UNIT-3

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.

Lists:

- List is used to store the group of values & we can manipulate them, in list the values are stores in index format starts with 0.
- List is mutable object so we can do the manipulations.
- A python list is enclosed between square ([]) brackets.
- In list insertion order is preserved it means in which order we inserted element same order output is printed.
- A list can be composed by storing a sequence of different type of values separated by commas.
- The list contains forward indexing & backward indexing.

Syntax: <list_name> = [value1,value2,value3,...,valuen]

Example:

```
data1=[1,2,3,4]              # list of integers
data2=['x','y','z']          # list of String
data3=[12.5,11.6]            # list of floats
data4=[]                     # empty list
data5=['TEC',10,56.4,'a']    # list with mixed data types
```

Accessing List data:

- The list items are accessed by **referring to the index number**.
- **Negative indexing** means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.
- A **range of indexes** can be specified by specifying where to start and where to end the range. When specifying a range, the return value will be a new list with the specified items.
- **Note:** The search will start at index 2 (included) and end at index 5 (not included).
- By leaving out the start value, the range will start at the first item.
- By leaving out the end value, the range will go on to the end of the list.

Examples:

1.
```
x=["apple", "banana", "cherry"]
print(x[1])
```
Output: banana

2.
```
x= ["apple", "banana", "cherry"]
print(x[-1])
```
Output: cherry

3.
```
x= ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(x[2:5])
```
Output: ['cherry', 'orange', 'kiwi']

4.
```
x = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(x[:4])
```
Output: ['apple', 'banana', 'cherry', 'orange']

5.
```
x= ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(x[2:])
```
Output: ['cherry', 'orange', 'kiwi', 'melon', 'mango']

- ***Range of Negative Indexes***: Specify negative indexes if you want to start the search from the end of the list.

Example:
```
x = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(x[-4:-1])
```
Output: ['orange', 'kiwi', 'melon']

- ***Check if Item Exists***: To determine if a specified item is present in a list use the in keyword.

Example: Check if "apple" is present in the list:

```
fruits= ["apple", "banana", "cherry"]
if "apple" in fruits:
    print("Yes, 'apple' is in the fruits list")
```
Output: Yes, 'apple' is in the fruits list

Change Item Value:

- To change the value of a specific item, refer to the index number.

Example: Change the second item:

```
x = ["apple", "banana", "cherry"]
x[1] = "blackcurrant"
print(x)
```

Output: ['apple', 'blackcurrant', 'cherry']

- To insert more than one item, create a list with the new values, and specify the index number where you want the new values to be inserted.

Example: Change the second value by replacing it with two new values:

```
x = ["apple", "banana", "cherry"]
x[1] = ["blackcurrant", "watermelon"]
print(x)
```

Output: ['apple', ['blackcurrant', 'watermelon'], 'cherry']

Change a Range of Item Values

- To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.

Example: Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon":

```
x = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
x[1:3] = ["blackcurrant", "watermelon"]
print(x)
```

Output: ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']

Python List/Array Methods: Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

1. List append():

- The append() method appends an element to the end of the list.

Syntax: list.append(elmnt)

Here,

elmnt Required. An element of any type (string, number, object etc.)

Example: 1 Add an element to the fruits list.

```
fruits = ['apple', 'banana', 'cherry']
fruits.append("orange")
```

Output: ['apple', 'banana', 'cherry', 'orange']

Example: 2 Add a list to a list.

```
a = ["apple", "banana", "cherry"]
b = ["Ford", "BMW", "Volvo"]
a.append(b)
```

Output: ['apple', 'banana', 'cherry', ["Ford", "BMW", "Volvo"]]

2. List clear():

- The clear() method removes all the elements from a list.

Syntax: list.clear()

Example: Remove all elements from the fruits list.

```
fruits = ['apple', 'banana', 'cherry', 'orange']
fruits.clear()
```

Output: []

3. List copy():

- The copy() method returns a copy of the specified list.

Syntax: list.copy()

Example: Copy the fruits list.

```
fruits = ['apple', 'banana', 'cherry', 'orange']
x = fruits.copy()
```

Output: ['apple', 'banana', 'cherry']

4. List count()

- The count() method returns the number of elements with the specified value.

