UNIT- III

Prepared By: Bhavna Barad

Agenda of Unit - III

No.	Topics(Functions & File Handling)
1.	**Functions:** Defining and calling functions
2.	Arguments, return
3.	Global v/s Local Variables
4.	Defining and Using Lambda functions map(), filter(), reduce() functions
5.	**File Handling:** read, write and append modes: r, w, a, r+, w+, a+
6.	reading-read(), readline(), readlines(), writing-write(), writelines(), seek(), tell()
7.	Word count, copy file script through file handling concept

❖ Functions

- Python Functions is a block of statements that return the specific task.

- The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

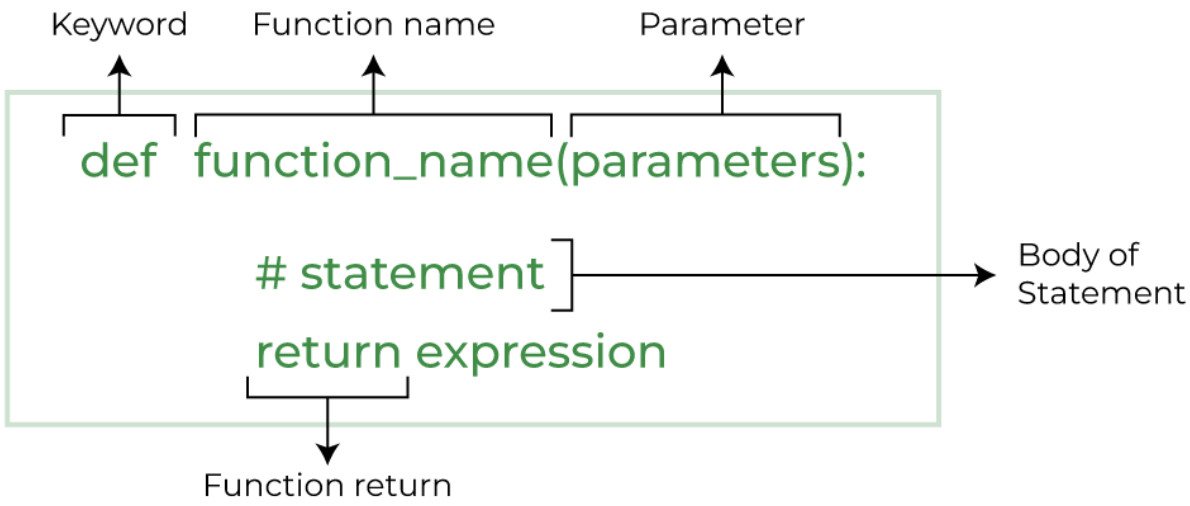
- A function is a block of code which only runs when it is called.

- You can pass data, known as parameters, into a function.

- A function can return data as a result.

- We can define a function in Python, using the **def** keyword.
- Function call using name of function.
- We can add any type of functionalities and properties to it as we require.
- **Benefits of Using Functions**
    - Increase Code Readability
    - Increase Code Reusability

✓ **Syntax:**



✓ **Types of Functions in Python**

- **Built-in library function:** These are Standard functions in Python that are available to use.

- **User-defined function:** We can create our own functions based on our requirements.

✓ **Example:**

```
# A simple create User Define function
    def fun():
        print("Welcome to UDF")

# calling function using function name
fun()
```

❖ Arguments

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses.
- You can add as many arguments as you want, just separate them with a comma.

✓ Parameters or Arguments?

- 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.

Example:

```
def my_function(fname):
print(“Hello, ” + fname)

my_function("Guido van Rossum")
my_function("Taylor Otwell")
my_function("Joan of  Arc")
```

✓ Types of Python Function Arguments

1. **Default argument**
2. **Keyword arguments (named arguments)**
3. **Positional arguments**
4. **Arbitrary arguments** (variable-length arguments *args and **kwargs)

1. Default Arguments

- A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument

Example:

```
# default arguments
        def myFun(x, y=50):
            print("x: ", x)
            print("y: ", y)


myFun(10)
```

2. Keyword Arguments

- The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

Example:

```
def student(firstname, lastname):
 print(firstname, lastname)


# Keyword arguments
student(firstname=’Bhavna’, lastname=’Zala’)
student(lastname='Python', firstname='Programming')
```

3. Positional Arguments

We used the Position argument during the function call so that the first argument (or value) is assigned to name and the second argument (or value) is assigned to age.

