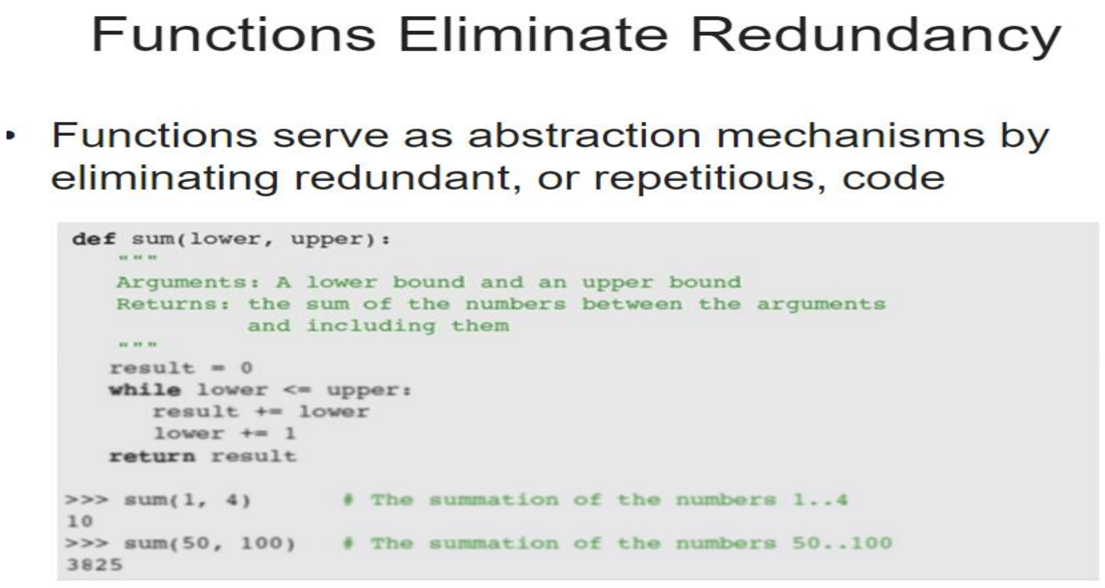
Access: print(list[0])	Access: print(words[0])	Access: print(dict["ram"])
Can contain duplicate elements	Can contain duplicate elements. Faster compared to lists	Cant contain duplicate keys, but can contain duplicate values
Slicing can be done	Slicing can be done	Slicing can't be done
Usage: ❖ List is used if a collection of data that doesnt need random access. ❖ List is used when data can be modified frequently	Usage: ❖ Tuple can be used when data cannot be changed. ❖ A tuple is used in combination with a dictionary i.e.a tuple might represent a key.	Usage: ❖ Dictionary is used when a logical association between key:value pair. ❖ When in need of fast lookup for data, based on a custom key. ❖ Dictionary is used when data is being constantly modified.

Python Function

- Functions are the most important aspect of an application.
- A function can be defined as the organized block of reusable code, which can be called whenever required.
- A function in any programming language is a single comprehensive unit (self-contained block) containing a block of code that performs a specific task.
- Python allows us to divide a large program into the basic building blocks known as a function.
- A function can be called multiple times to provide reusability and modularity to the Python program.
- The Function helps to programmer to break the program into the smaller part. It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.
- Python provide us various inbuilt functions like range() or print(). Although, the user can create its functions, which can be called user-defined functions.

Functions as Abstraction Mechanisms

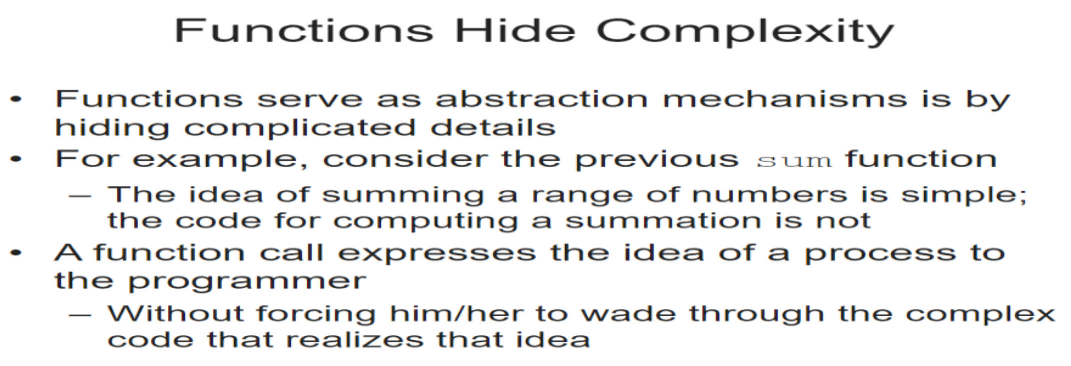
- Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but inner working is hidden. User is familiar with that **"what function does"** but they don't know **"how it does."**
- In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity. It also enhances the application efficiency. In Python, abstraction can be achieved by using abstract classes and interfaces.



Functions Eliminate Redundancy

- Functions serve as abstraction mechanisms by eliminating redundant, or repetitious, code

```
def sum(lower, upper):
    """
    Arguments: A lower bound and an upper bound
    Returns: the sum of the numbers between the arguments
             and including them
    """
   result = 0
   while lower <= upper:
      result += lower
      lower += 1
   return result

>>> sum(1, 4)       # The summation of the numbers 1..4
10
>>> sum(50, 100)    # The summation of the numbers 50..100
3825
```



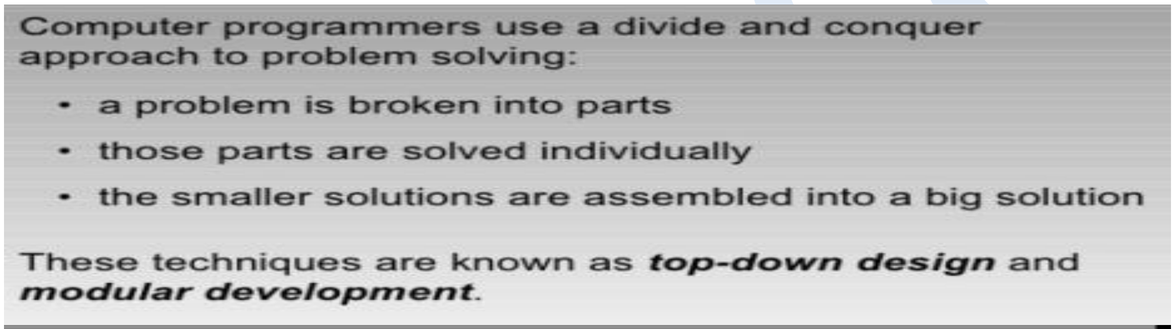
Functions Hide Complexity

- Functions serve as abstraction mechanisms is by hiding complicated details
- For example, consider the previous `sum` function
  - The idea of summing a range of numbers is simple; the code for computing a summation is not
- A function call expresses the idea of a process to the programmer
  - Without forcing him/her to wade through the complex code that realizes that idea

Top-Down Design Model:

In the top-down model, an overview of the system is formulated without going into detail for any part of it. Each part of it then refined into more details, defining it in yet more details until the entire specification is detailed enough to validate the model.