Syntax: list.count(value)

Here,

value = Required. Any type (string, number, list, tuple, etc.). The value to search for.

Example-1: Return the number of times the value "cherry" appears in the fruits list.

```
fruits = ['apple', 'banana', 'cherry']
x = fruits.count("cherry")
```

Output: 1

Example-2: Return the number of times the value 9 appears in the list.

```
points = [1, 4, 2, 9, 7, 8, 9, 3, 1]
x = points.count(9)
```

Output: 2

5. List extend(): The extend() method adds the specified list elements (or any iterable) to the end of the current list.

Syntax: list.extend(iterable)

Here,

iterable	Required. Any iterable (list, set, tuple, etc.)

Example – 1: Add the elements of cars to the fruits list.

```
fruits = ['apple', 'banana', 'cherry']
cars = ['Ford', 'BMW', 'Volvo']
fruits.extend(cars)
```

Output: ['apple', 'banana', 'cherry', 'Ford', 'BMW', 'Volvo']

Example – 2: Add a tuple to the fruits list.

```
fruits = ['apple', 'banana', 'cherry']
points = (1, 4, 5, 9)
fruits.extend(points)
```

Output: ['apple', 'banana', 'cherry', 1, 4, 5, 9]

6. List index(): The index() method returns the position at the first occurrence of the specified value.

Note: The index() method only returns the first occurrence of the value.

Syntax: list.index(elmnt)

Here,

elmnt = Required. Any type (string, number, list, etc.). The element to search for.

Example – 1: What is the position of the value "cherry".

```
fruits = ['apple', 'banana', 'cherry']
x = fruits.index("cherry")
```

Output: 2

Example – 2:What is the position of the value 32.

```
fruits = [4, 55, 64, 32, 16, 32]
x = fruits.index(32)
```

Output: 3

7. List insert():

- To insert a new list item, without replacing any of the existing values, we can use the insert() method. The insert() method inserts an item at the specified index:

Syntax: list.insert(pos, elmnt)

Here,

pos = Required. A number specifying in which position to insert the value

elmnt =Required. An element of any type (string, number, object etc.)

Example: Insert "watermelon" as the third item.

```
x = ["apple", "banana", "cherry"]
x.insert(2, "watermelon")
print(x)
```

Output: ['apple', 'banana', 'watermelon', 'cherry']

8. List pop():

- The pop() method removes the element at the specified position.
- Note: The pop() method returns removed value.

Syntax: list.pop(pos)

Here,

Pos = Optional. A number specifying the position of the element you want to remove, default value is -1, which returns the last item

Example - 1: Remove the second element of the fruit list:

```
fruits = ['apple', 'banana', 'cherry']
fruits.pop(1)
```

Output: ['apple', 'cherry']

Example – 2: Return the removed element:

```
fruits = ['apple', 'banana', 'cherry']
x = fruits.pop(1)
```

Output: banana

9. List remove() :

- The remove() method removes the first occurrence of the element with the specified value.

Syntax: list.remove(elmnt)

Here,

elmnt Required. Any type (string, number, list etc.) The element you want to remove

Example: Remove the "banana" element of the fruit list.

```
fruits = ['apple', 'banana', 'cherry']
fruits.remove("banana")
```

Output: ['apple', 'cherry']

10. List reverse():

- The reverse() method reverses the sorting order of the elements.

Syntax: list.reverse()

Example:1 Reverse the order of the fruit list.

```
fruits = ['apple', 'banana', 'cherry']
fruits.reverse()
```

Output: ['cherry', 'banana', 'apple']

Example:2

```
List1=["banana", "Orange", "Kiwi", "cherry"]
List1.reverse()
print(List1)
```

Output: ['cherry', 'Kiwi', 'Orange', 'banana']

Loop Through a List:

- We can loop through the list items by using a for loop.
- Print all items in the list, one by one:
- Use the range() and len() functions to print all items by referring to their index number:

Example – 1:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

Output:

```
apple
banana
cherry
```

Example-2:

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])
```

Output:

```
apple
banana
cherry
```

- ***Using a While Loop***: Use the len() function to determine the length of the list, then start at 0 and loop your way through the list items by refering to their indexes. Remember to increase the index by 1 after each iteration.

Example – 1:

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
```

Output:

```
apple
banana
cherry
```

- ***Looping Using List Comprehensive:*** It offers the shortest syntax for looping through lists.