By changing the position, or if you forget the order of the positions, the values can be used in the wrong places

Example:

```
def nameAge(name, age):
    print("Hi, I am", name)
    print("My age is ", age)


# You will get correct output because
# argument is given in order
print("Case-1:")
nameAge("Suraj", 27)
# You will get incorrect output because
# argument is not in order
print("\nCase-2:")
nameAge(27, "Suraj")
```

Output:

```
Case-1:
Hi, I am Suraj
My age is  27
Case-2:
Hi, I am 27
My age is  Suraj
```

4. Arbitrary Keyword Arguments

- Arbitrary Keyword Arguments, *args, and **kwargs can pass a variable number of arguments to a function using special symbols.

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

- This way the function will receive a *tuple* of arguments, and can access the items accordingly

There are two special symbols:

- *args in Python (Non-Keyword Arguments)
- **kwargs in Python (Keyword Arguments)\

✓ **Example(*args):** *Variable length non-keywords argument*

```
def myFun(*kids):

print("The youngest cshild is " + kids[0])


myFun("Vashu", "Pihu", "Krushnavi")
```

O/P:

The youngest child is Vashu

✓ **Example(*kwargs):** *Variable length keyword arguments*

```
def my_function(**kid):

  print("His last name is " + kid["lName"])

myFun(fName=’Guido’, mName='van', lName ='Rossum')
```

O/P: His last name is Rossum

❖ Recursion

- Python also accepts function recursion, which means a defined function can call itself.
- Recursion is a common mathematical and programming concept. It means that a function calls itself.
- This has the benefit of meaning that you can loop through data to reach a result.
- The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power.
- However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.
- In this example, tri_recursion() is a function that we have defined to call itself ("recurse").
- We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

Example:

```
def tri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
    result = 0
  return result

print("Recursion Example Results:")
tri_recursion(6)
```

❖ Return Values

- A return statement is used to end the execution of the function call and “returns” the result (value of the expression following the return keyword) to the caller.
- The statements after the return statements are not executed.

- If the return statement is without any expression, then the special value None is returned.

- A return statement is overall used to invoke a function so that the passed statements can be executed.

Note: Return statement cannot be used outside the function.

Syntax:

```
def fun():
    statements
    .
    .
    return [expression]
```

Example:

```
def cube(x):
    r=x**3
    return r
```

❖ Global and Local Variables

✓ Global Variables

- Variables that are created outside of a function (as in all of the examples in the previous pages) are known as global variables.
- Global variables can be used by everyone, both inside of functions and outside
- If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function.
- The global variable with the same name will remain as it was, global and with the original value.

Example-1:

```
x = "awesome"
        def myfunc():
  print("Python is " + x)


myfunc()
```

Example: (same name inside of function**)**

```
x = "awesome"
def myfunc():
  x = "fantastic"
  print("Python is " + x)
myfunc()


print("Python is " + x)
```

✓ global Keyword

- Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.
- To create a global variable inside a function, you can use the global keyword.

Example:

```
def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

✓ Local Scope

- A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

Example:

```
def myfunc():
  x = 300
  print(x)

myfunc()
```

Difference between Local Variable Vs. Global Variables

Comparision Basis	Global Variable	Local Variable
Definition	declared outside the functions	declared within the functions
Lifetime	They are created the execution of the program begins and are lost when the program is ended	They are created when the function starts its execution and are lost when the function ends
Data Sharing	Offers Data Sharing	It doesn't offers Data Sharing
Scope	Can be access throughout the code	Can access only inside the function
Parameters needed	parameter passing is not necessary	parameter passing is necessary
Storage	A fixed location selected by the compiler	They are kept on the stack
Value	Once the value changes it is reflected throughout the code	once changed the variable don't affect other functions of the program

❖ Lambda Functions

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.
- Lambda Functions are anonymous functions means that the function is without a name.
- As we already know the *def* keyword is used to define a normal function in Python. Similarly, the *lambda* keyword is used to define an anonymous function in Python.

Syntax:

lambda *arguments* : *expression*

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming, besides other types of expressions in functions

Why Use Lambda Functions?

- The power of lambda is better shown when you use them as an anonymous function inside another function.

Example:

```
x = lambda a : a + 10
print(x(5))
```

❖ map()

- The map () function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple, etc.).

Syntax: map(fun, iter)

Parameters:

- *fun: It is a function to which map passes each element of given iterable.*
- *iter: iterable object to be mapped.*

Example:

```
# Function to return double of n
def double(n):
    return n * 2

# Using map to double all numbers
numbers = [5, 6, 7, 8]
result = map(double, numbers)
print(list(result))
```

❖ filter()

- The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

Syntax: filter(function, sequence)

Parameters:

- *function: function that tests if each element of a sequence is true or not.*
- *sequence: sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.*

Example:

```
# Define a function to check if a number is even
def is_even(n):
    return n % 2 == 0

# Define a list of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Use filter to filter out even numbers
even_numbers = filter(is_even, numbers)
print("Even numbers:", list(even_numbers))
```

❖ reduce()

- The reduce function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along.
- This function is defined in “functools” module.

Syntax: reduce(func, iterable[, initial])

Parameters:

- ***fun***: *It is a function to execuate on each element of the iterable object*
- ***iter***: *It is iterable to be reduced*

Example:

```
import functools


# Define a list of numbers
numbers = [1, 2, 3, 4]


# Use reduce to compute the product of list elements
product = functools.reduce(lambda x, y: x * y, numbers)
print("Product of list elements:", product)
```

❖ File Handling

- File handling is an important part of any web application.
- Python has several functions for creating, reading, updating, and deleting files.
- File handling in Python is a powerful and versatile tool that can be used to perform a wide range of operations
- Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.
- The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, like other concepts of Python, this concept here is also easy and short.

✓ Advantages of File Handling:

- **Versatility**: File handling allows you to perform a wide range of operations, such as creating, reading, writing, appending, renaming, and deleting files.

- **Flexibility** : File handling is highly flexible, as it allows you to work with different file types (e.g. text files, binary files, CSV files , etc.), and to perform different operations on files (e.g. read, write, append, etc.).

- **User – friendly**: Python provides a user-friendly interface for file handling, making it easy to create, read, and manipulate files.

- **Cross-platform**: Python file-handling functions work across different platforms (e.g. Windows, Mac, Linux), allowing for seamless integration and compatibility.

✓ **Disadvantages of File Handling:**

- **Error-prone:** File handling operations can be prone to errors, especially if the code is not carefully written or if there are issues with the file system (e.g. file permissions, file locks, etc.).
- **Security risks**: File handling can also pose security risks, especially if the program accepts user input that can be used to access or modify sensitive files on the system.

- **Complexity**: File handling can be complex, especially when working with more advanced file formats or operations. Careful attention must be paid to the code to ensure that files are handled properly and securely.

- **Performance**: File handling operations can be slower than other programming languages, especially when dealing with large files or performing complex operations.

❖ **File Open**

- The key function for working with files in Python is the open() function.
- The open() function takes two parameters; *filename*, and *mode*.
- There are four different methods (modes) for opening a file

✓ **"r"**

- Read.
- Default value.
- open an existing file for a read operation

✓ **"a"**

- Append
- open an existing file for append operation. It won't override existing data.

- ✓ **"w"**
  - Write
  - open an existing file for a write operation.
  - If the file already contains some data, then it will be overridden but if the file is not present then it creates the file as well.
- ✓ **"x"**
  - Create
  - Creates the specified file, returns an error if the file exists
- ✓ **“r+”**
  - To read and write data into the file.
  - This mode does not override the existing data, but you can modify the data starting from the beginning of the file.
- ✓ **“w+”**
  - To write and read data.
  - It overwrites the previous file if one exists, it will truncate the file to zero length or create a file if it does not exist.
- ✓ **“a+”**
  - To append and read data from the file.
  - It won’t override existing data.

In addition you can specify if the file should be handled as binary or text mode

- ✓ **"t"** - Text - Default value. Text mode
- ✓ **"b"** - Binary - Binary mode (e.g. images)

Syntax: f = open("demofile.txt")

Example: f = open("demofile.txt", "rt")

Example:

```
file1 = open("myfile.txt", "w")
L = ["This is Delhi \n", "This is Paris \n", "This is London"]
file1.writelines(L)
file1.close()

# Append-adds at last
file1 = open("myfile.txt", "a")              # append mode
file1.write("Today \n")
file1.close()

file1 = open("myfile.txt", "r")
print("Output of Readlines after appending")
print(file1.read())
print()
file1.close()