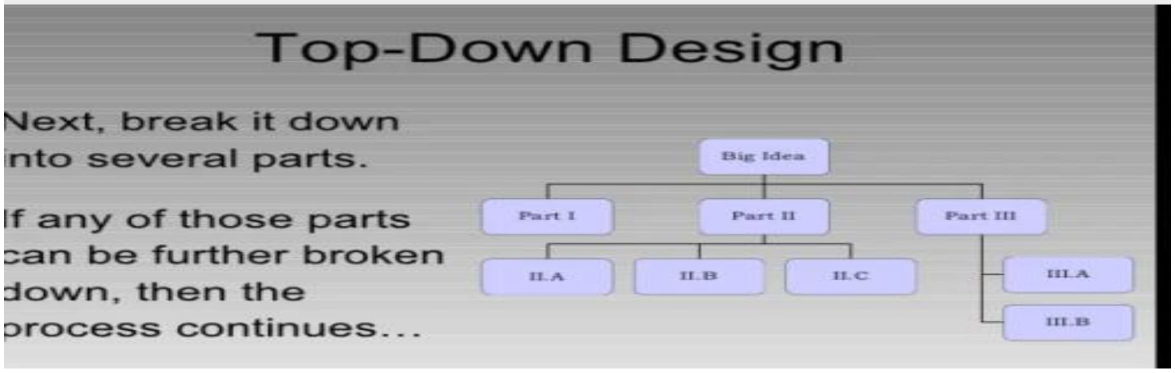
- It is process of designing a solution to a problem by systematically breaking a problem into smaller ,more manageable parts.



Computer programmers use a divide and conquer approach to problem solving:

- a problem is broken into parts
- those parts are solved individually
- the smaller solutions are assembled into a big solution

These techniques are known as ***top-down design*** and ***modular development***.



Top-Down Design

Next, break it down into several parts.

If any of those parts can be further broken down, then the process continues...

It is conjointly referred to as Stepwise refinement where,

1. We break the problem into parts,
2. Then break the parts into parts soon and now each of parts will be easy to do.

Advantages:

- Breaking problems into parts help us to identify what needs to be done.
- At each step of refinement, new parts will become less complex and therefore easier to solve.
- Parts of the solution may turn out to be reusable.
- Breaking problems into parts allows more than one person to solve the problem

There are mainly two types of functions.

Built-in functions - The built-in functions are those functions that are pre-defined in Python.

Examples:
abs(),any(),all(),ascii(),bin(),bool(),callable(),chr(),compile(),classmethod(),delattr(),dir(),divmod(),st at
icmethod(),filter(),getattr(),globals(),exec(),hasattr(),hash(),isinstance(),issubclass(),iter(),locals( ),
map(), next(),memoryview(),object(),property(),repr(),reversed(),vars(),__import__(),super()

User-define functions - The user-defined functions are those define by the user to perform the specific task.

Advantage of Functions in Python

Using functions, we can avoid rewriting the same logic/code again and again in a program.
We can call Python functions multiple times in a program and anywhere in a program.
We can track a large Python program easily when it is divided into multiple functions.
Reusability is the main achievement of Python functions.

Creating a Function

Python provides the **def** keyword to define the function. The syntax of the define function is given below.

Syntax:

```
def my_function(parameters):
    """docstring"""
    function_block
return expression
```

- Keyword def that marks the start of the function header.
- A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of the function header.
- documentation string (docstring) to describe what the function does which is optional.
- One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
- return statement to return a value from the function which is optional.

Function Calling

In Python, after the function is created, we can call it from another function. A function must be defined before the function call; otherwise, the Python interpreter gives an error. To call the function, use the function name followed by the parentheses.
Consider the following example of a simple example that prints the message "Hello World".

```
#function definition
```

```
def hello_world():
    print("hello world")
# function calling
hello_world()
```

Output:

```
hello world
```

Docstring

The first string after the function is called the Document string or Docstring in short. These strings are used to describe the functionality of the function. The use of docstring in functions is optional but it is considered a good practice.

We can access the docstring by __doc__ attribute. For example function_name.__doc__.

Note: The doc attribute has two underscores(__) before and after. Using single underscore (_) will raise an error.

Here is the example to show how we can access the docstring in Python functions.

```
 def Hello():
  """ Hello World """
  print ('Hi')
print ("The doctsring of the function Hello is: "+ Hello.__doc__)
```

Output:

This script will generate following output.

```
The docstring of the function Hello is: Hello World
```

The return statement

The return statement is used at the end of the function and returns the result of the function. It terminates the function execution and transfers the result where the function is called. The return statement cannot be used outside of the function.

Syntax

```
 return [expression_list]
```

It can contain the expression which gets evaluated and value is returned to the caller function. If the return statement has no expression or does not exist itself in the function then it returns the **None** object.

Example 1

```
# Defining function
def sum():
    a = 10
    b = 20
    c = a+b
    return c
# calling sum() function in print statement
print("The sum is:",sum())
```

Output:

```
The sum is: 30
```

In the above code, we have defined the function named **sum,** and it has a statement **c = a+b,** which computes the given values, and the result is returned by the return statement to the caller function.

Example 2 Creating function without return statement

```
# Defining function
def sum():
    a = 10
    b = 20
    c = a+b
# calling sum() function in print statement
print(sum())
```

Output: None

In the above code, we have defined the same function without the return statement as we can see that the **sum()** function returned the **None** object to the caller function.

Arguments in function

The arguments are types of information which can be passed into the function. The arguments are specified in the parentheses. We can pass any number of arguments, but they must be separate them with a comma.

Consider the following example, which contains a function that accepts a string as the argument.