Example – 1:

```
thislist = ["apple", "banana", "cherry"]
[print x for x in thislist]
```

Output:

```
apple
banana
cherry
```

Sort Lists: Sort List Alphanumerically. List objects have a sort() method that will sort the list alphanumerically, ascending, by default.To sort descending, use the keyword argument reverse = True.

Example -1:

```
List1= ["orange", "mango", "kiwi", "pineapple", "banana"]
List1.sort()
print(List1)
```

Output:

```
['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

Example – 2:

```
List1 = [100, 50, 65, 82, 23]
List1.sort()
print(List1)
```

Output:

```
[23, 50, 65, 82, 100]
```

Example – 3: Sort the list descending

```
List1= ["orange", "mango", "kiwi", "pineapple", "banana"]
List1.sort(reverse = True)
print(List1)
```

Output:

```
['pineapple', 'orange', 'mango', 'kiwi', 'banana']
```

- **Customize Sort Function:** we can also customize you own function by using the keyword argument key = function. The function will return a number that will be used to sort the list (the lowest number first):

Example -1: Sort the list based on how close the number is to 50:

```
def myfunc(n):
  return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)
```

Output:
[50, 65, 23, 82, 100]

Note: By default the sort() method is case sensitive, resulting in all capital letters being sorted after lower case letters:

Example – 1:

```
List1 = ["banana", "Orange", "Kiwi", "cherry"]
List1.sort()
print(List1)
```

Output: `['Kiwi', 'Orange', 'banana', 'cherry']`

Case Insensitive Sort: we can use built-in functions as key functions when sorting a list. So if you want a case-insensitive sort function, use str.lower as a key function.

Example – 1:

```
List1 = ["banana", "Orange", "Kiwi", "cherry"]
List1.sort(key = str.lower)
print(List1)
```

Output: `['banana', 'cherry', 'Kiwi', 'Orange']`

Dictionaries

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is unordered, it means that the items does not have a defined order, you cannot refer to an item by using an index.
- It is changeable, means that we can change, add or remove items after the dictionary has been created.
- It does not allow duplicates means cannot have two items with the same key.
- Dictionaries are written with curly brackets, and have keys and values.
- It can be referred by using the key name.
- The values in dictionary items can be of any data type.

Example-1: Create and print a dictionary.

```
dict1 = {
 "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(dict1)
```

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

Example-2: Print the "brand" value of the dictionary

```
dict1 = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(dict1["brand"])
```

Output: Ford

Example -3: Duplicate values will overwrite existing values

```
dict1 = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(dict1)
```

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 2020}

Example – 4: String, int, boolean, and list data types

```
dict1 = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
print(dict1)
```

Output: {'brand': 'Ford', 'electric': False, 'year': 1964, 'colors': ['red', 'white', 'blue']}

Python Dictionary Methods: Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

1. Dictionary clear():

- The clear() method removes all the elements from a dictionary.

Syntax: dictionary.clear()

Example: Remove all elements from the car list.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
car.clear()
print(car)
```

Output: {}

2. Dictionary copy():

- You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.
- The copy() method returns a copy of the specified dictionary.

Syntax: dictionary.copy()

Example: Copy the car dictionary.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.copy()
print(x)
```

Output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

- Another way to make a copy is to use the built-in function dict().

Example: Make a copy of a dictionary with the dict() function.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

Output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

3. Dictionary fromkeys():

- The fromkeys() method returns a dictionary with the specified keys and the specified value.

Syntax: dict.fromkeys(keys, value)

Here,

keys Required. An iterable specifying the keys of the new dictionary

value Optional. The value for all keys. Default value is None

Example: 1 Create a dictionary with 3 keys, all with the value 0.

```
x = ('key1', 'key2', 'key3')
y = 0
thisdict = dict.fromkeys(x, y)
print(thisdict)
```

Output:

```
{'key1': 0, 'key2': 0, 'key3': 0}
```

Example: 2 Same example as above, but without specifying the value.