# Write-Overwrites
file1 = open("myfile.txt", "w")              # write mode
file1.write("Tomorrow \n")
file1.close()

file1 = open("myfile.txt", "r")
print("Output of Readlines after writing")
print(file1.read())
print()
file1.close()
```

These functions are used for file handling:

- ✓ **Reading**
  - **read():** Reads the entire file content as a single string.
  - **readline():** Reads a single line from the file and returns it as a string.
  - **readlines():** Reads all lines from the file and returns them as a list of strings.

- ✓ **Writing**
  - **write():** Writes a string to the file.
  - **writelines():** Writes a list of strings to the file.

- ✓ **Other**
  - **seek(offset, whence=0):** Moves the file pointer to a specific position within the file.
  - **tell():** Returns the current position of the file pointer.

1. read():

- To open the file, use the built-in open() function.
- The open() function returns a file object, which has a read() method for reading the content of the file
- By default the read() method returns the whole text, but you can also specify how many characters you want to return.

Assume we have the following file.

demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

Example-1:

```
f = open("D:\\myfiles\demofile.txt", "r")
print(f.read())
```

Example-2:

```
f = open("D:\\myfiles\demofile.txt", "r")
print(f.read(5))
```

2. readline():

- You can return one line by using the readline() method.
- By calling readline() two or more times, you can read the two first lines

Example-1:

```
f = open("demofile.txt", "r")
print(f.readline())
```

Example-2:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

❖ Close Files

Note: You should always close your files. In some cases, due to buffering, changes made to a file may not show until you close the file.

Example:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

3. readlines():

- The readlines() method returns a list containing each line in the file as a list item.
- Use the hint parameter to limit the number of lines returned. If the total number of bytes returned exceeds the specified number, no more lines are returned.

Syntax: file.*readlines(*hint*)*

Parameter

Hint: Optional. If the number of bytes returned exceed the hint number, no more lines will be returned. Default value is -1, which means all lines will be returned.

Example-1: (Return all lines in the file, as a list where each line is an item in the list object:**)**

```
f = open("demofile.txt", "r")
print(f.readlines())
```

Example-2: (Do not return the next line if the total number of returned bytes are more than 33)

```
f = open("demofile.txt", "r")
print(f.readlines(33))
```

4. write():

- Inserts the any string in a single line in the text file.

Example:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

5. writelines():

- The writelines() method writes the items of a list to the file.
- Where the texts will be inserted depends on the file mode and stream position.
- For a list of string elements, each string is inserted in the text file. Used to insert multiple strings at a single time.

Syntax: file.*writelines(*list*)*

Parameter

List: The list of texts or byte objects that will be inserted.

Example:

```
f = open("demofile3.txt", "a")
f.writelines(["\nSee you soon!", "\nOver and out."])
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

6. seek():

- seek() method is used to change the position of the file pointer within a file.
- This is useful when you want to read or write data from a specific location in the file.

Syntax: file.seek(offset, whence=0)

- **offset:** An integer representing the number of bytes to move the file pointer.
- **whence:** An optional integer that specifies the reference point for the offset.

- **0 (default):** Beginning of the file (os.SEEK_SET)
- **1**: Current position (os.SEEK_CUR)
- **2**: End of the file (os.SEEK_END)

Example:

```
with open("myfile.txt", "r") as f:
    # Move to the 10th byte from the beginning
    f.seek(10)
    data = f.read(5)  # Read 5 bytes from the 10th byte onwards
    print(data)

    # Move 5 bytes backward from the current position
    f.seek(-5, 1)
    data = f.read(5)
    print(data)
```

7. tell():

- tell() method is used to get the current position of the file pointer within the file.

Syntax: file_object.tell()

Example:

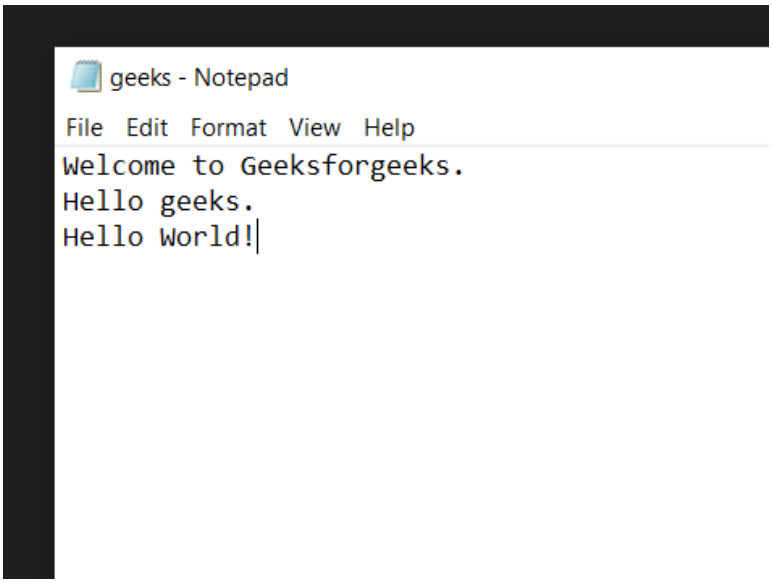
```
with open("myfile.txt", "r") as f:
    # Read the first 10 characters
    f.read(10)

    # Get the current file pointer position
    position = f.tell()

    print("Current position:", position)
```

Explanation:

- The with open(...) as f: statement opens the file "myfile.txt" in read mode ("r").
- f.read(10) reads the first 10 characters from the file.
- f.tell() returns the current position of the file pointer, which is the number of bytes read from the beginning of the file.
- The print() statement outputs the current position.

❖ Count Words

- we create a text file of which we want to count the number of words. Let this file be SampleFile.txt with the following contents.



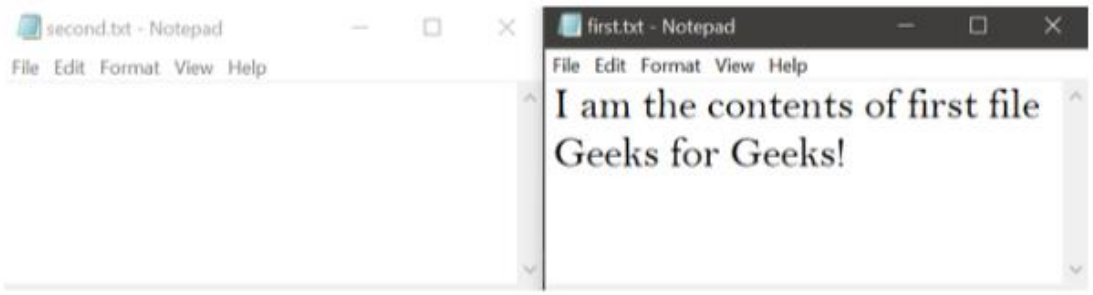
Example:

```
number_of_words = 0

with open(r'SampleFile.txt','r') as fil

data = file.read()
lines = data.split()
number_of_words += len(lines)

print(number_of_words)
```

❖ Copy file scripts through file handing concepts

Given two text files, the task is to write a Python program to copy contents of the first file into the second file.

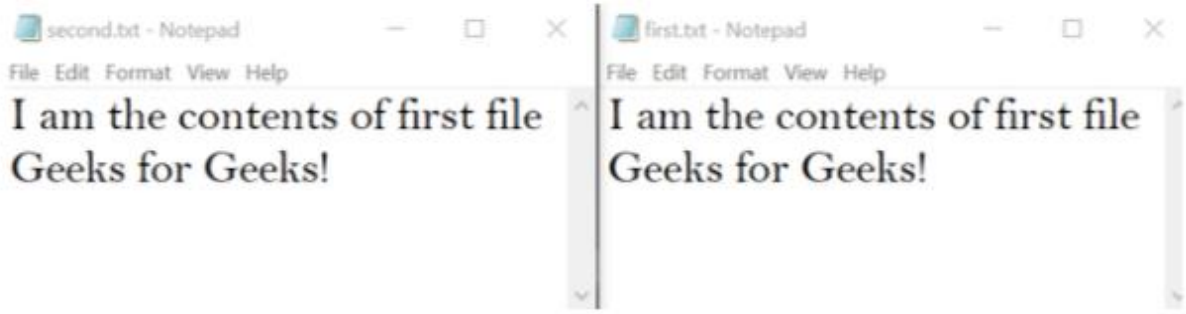
The text files which are going to be used are *second.txt* and *first.txt*:



Example:

```
# open both files
with open('first.txt','r') as firstfile, open('second.txt','a') as secondfile:

    # read content from first file
    for line in firstfile:

        # append content to second file
        secondfile.write(line)
```

Output:



Thank You