Example 1

```
#defining the function
def func (name):
        print("Hi ",name)
#calling the function
func("Devansh")
```

Output:

```
Hi  Devansh
Hi Devansh
```

Example 2

```
#Python function to calculate the sum of two variables
#defining the function
def sum (a,b):
    return a+b;
#taking values from the user
a = int(input("Enter a: "))
```

```
b = int(input("Enter b: "))

#printing the sum of a and b
print("Sum = ",sum(a,b))
```

Output:

```
Enter a: 10
Enter b: 5
Sum =  15
```

Call by reference in Python

In Python, call by reference means passing the actual value as an argument in the function. All the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.

Example 1 Passing mutable Object (List)

```
#defining the function
def change_list(list1):
    list1.append(20)
    list1.append(30)
    print("list inside function = ",list1)
#defining the list
list1 = [10,30,40,50]
#calling the function
change_list(list1)
print("list outside function = ",list1)
```

Output:

```
list inside function =  [10, 30, 40, 50, 20, 30]
list outside function =  [10, 30, 40, 50, 20, 30]
```

The pass Statement

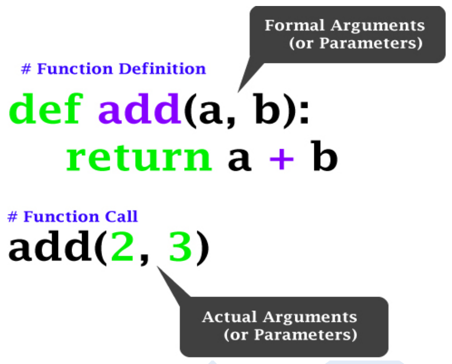
function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example

```
def myfunction():
  pass
```

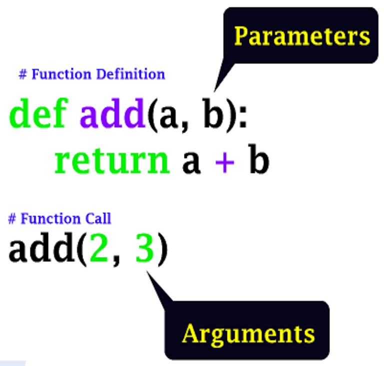
Types of arguments

- .A Python Function can have two types of Arguments / Parameters, First is **Actual Arguments** and Second is **Formal Arguments**.
- **Formal arguments** are identifiers used in the function definition to represent corresponding actual arguments.
  Actual arguments are values(or variables)/expressions that are used inside the parentheses of a function call.
- Above definition and terminology of Formal/Actual arguments is same across different programming languages.
- The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.
- From a function's perspective:
- A parameter is the variable listed inside the parentheses in the function definition.
- An argument is the value that is sent to the function when it is called.



```
# Function Definition
def add(a, b):
    return a + b

# Function Call
add(2, 3)
```

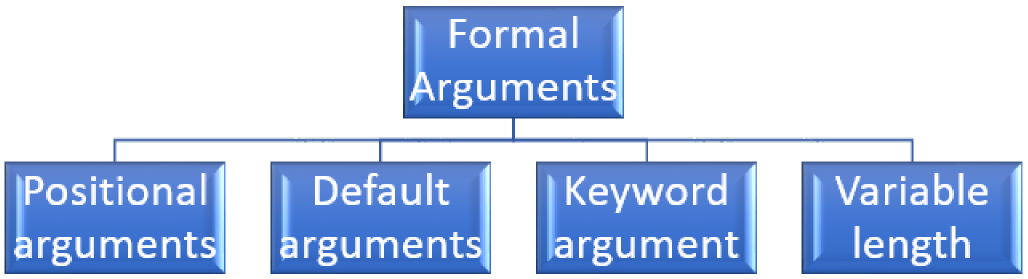
Formal Arguments (or Parameters)
Actual Arguments (or Parameters)



```
# Function Definition
def add(a, b):
    return a + b

# Function Call
add(2, 3)
```

There may be several types of arguments which can be passed at the time of function call

Formal parameters :are the variables defined by the function that receives values when the function is called. According to the above program, the values 2 and 3 are passed to the function addition. In the addition function, there are two variables called a and b. The value 2 is copied into variable a, and value 3 is copied into variable b. The variable a and b are not the actual parameters. They are copies of the actual parameters. They are known as formal parameters. These variables are only accessible within the method. After printing the addition of two numbers, the control is returned back to the main program.



Required arguments or positional arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

. If either of the arguments is not provided in the function call, or the position of the arguments is changed, the Python interpreter will show the error.

When the function call statement matches the number and order of arguments as defined in the function definition, this is called positional argument.

```
#!/usr/bin/python
# Function definition is here
def printme( str ):
  "This prints a passed string into this function"
  print (str)
  return;

# Now you can call printme function
printme()
```

When the above code is executed, it produces the following result −

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Example 2

```
def func(name):
   message = "Hi "+name
   return message
name = input("Enter the name:")
print(func(name))
```

Output:

```
Enter the name: John
Hi John
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   Print( str)
   return;

# Now you can call printme function
printme( str = "My string")
```

When the above code is executed, it produces the following result −

```
My string
```

The following example gives more clear picture. Note that the order of parameters does not matter.

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print ("Name: ", name)
   print ("Age ", age)
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

```
# Function definition is here
def printinfo( name, age = 35 ):
  "This prints a passed info into this function"
  print ("Name: ", name)
  print( "Age ", age)
  return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

These arguments are used when the number of arguments to be passed to the function is not known beforehand.

There are two types of variable-length arguments namely variable-length positional arguments and variable-length keyword arguments

i) Variable-length positional arguments (*args) or arbitrary positional arguments or Non-Keyword Arguments

When you prefix a parameter with an asterisk (*) symbol, it collects all the positional arguments and stores them in the form of a tuple. The below-given function prints all the arguments passed to the function in the form of a tuple.

Example1:

```
def student(*details):
        print(details)
student("sai","ravi",90,9.9)
```

output:

```
('sai', 'ravi', 90, 9.9)
```

Example2

```python
def printme(*names):
    print("type of passed argument is ",type(names))
    print("printing the passed arguments...")
    for name in names:
        print(name)
printme("john","David","smith","nick")
printme ( 70, 60, 50 )
```

Output:

```
type of passed argument is  <class 'tuple'>
printing the passed arguments...
john
David
smith
nick
70
60
50
```

In the above code, we passed ***names** as variable-length argument. We called the function and passed values which are treated as tuple internally. The tuple is an iterable sequence the same as the list. To print the given values, we iterated ***arg names** using for loop.

ii) Variable-length keyword arguments (kwargs) arbitrary keyword arguments**

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

**kwargs works only for keyword arguments. It collects all the keyword arguments into a dictionary, where the argument names are the keys, and arguments are the corresponding dictionary values.

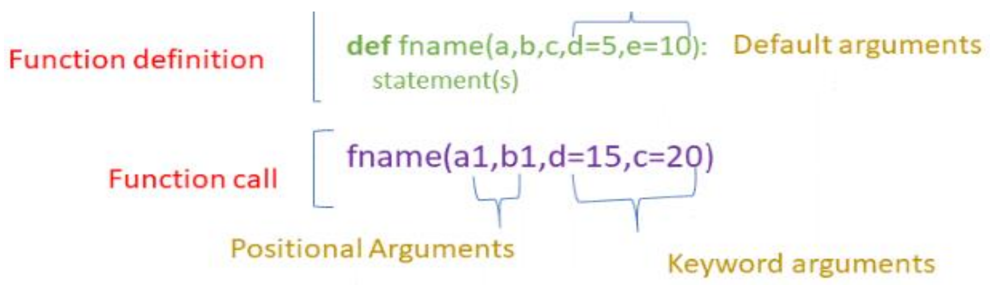
Example1:

```python
def food(**kwargs):
    print(kwargs)
food(a="Apple")
food(fruits="Orange", Vagitables="Carrot")
```

Output:

```
{'a': 'Apple'}
{'fruits': 'Orange', 'Vagitables': 'Carrot'}
```

{'a': 'Apple'}
{'fruits': 'Orange', 'Vagitables': 'Carrot'}



Recursive Function:

• **A function that calls itself is known as Recursive Function. Python Recursive Function: Introduction**

Recursion means iteration. A function is called recursive, if the body of function calls the function itself until the condition for recursion is true. Thus, a Python recursive function has a termination condition.

Why does a recursive function in Python has termination condition?

Well, the simple answer is to prevent the function from infinite recursion.

When a function body calls itself with any condition, this can go on forever resulting an infinite loop or recursion. This termination condition is also called base condition.

Is there any special syntax for recursive functions?

The answer is **NO**.

In Python, there is no syntactic difference between functions and recursive functions. There is only a logical difference.

Advantages of Python Recursion

1. Reduces unnecessary calling of function, thus reduces length of program.
2. Very flexible in data structure like stacks, queues, linked list and quick sort.
3. Big and complex iterative solutions are easy and simple with Python recursion.
4. Algorithms can be defined recursively making it much easier to visualize and prove.

Disadvantages of Python Recursion

1. Slow.
2. Logical but difficult to trace and debug.
3. Requires extra storage space. For every recursive calls separate memory is allocated for the variables.
4. Recursive functions often throw a Stack Overflow Exception when processing or operations are too large.

Python Recursion: Example

Let’s get an insight of Python recursion with an example to find the factorial of 3.

```
3! = 3 * 2! = 3 * (2 * 1!) = 3 * 2 * 1
```

This is how a factorial is calculated. Let’s implement this same logic into a program.

```
#recursive function to calculate factorial
def fact(n):
  """ Function to find factorial """
  if n == 1:
    return 1
  else:
    return (n * fact(n-1))
```

```
print ("3! = ",fact(3))
```

Output

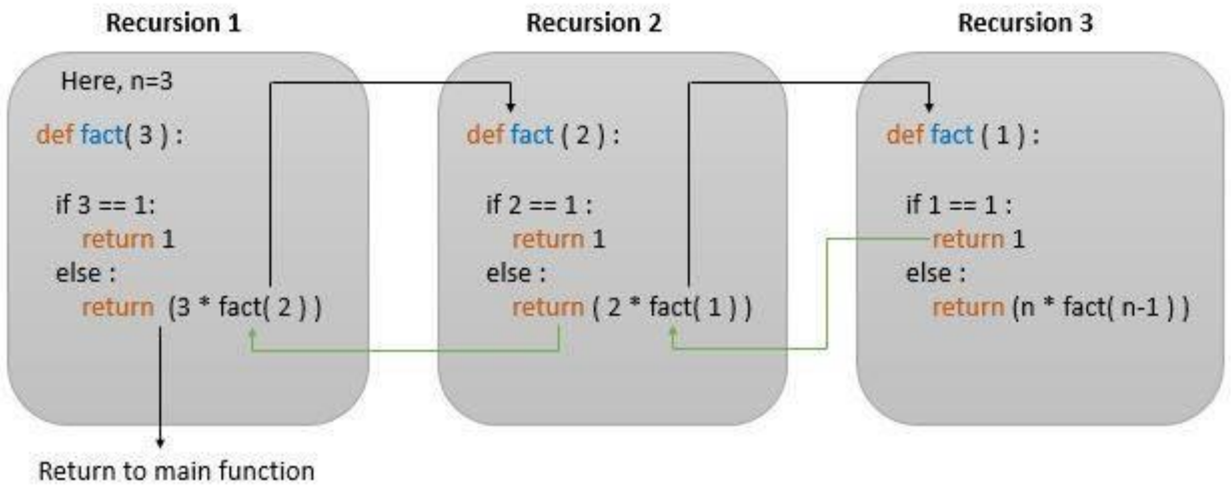
```
3! = 6
```

This program can calculate factorial of any number supplied as the argument.

Explanation of the program

First is a base condition in any recursive function. If the value of n is equal to 1, then the function will return 1 and exit. Till this base condition is met, the function will be iterated.

In the program above, the value of the argument supplied is 3. Hence, the program operates in following way.



When the function is called with the value of n

In recursion 1: Function returns 3 * fact(2). This invokes the function again with the value of n = 2.

In recursion 2: Function checks if n = 1. Since it's False function return 3 * 2 * fact(1). This again invokes the function with the value of n = 1.

In recursion 3: Function checks if n = 1. This returns True making the function to exit and return 1.

Hence after 3 recursions, the final value returned is 3 * 2 * 1 = 6.

Python Anonymous/Lambda Function

In Python, 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 of Lambda Function in python

```
lambda arguments: 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.