```
x = ('key1', 'key2', 'key3')
thisdict = dict.fromkeys(x)
print(thisdict)
```

Output: {'key1': None, 'key2': None, 'key3': None}

4. Dictionary get():

- The get() method returns the value of the item with the specified key.

Syntax: dictionary.get(keyname, value)

Here,

keyname Required. The keyname of the item you want to return the value from

value Optional. A value to return if the specified key does not exist. Default value None

Example-1: Get the value of the "model" item.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.get("model")
print(x)
```

Output: Mustang

Example-2: Try to return the value of an item that do not exist.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.get("price", 15000)
print(x)
```

Output: 15000

5. Dictionary items():

- The items() method returns a view object. The view object contains the key-value pairs of the dictionary, as tuples in a list.

- The view object will reflect any changes done to the dictionary.

Syntax: dictionary.items()

Example-1: Return the dictionary's key-value pairs:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.items()
print(x)
```

Output:
```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
```

Example-2: When an item in the dictionary changes value, the view object also gets updated.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.items()
car["year"] = 2018
print(x)
```

Output:
```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2018)])
```

6. Dictionary keys():

- The keys() method returns a view object. The view object contains the keys of the dictionary, as a list.
- The view object will reflect any changes done to the dictionary.

Syntax: dictionary.keys()

Example – 1: Return the keys.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.keys()
```

```
print(x)
```

Output:
```
dict_keys(['brand', 'model', 'year'])
```

Example – 2: When an item is added in the dictionary, the view object also gets updated.

```
car = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
x = car.keys()
car["color"] = "white"
print(x)
```

Output:
```
dict_keys(['brand', 'model', 'year', 'color'])
```

7. Dictionary pop():

- The pop() method removes the specified item from the dictionary.
- The value of the removed item is the return value of the pop() method.

Syntax: dictionary.pop(keyname, defaultvalue)

Here,

keyname	Required. The keyname of the item you want to remove
defaultvalue	Optional. A value to return if the specified key do not exist. If this parameter is not specified, and the no item with the specified key is found, an error is raised.

Example: 1 **Remove** "model" from the dictionary.

```
car = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
car.pop("model")
print(car)
```

Output:
```
{'brand': 'Ford', 'year': 1964}
```

Example: 2 The value of the removed item is the return value of the pop() method.

```
car = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
}
```

```
x = car.pop("model")
print(x)
```

Output: `Mustang`

8. Dictionary popitem():

- The popitem() method removes the item that was last inserted into the dictionary. In versions before 3.7, the popitem() method removes a random item.
- The removed item is the return value of the popitem() method, as a tuple.

Syntax: dictionary.popitem()

Example: 1 **Remove** the last item from the dictionary.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
car.popitem()
print(car)
```

Output: `{'brand': 'Ford', 'model': 'Mustang'}`

Example: 2 The removed item is the return value of the pop() method.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.popitem()
print(x)
```

Output: `('year', 1964)`

9. Dictionary setdefault():

- The setdefault() method returns the value of the item with the specified key.
- If the key does not exist, insert the key, with the specified value.

Syntax: dictionary.setdefault(keyname, value)

Here:

keyname Required. The keyname of the item you want to return the value from

value Optional. If the key exist, this parameter has no effect.
If the key does not exist, this value becomes the key's value
Default value None.

Example: 1 Get the value of the "model" item.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.setdefault("model", "Bronco")
print(x)
```

Output: Mustang

Example: 2 Get the value of the "color" item, if the "color" item does not exist, insert "color" with the value "white"

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.setdefault("color", "white")
print(x)
```

Output: White

10. Dictionary update():

- The update() method inserts the specified items to the dictionary.
- The specified items can be a dictionary, or an iterable object with key value pairs.

Syntax: dictionary.update(iterable)

Here,

iterable A dictionary or an iterable object with key value pairs, that will be inserted to the dictionary.

Example: Insert an item to the dictionary.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
car.update({"color": "White"})
```

```
print(car)
```

Output:
```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'White'}
```

11. Dictionary values():

- The values() method returns a view object. The view object contains the values of the dictionary, as a list.
- The view object will reflect any changes done to the dictionary, see example below.

Syntax: dictionary.values()

Example: 1 Return the values.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.values()
print(x)
```

Output:
```
dict_values(['Ford', 'Mustang', 1964])
```

Example: 2 when a value is changed in the dictionary, the view object also gets updated.

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = car.values()
car["year"] = 2018
print(x)
```

Output:
```
dict_values(['Ford', 'Mustang', 2018])
```

12. del

- The del keyword removes the item with the specified key name.

Example: 1

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

Output:

```
{'brand': 'Ford', 'year': 1964}
```

Example: 2 The del keyword can also delete the dictionary completely.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict)          #this will cause an error because "thisdict" no longer exists.
```

Output:

```
Traceback (most recent call last):
  File "demo_dictionary_del3.py", line 7, in <module>
    print(thisdict) #this will cause an error because "thisdict" no longer exists.
NameError: name 'thisdict' is not defined
```

Check if Key Exists:

- To determine if a specified key is present in a dictionary use the **in** keyword.

Example:1 Check if "model" is present in the dictionary.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Output:

```
Yes, 'model' is one of the keys in the thisdict dictionary
```

Dictionary Length:

- To determine how many items a dictionary has, use the len() function:

Example: Print the number of items in the dictionary.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(len(thisdict))
```

Output: 3

Loop Through a Dictionary:

- You can loop through a dictionary by using a for loop.

- When looping through a dictionary, the return values are the keys of the dictionary, but there are methods to return the values as well.

Example: 1 Print all key names in the dictionary, one by one:

```
thisdict =      {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

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

Output:

```
brand
model
year
```

Example: 2 Print all values in the dictionary, one by one.

```
for x in thisdict:
  print(thisdict[x])
```

Output:
Ford
Mustang
1964

Example: 3 You can also use the values() method to return values of a dictionary.

```
for x in thisdict.values():
  print(x)
```

Output:
Ford
Mustang
1964

Example: 4 You can use the keys() method to return the keys of a dictionary.

```
for x in thisdict.keys():
  print(x)
```

Output:
brand
model
year

Example: 5 Loop through both keys and values, by using the items() method.

```
for x, y in thisdict.items():
  print(x, y)
```

Output:
brand Ford
model Mustang
year 1964

Nested Dictionaries:

- A dictionary can contain dictionaries, this is called nested dictionaries.

Example: 1 Create a dictionary that contain three dictionaries.

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

Output:

```
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007},
'child3': {'name': 'Linus', 'year': 2011}}
```

Example: 2 Create three dictionaries, then create one dictionary that will contain the other three dictionaries.

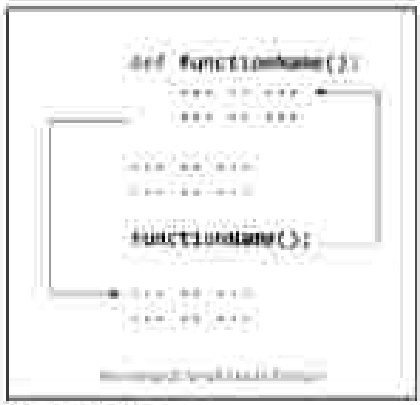
```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}
myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

Output:

```
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007},
'child3': {'name': 'Linus', 'year': 2011}}
```

Python Functions

- In Python, a function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- It avoids repetition and makes the code reusable.



Syntax of Function:

```
def function_name(parameters):
    """docstring"""
    statement(s)
    return
```

Above shown is a function definition that consists of the following components.

- 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.

Example of a function:

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

To call a function in python:

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.

```
>>> greet('Tirumala')
Hello, Tirumala. Good morning!
```

Note: Try running the above code in the Python program with the function definition to see the output.

Docstrings:

- The first string after the function header is called the docstring and is short for documentation string. It is briefly used to explain what a function does.
- Python docstrings are the string literals that appear after the definition of a method, class, or module also.
- In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines.
- We can access these docstrings using the **__doc__** attribute.

Example: 1

```
>>> print(greet.__doc__)

    This function greets to
    the person passed in as
    a parameter
```

Example 2:

```
def square(n):
    '''Takes in a number n, returns the square of n'''
    return n**2

print(square.__doc__)
```

Output:

```
Takes in a number n, returns the square of n
```

Example 3: Docstrings for the built-in print() function

```
print(print.__doc__)
```

Output:

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
Prints the values to a stream, or to sys.stdout by default.
```

Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.

Docstrings for Python Modules:

- The docstrings for Python Modules should list all the available classes, functions, objects and exceptions that are imported when the module is imported.
- They should also have a one-line summary for each item.
- They are written at the beginning of the Python file.