Example of Lambda Function in python

Here is an example of lambda function that doubles the input value.

```
# 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. The statement

```
double = lambda x: x * 2
```

is nearly the same as:

```
def double(x):
  return x * 2
```

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −
lambda [arg1 [,arg2,.....argn]]:expression

Following is the example to show how *lambda* form of function works −

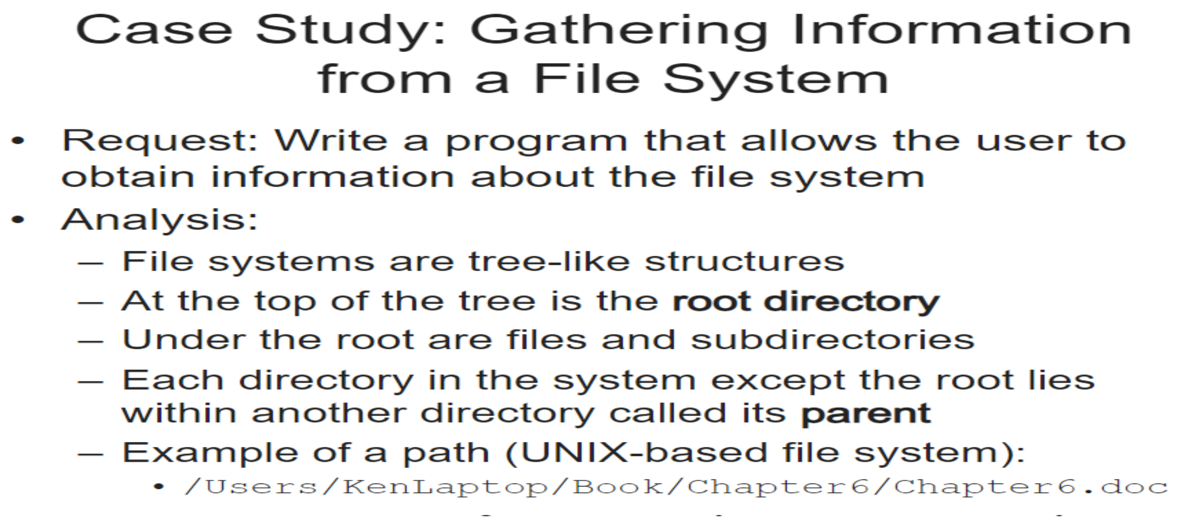
```
#!/usr/bin/python
```

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

When the above code is executed, it produces the following result −

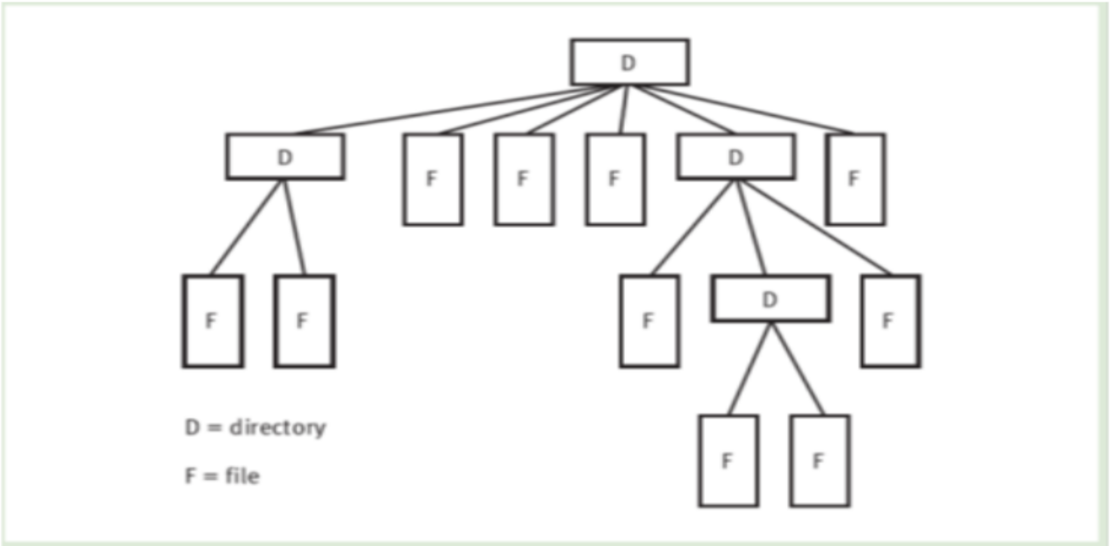
```
Value of total :  30
Value of total :  40
```

Case Study Gathering Information from a File System

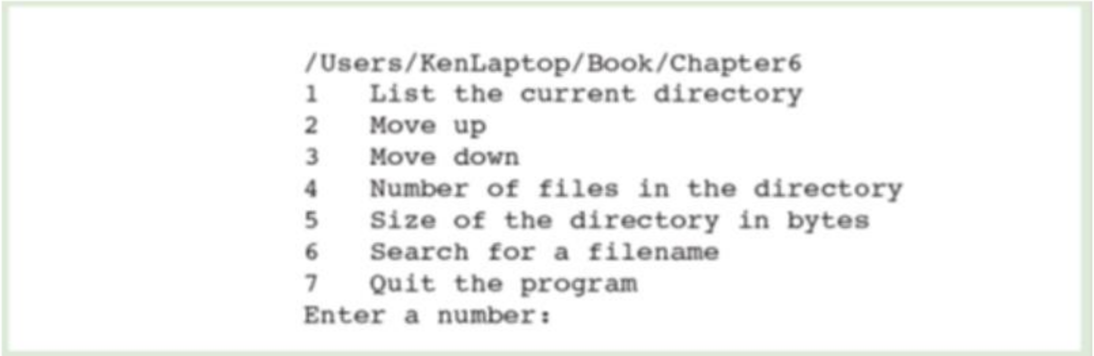


Case Study: Gathering Information from a File System

- Request: Write a program that allows the user to obtain information about the file system
- Analysis:
  - File systems are tree-like structures
  - At the top of the tree is the **root directory**
  - Under the root are files and subdirectories
  - Each directory in the system except the root lies within another directory called its **parent**
  - Example of a path (UNIX-based file system):
    - `/Users/KenLaptop/Book/Chapter6/Chapter6.doc`



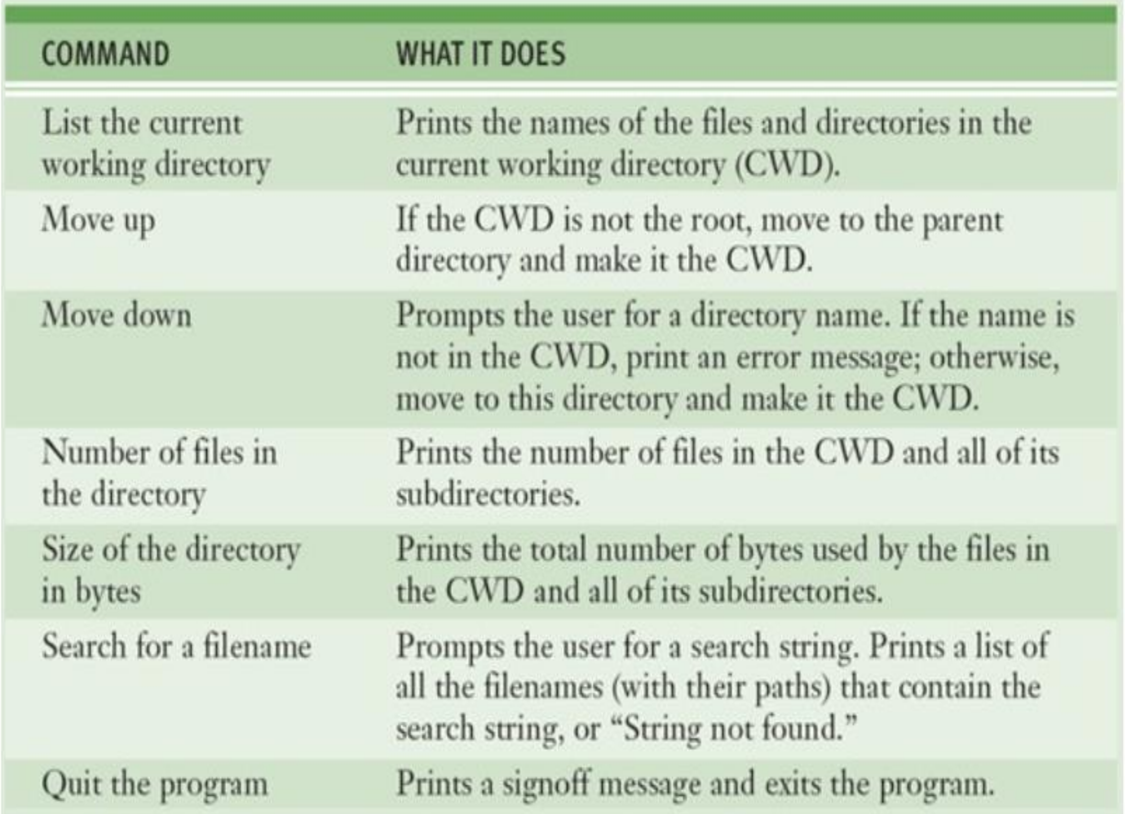
[FIGURE 6.5] The structure of a file system