Example 4: docstrings for the builtin module in Python called pickle.

```
import pickle
print(pickle.__doc__)
```

Docstrings for Python Classes:

- The docstrings for classes should summarize its behavior and list the public methods and instance variables.
- The subclasses, constructors, and methods should each have their own docstrings.

Example 5: Docstrings for Python class.

```
class Person:
    """
    A class to represent a person.
    ...
    Attributes
    ----------
    name : str
        first name of the person
    age : int
        age of the person
    """
Methods
----------
    info(additional=""):
        Prints the person's name and age.
    """
```

```
    def info(self, additional=""):
    """
    Prints the person's name and age.
    If the argument 'additional' is passed, then it is appended after the main info.
```

Output:

```
>>> print(Person.__doc__)
```

Using the help() Function for Docstrings

- We can also use the help() function to read the docstrings associated with various objects.
- Here, we can see that the help() function retrieves the docstrings of the Person class along with the methods associated with that class.

Example 6: Read Docstrings with the help() function

```
>>> help(Person)
```

The return statement:

- The return statement is used to exit a function and go back to the place from where it was called.
- This statement can contain an expression that gets evaluated and the value is returned.
- If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

Syntax of return: return [expression_list]

Example:1

```
>>> print(greet("May"))
Hello, May. Good morning!
None
```

Here, None is the returned value since greet() directly prints the name and no return statement is used.

Example: 2

```
def absolute_value(num):
    """This function returns the absolute value of the entered number"""
    if num >= 0:
        return num
    else:
        return -num
print(absolute_value(2))
print(absolute_value(-4))
```

Output:

```
2
4
```

Types of Functions:

Basically, we can divide functions into the following two types:

1. Built-in functions - Python has several functions that are readily available for use. These functions are called built-in functions.

Examples:

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

Note: Refer this link https://www.programiz.com/python-programming/methods/built-in/abs for detailed explanation of each function.

2. User-defined functions -

- Functions that we define ourselves to do certain specific task are referred as user-defined functions.
- If we use functions written by others in the form of library, it can be termed as library functions.
- All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

Advantages of user-defined functions:

- User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmars working on large project can divide the workload by making different functions.

Example:

```
def add_numbers(x,y):
  sum = x + y
  return sum

num1 = 5
num2 = 6
print("The sum is", add_numbers(num1, num2))
```

Output:

```
The sum is 11
```

Python Function Arguments:

- In Python, you can define a function that takes variable number of arguments.

Example:

```
def greet(name, msg):
    """This function greets to
    the person with the provided message"""
    print("Hello", name + ', ' + msg)

greet("Tirumala", "Good morning!")
```

Output:

Hello Tirumala, Good morning!

- Here, the function greet() has two parameters. Since we have called this function with two arguments, it runs smoothly and we do not get any error.

- If we call it with a different number of arguments, the interpreter will show an error message.
- If we call to this function with one and no arguments along with their respective error messages.

```
>>> greet("Monica")    # only one argument
TypeError: greet() missing 1 required positional argument: 'msg'

>>> greet()    # no arguments
TypeError: greet() missing 2 required positional arguments: 'name' and 'msg'
```

- Three different forms of Function Arguments are there. they are below.

1. Default Arguments
2. Keyword Arguments
3. Arbitrary Arguments/Variable Length Arguments.

1. Default Arguments:

Function arguments can have default values. We can provide a default value to an argument by using the assignment operator (=).

Example:

```
def greet(name, msg="Good morning!"):
   print("Hello", name + ', ' + msg)

greet("Tirumala")
greet("CSE", "Welcome to Technical Wing.")
```

Output:

Hello Tirumala, Good morning!
Hello CSE, Welcome to Technical Wing.

Explanation:

- The parameter **name** does not have a default value and is required (mandatory) during a call.
- The parameter **msg** has a default value of "**Good morning!**". So, it is optional during a call. If a value is provided, it will overwrite the default value.
- Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.
- This means to say, non-default arguments cannot follow default arguments.

  Example:

```
def greet(msg = "Good morning!", name):
```

  Output: Syntax Error: non-default argument follows default argument.

2. Keyword Arguments:

- Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed.
- Following calls to the above function are all valid and produce the same result.

```
# 2 keyword arguments
greet(name = "TEC",msg = "Good Morning")

# 2 keyword arguments (out of order)
greet(msg = "Good Morning",name = "TEC")

1 positional, 1 keyword argument
greet("TEC", msg = "Good Morning")
```

Output: Hello TEC, Good Morning

- Having a positional argument after keyword arguments will result in errors.

  Example: greet(name="TEC","Good Morning")

  Output: SyntaxError: non-keyword arg after keyword arg

3. Arbitrary Arguments - 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 an arbitrary number of arguments.
- In the function definition, we use an asterisk (*) before the parameter name to denote this kind of argument.

Example:

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

greet("Monica", "Lakshmi", "Sravan", "Jasmin")
```

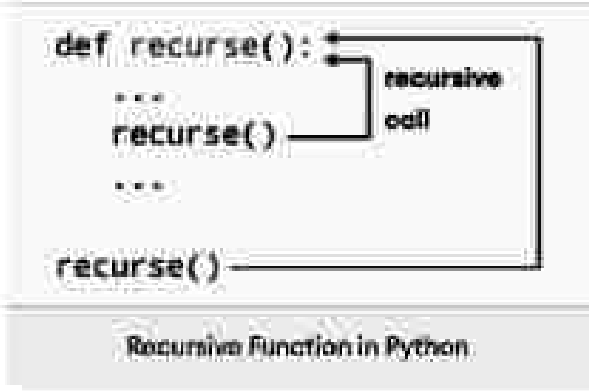
Output:

Hello Monica
Hello Lakshmi
Hello Sravan
Hello Jasmin

Explanation: 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.

Recursive Function:

- A function that calls itself is known as Recursive Function.
- A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.
- The following image shows the working of a recursive function called *recurse*.



Recursive Function in Python

Example:

- Factorial of a number is the product of all the integers from 1 to that number.
- the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

```
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))

num = 3
print("The factorial of", num, "is", factorial(num))
```

Output:

The factorial of 3 is 6

This recursive call can be explained in the following steps.

```
factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call
```

Advantages of Recursion:

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion:

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

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.
- 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.

Syntax of Lambda Function: lambda arguments: expression

Example:

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

print(double(5))
```

Output:
10

Use of Lambda Function in python:

- We use lambda functions when we require a nameless function for a short period of time.
- In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

Example use with filter(): The filter() function in Python takes in a function and a list as arguments.

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)
```

Output:
[4, 6, 8, 12]

Example use with map(): The map() function in Python takes in a function and a list.

```
# Program to double each item in a list using map()
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)
```

Output: [2, 10, 8, 12, 16, 22, 6, 24]

Scope and Lifetime of variables:

- Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.
- The lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Example:

```
def my_fun():
        x = 10
        print("Value inside function:",x)

x = 20
my_fun()
print("Value outside function:",x)
```

Output:

Value inside function: 10

Value outside function: 20

Explanation:

- Here, we can see that the value of x is 20 initially. Even though the function my_fun() changed the value of x to 10, it did not affect the value outside the function.
- This is because the variable x inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with different scopes.

Python Global, Local and Nonlocal variables

1. **Global Variables:** In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Example 1: Create a Global Variable

```
x = "global"
def fun():
    print("x inside:", x)
fun()
print("x outside:", x)
```

Output:

x inside: global

x outside: global

- **What if you want to change the value of x inside a function?**

Example 2:

```
x = "global"
def fun():
    x = x * 2
    print(x)

fun()
```

Output:

UnboundLocalError: local variable 'x' referenced before assignment

Explanation: The output shows an error because Python treats x as a local variable and x is also not defined inside fun(). To make this work, we use the global keyword.

Example 3: Changing Global Variable from inside a Function using global

```
c = 0          # global variable
def add():
    global c
    c = c + 2      # increment by 2
    print("Inside add():", c)

print("Before In main:", c)
add()
print("In main:", c)
```

Output:

```
Before In main: 0
Inside add(): 2
In main: 2
```

2. Local Variables: A variable declared inside the function's body or in the local scope is known as a local variable.

Example 1: Accessing local variable outside the scope

```
def fun():
    y = "local"
fun()
print(y)
```

Output:

NameError: name 'y' is not defined

Explanation: The output shows an error because we are trying to access a local variable y in a global scope whereas the local variable only works inside fun() or local scope.

Example 2: Create a Local Variable

```
def fun():
    y = "local"
    print(y)

fun()
```

Output:
local

Example 3: Global and local variables.