```
/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:
```

[FIGURE 6.6] The command menu of the **filesys** program

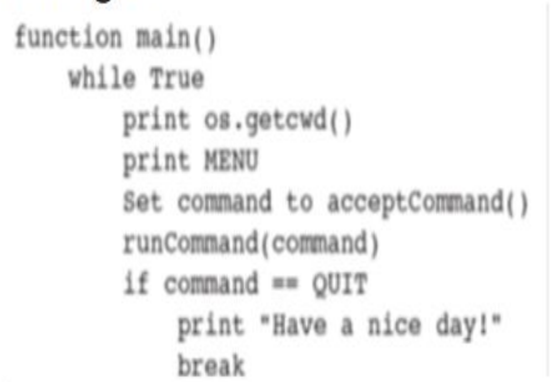
– When user enters a number, program runs command; then, displays CWD and menu again
– An unrecognized command produces an error message



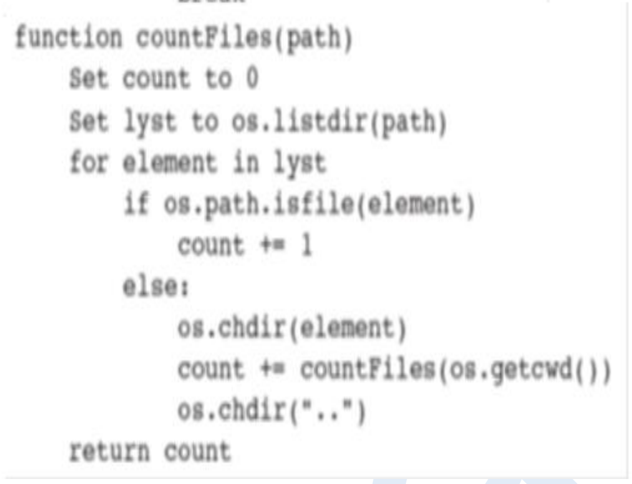
COMMAND	WHAT IT DOES
List the current working directory	Prints the names of the files and directories in the current working directory (CWD).
Move up	If the CWD is not the root, move to the parent directory and make it the CWD.
Move down	Prompts the user for a directory name. If the name is not in the 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.

[TABLE 6.1] The commands in the **filesys** program

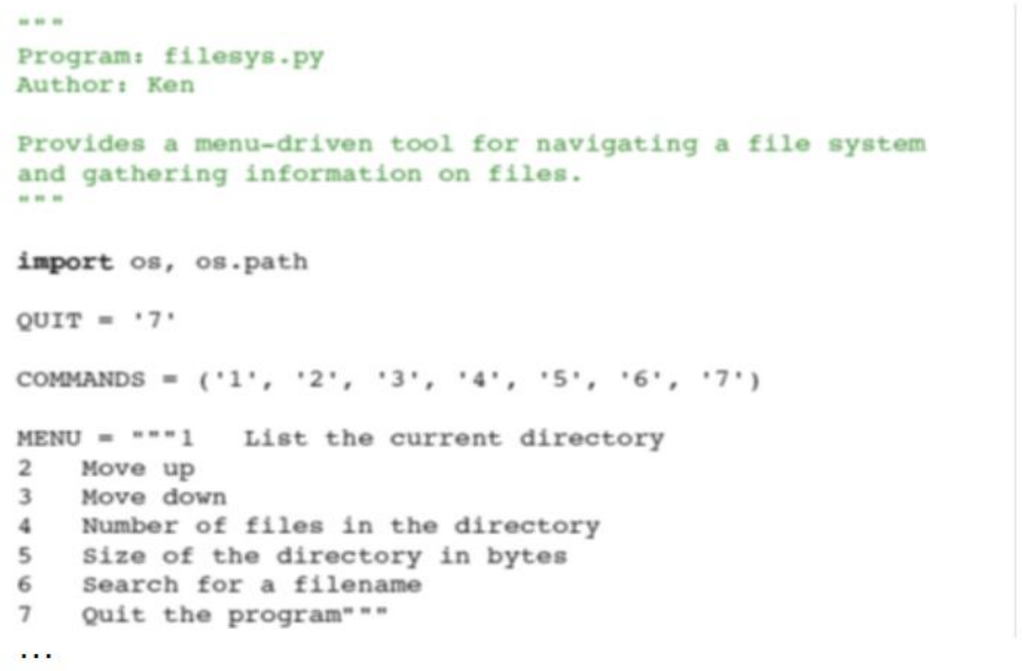
- **Design:**



```
function main()
    while True
        print os.getcwd()
        print MENU
        Set command to acceptCommand()
        runCommand(command)
        if command == QUIT
            print "Have a nice day!"
            break
```



```
function countFiles(path)
    Set count to 0
    Set lyst to 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
```



```
"""
Program: filesys.py
Author: Ken

Provides a menu-driven tool for navigating a file system
and gathering information on files.
"""

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"""
...
```

Managing a Program's Namespace

namespace: A namespace is a system that has a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a Python dictionary.

Name (also called identifier) is simply a name given to objects. Everything in Python is an object. Name is a way to access the underlying object.

For example, when we do the assignment a = 2, 2 is an object stored in memory and a is the name we associate it with. We can get the address (in RAM) of some object through the built-in function id().

```
# Note: You may get different values for the id
a = 2
print('id(2) =', id(2))

print('id(a) =', id(a))
```

Output

```
id(2) = 9302208
id(a) = 9302208
```

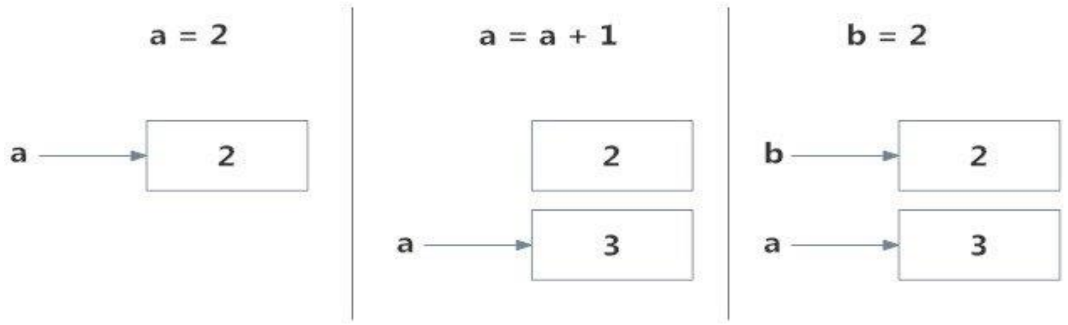
Here, both refer to the same object 2, so they have the same id(). Let's make things a little more interesting.

```
# Note: You may get different values for the id
a = 2
print('id(a) =', id(a))

a = a+1
print('id(a) =', id(a))

print('id(3) =', id(3))

b = 2
print('id(b) =', id(b))
print('id(2) =', id(2))
```

Output

```
id(a) = 9302208
id(a) = 9302240
id(3) = 9302240
id(b) = 9302208
id(2) = 9302208
```

What is happening in the above sequence of steps? Let's use a diagram to explain this:



Memory diagram of variables in Python

Initially, an object 2 is created and the name a is associated with it, when we do a = a+1, a new object 3 is created and now a is associated with this object.

Note that id(a) and id(3) have the same values.

Furthermore, when b = 2 is executed, the new name b gets associated with the previous object 2.

This is efficient as Python does not have to create a new duplicate object. This dynamic nature of name binding makes Python powerful; a name could refer to any type of object.

```
>>> a = 5
>>> a = 'Hello World!'
>>> a = [1,2,3]
```

All these are valid and a will refer to three different types of objects in different instances. Functions are objects too, so a name can refer to them as well.

```
def printHello():
    print("Hello")
a = printHello
a()
```

Output

```
Hello
```

The same name a can refer to a function and we can call the function using this name.

What is a Namespace in Python?

Now that we understand what names are, we can move on to the concept of namespaces.

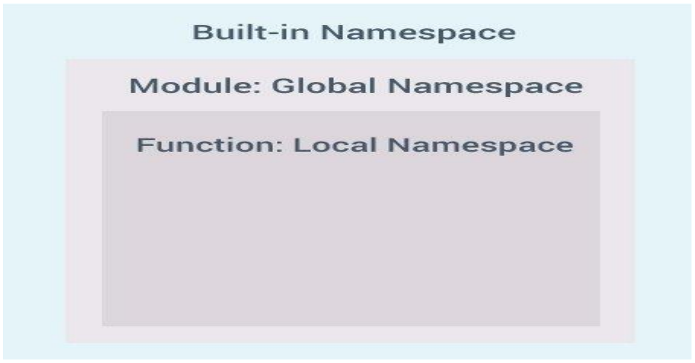
To simply put it, a namespace is a collection of names.

In Python, you can imagine a namespace as a mapping of every name you have defined to corresponding objects.

Different namespaces can co-exist at a given time but are completely isolated.

A namespace containing all the built-in names is created when we start the Python interpreter and exists as long as the interpreter runs.

Modules can have various functions and classes. A local namespace is created when a function is called, which has all the names defined in it. Similar is the case with class. The following diagram may help to clarify this concept.



A diagram of different namespaces in Python

Python Variable Scope

Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play.

A scope is the portion of a program from where a namespace can be accessed directly without any prefix.

At any given moment, there are at least three nested scopes.

1. Scope of the current function which has local names
2. Scope of the module which has global names
3. Outermost scope which has built-in names

When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.

If there is a function inside another function, a new scope is nested inside the local scope.

Example of Scope and Namespace in Python

```
def outer_function():
    b = 20
    def inner_func():
        c = 30
a = 10
```

Here, the variable a is in the global namespace. Variable b is in the local namespace of outer_function() and c is in the nested local namespace of inner_function().

When we are in inner_function(), c is local to us, b is nonlocal and a is global. We can read as well as assign new values to c but can only read b and a from inner_function().

If we try to assign as a value to b, a new variable b is created in the local namespace which is different than the nonlocal b. The same thing happens when we assign a value to a.

However, if we declare a as global, all the reference and assignment go to the global a. Similarly, if we want to rebind the variable b, it must be declared as nonlocal. The following example will further clarify this.

```
def outer_function():
    a = 20
    def inner_function():
        a = 30
        print('a =', a)

    inner_function()
    print('a =', a)

a = 10
outer_function()
print('a =', a)
```

the output of this program is

```
a = 30
a = 20
a = 10
```

Higher-Order Function in Python

A function will be called a Higher Order Function when it contains the other function in the form of the parameter and returns it as an output. We will call the Higher-Order Function to those function which works with the other function. Python also supports the Higher-Order Function.

Properties of the Higher-Order Function

1. The function will work as an instance of the object type.
2. In the Higher-Order Function, we can easily store the function in the form of the variable.
3. In the Higher-Order Function, a function can contain another function as a parameter.
4. In the higher-order function, a function can also return the function.
5. We can store these functions in the data structure in the hash table, lists etc.

Functions as object

In Python, programming language function can be assigned as a variable. Here function will not be called; instead, the reference of the function will be created.