```
x = "global "

def fun():
    global x
    y = "local"
    x = x * 2
    print(x)
    print(y)

fun()
```

Output:
global global
local

Example 4: Global variable and Local variable with same name

```
x = 5
def fun():
    x = 10
    print("local x:", x)

fun()
print("global x:", x)
```

Output:
local x: 10
global x: 5

Explanation: In the above code, we used the same name x for both global variable and local variable. We get a different result when we print the same variable because the variable is declared in both scopes, i.e. the local scope inside foo() and global scope outside foo().

Python Modules

- Modules refer to a file containing Python statements and definitions.We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

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 the following 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. **Refer standard modules in UNIT-1 Material**. These files are in the Lib directory inside the location where you installed Python.

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

1. Python import statement
2. import with renaming
3. Python from...import statement
4. import all names

1. Python import statement:

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

Example:

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

Output:

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

2. import with renaming:

- We can import a module by renaming it. We have renamed the math module as m. This can save us typing time in some cases.
- Note that the name math is not recognized in our scope. Hence, math.pi is invalid, and m.pi is the correct implementation.

Example:

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

3. Python from...import statement:

- We can import specific names from a module without importing the module as a whole. Here, we imported 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
```

4. 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)

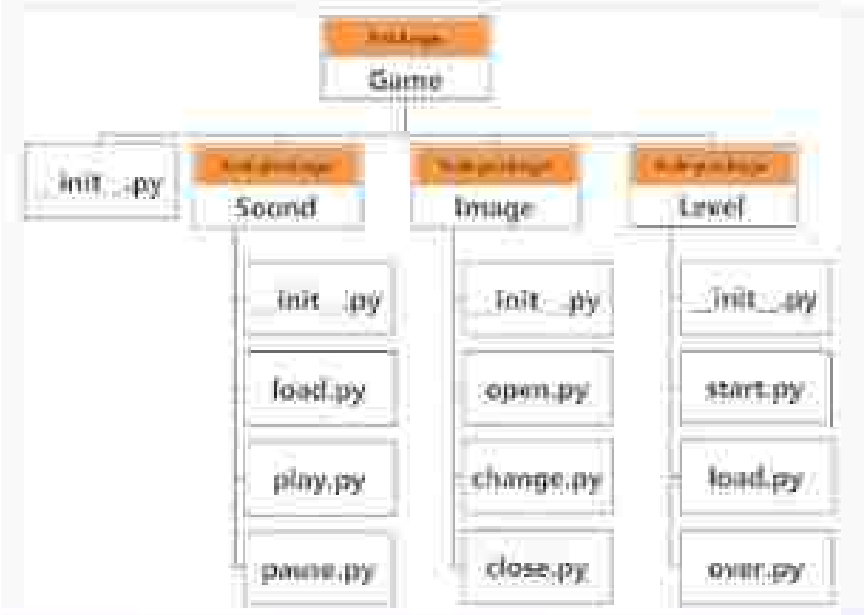
- Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

Example:

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

Python Package

- We don't usually store all of our files on our computer in the same location. We use a well-organized hierarchy of directories for easier access.
- Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.
- As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.
- Similarly, as 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.
- Here is an example. Suppose we are developing a game. One possible organization of packages and modules could be as shown in the figure below.



Package Module Structure in Python Programming

Importing module from a package:

- We can import modules from packages using the dot (.) operator. For example, if we want to import the start module in the above example, it can be 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)
```

- If this construct seems lengthy, we can import the module without the package prefix as follows.

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

- We can now call the function simply as follows

```
start.select_difficulty(2)
```

- Another way of importing just the required function (or class or variable) from a module within a package would be as follows.

```
from Game.Level.start import select_difficulty
```

- Now we can directly call this function.

```
select_difficulty(2)
```

- Although easier, this method is not recommended. Using the full namespace avoids confusion and prevents two same identifier names from colliding.

- While importing packages, Python looks in the list of directories defined in sys.path, similar as for module search path.