```
# Here is the Python program to define the functions
# In Python, the function can be treated as objects
def call(text):
    return text.upper()

print(call('Hello'))

# function assigning to the variable
yell = call

print(yell('Hello'))
```

The output of the above program will look like, as shown below:

```
HELLO
HELLO
```

In the above example, a function object referenced by a call creates a second name which is, yell.

Pass function as an argument to the other function

Functions are known as Python; therefore, we can pass them as an argument to other functions. The below example shows us how we can create the greet function, which contains the function as an argument.

```
# Python program to create the functions
# function contained the other function as an argument
def Shout(text):
   return text.upper()

def whispered(text):
   return text.lower()

def greets(func):
   # storing the function in a variable
   greeting = func(" function created \ and
   passed as an argument.")
   print(greeting)

greet(Shout)
greet(whisper)
```

The output of the above program will look like, as shown below:

```
FUNCTION CREATED\ AND PASSED AS AN ARGUMENT.
 Function created \ and passed as an argument
```

Returning Function

Functions are known as objects; we can return a function from another function. In the below example, the create_address function returns the address function.

```
# Python program to define the functions
# Functions can return another function

def create_address(z):
   def address(c):
      return z + c

   return address

add_15 = create_address(15)

print(add_15(10))
```

The output of the above program will look like, as shown below:

```
25
```

Decorators

For the higher-order functions, decorators are used in Python. We can modify the behavior of the class or the function by using the decorators. The decorator wraps another function to extend the wrapped function's behavior without modifying the function. Decorators contain the functions as an argument into another function and then called the decorators inside the wrapper function.

The below example shows us how we can write the higher order function in python.

```
def twiced(function):
  return lambda x: function(function(x))
def f(x):
  return x + 3
g = twiced(f)
print g(7)
```

The result of the above program will look like as shown below:

```
13
```

Python Modules

A module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized. We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Example: Type the following and save it as **example.py.**

```
# Python Module example

def add(a, b):
result = a + b
return result
```

Explanation: Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

How to import modules in Python?

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. To import our previously defined module example, we type thefollowing in the Python prompt.

```
>>> import example
```

This does not import the names of the functions defined in example directly in the current symbol table. It only imports the module name example there. Using the module name we can access the function using the dot . operator.

Example:

```
>>> example.add(4,5.5)
9.5
```

Python has lots of standard modules. These files are in the Libdirectory inside the location where you installed Python.

There are various ways to import modules. They are listed below..

- Python import statement
- import with renaming
- Python from...import statement
- import all names

Python import statement:

We can import a module using the import statement and access the definitions inside it using the dotoperator as described above.

Example:

```
import math
print("The value of pi is", math.pi)
```

Output:

```
The value of pi is 3.141592653589793
```

import with renaming:

We can import a module by renaming it. We have renamed the math module as m. This can save ustyping time in some cases.

Note: that the name math is not recognized in our scope. Hence, math.pi is invalid, and m.pi is thecorrect implementation.

Example:

```
import math as m
print("The value of pi is", m.pi)
```

Python from...import statement:

We can import specific names from a module without importing the module as a whole. Here, weimported only the pi attribute from the math module. In such cases, we don't use the dot operator.

Example:1

```
from math import pi print("The value of pi is", pi)
```

Example:2

We can also import multiple attributes as

```
>>> from math import pi, e
>>> pi 3.141592653589793
>>> e 2.718281828459045
```

import all names:
We can import all names(definitions) from a module. we can import all the definitions from the math module. This includes all names visible in our scope except those beginning with an underscore(private definitions).

Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

Example:

```
from math import *
print("The value of pi is", pi)
```

Python packages

Packages are a way of structuring many packages and modules which helps in a well-organized hierarchy of data set, making the directories and modules easy to access. Just like there are different drives and folders in an OS to help us store files, similarly packages help us in storing other sub-packages and modules, so that it can be used by the user when necessary.

The packages in python facilitate the developer with the application development environment by providing a hierarchical directory structure where a package contains sub-packages, modules, and sub-modules. The packages are used to categorize the application level code efficiently.
a directory can contain subdirectories and files, a Python package can have sub-packages and modules.
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

Let's create a package named Employees in your home directory. Consider the following steps.

1. Create a directory with name Employees on path /**home**.

2. Create a python source file with name ITEmployees.py on the path /**home**/**Employees**.

ITEmployees.py

```
 def getITNames():
     List = ["John", "David", "Nick",    "Martin"]
return List;
```

3. Similarly, create one more python file with name BPOEmployees.py and create a function getBPONames().

4. Now, the directory Employees which we have created in the first step contains two python modules. To make this directory a package, we need to include one more file here, that is __init__.py which contains the import statements of the modules defined in this directory.

__init__.py

```
from ITEmployees import getITNames
from BPOEmployees import getBPONames
```

5. Now, the directory **Employees** has become the package containing two python modules. Here we must notice that we must have to create __init__.py inside a directory to convert this directory to a package.

6. To use the modules defined inside the package Employees, we must have to import this in our python source file. Let's create a simple python source file at our home directory (/home) which uses the modules defined in this package.

Test.py

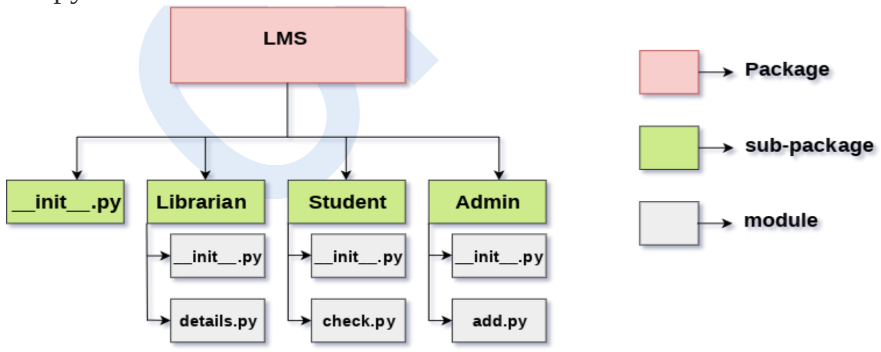
```
import Employees
print(Employees.getNames())
```

Output:

```
['John', 'David', 'Nick', 'Martin']
```

We can have sub-packages inside the packages. We can nest the packages up to any level depending upon the application requirements.

The following image shows the directory structure of an application Library management system which contains three sub-packages as Admin, Librarian, and Student. The sub-packages contain the python modules.



----------------------******---------------------------------