T302(a)
Python Programming
Unit-II

Learn to define and invoke functions and comprehend the use of arguments and parameters to pass information to a function as well as return information from a function.

Understand the purpose of functions in Python
Define and invoke functions
Receive the returned data from functions
Understand the use of modules and built-in functions in Python
Packages and PIP

Python function is a block of code that performs a specific and well defined task.

Two main advantages of function are:

- They help us divide our program into multiple tasks. For each task we can define a function. This makes the code modular.
- Functions provide a reuse mechanism. The same function can be called any number of times.

There are two types of Python functions:

- Built-in functions - Ex. len( ), sorted( ), min( ), max( ), etc.
- User-defined functions

The built-in function dir() returns a sorted list of comma separated strings containing the names of functions, classes and variables as defined in the module. For example, you can find all the functions supported by the math module by passing the module name as an argument to the dir() function.

Another built-in function you will find useful is the help() function which invokes the built-in help system. The argument to the help() function is a string, which is looked up as the name of a module, function, class, method, keyword, or documentation topic, and then a related help page is printed in the console. For example, if you want to find information about gcd() function in math module then pass the function name as an argument without parenthesis.

Modules in Python are reusable libraries of code having .py extension, which implements a group of methods and statements. Python comes with many built-in modules as part of the standard library.

```
>>>import math
>>>dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc',
'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2',
```

'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']

>>>help(math.factorial)

>>>math.factorial(5) # The module name and function name are separated by a dot.

120

To use a module in your program, import the module using import statement. All the import statements are placed at the beginning of the program.

The math module is part of the Python standard library which provides access to various mathematical functions and is always available to the programmer.

Third-party modules or libraries can be installed and managed using Python's package manager pip. The syntax for pip is,

>>>pip install module_name

Arrow is a popular Python library that offers a sensible, human-friendly approach to cre-ating, manipulating, formatting and converting dates, times, and timestamps.

To install the arrow module, open a command prompt window and type the below command from any location.

C:\> pip install arrow

```
>>>import arrow
>>>a=arrow.utcnow()
>>>a.now()
<Arrow [2022-03-20T15:59:40.421663+05:30]>
```

UTC is the time standard commonly used across the world. The world's timing centers have agreed to keep their time scales closely synchronized—or coordinated—therefore the name Coordinated Universal Time. Current date including time is displayed using now() function

Function Definition and Calling the Function

You can create your own functions and use them as and where it is needed. User-defined functions are reusable code blocks created by users to perform some specific task in the program.

In Python, a function definition consists of the def keyword, followed by

The name of the function. The function's name has to adhere to the same naming rules as variables: use letters, numbers, or an underscore, but the name cannot start with a number. Also, you cannot use a keyword as a function name.

A list of parameters to the function are enclosed in parentheses and separated by commas. Some functions do not have any parameters at all while others may have one or more parameters.

A colon is required at the end of the function header. The first line of the function definition which includes the name of the function is called the function header.

Block of statements that define the body of the function start at the next line of the function header and they must have the same indentation level.

The def keyword introduces a function definition. The term parameter or formal param-eter is often used to refer to the variables as found in the function definition.

Defining a function does not execute it. Defining a function simply names the function and specifies what to do when the function is called. Calling the function actually per-forms the specified actions with the indicated parameters.

The syntax for function call or calling function is,

function_name(argument_1, argument_2,…,argument_n)

Arguments are the actual value that is passed into the calling function. There must be a one to one correspondence between the formal parameters in the function definition and the actual arguments of the calling function. When a function is called, the formal parameters are temporarily “bound” to the arguments and their initial values are assigned through the calling function.

A function should be defined before it is called and the block of statements in the function definition are executed only after calling the function. Normally, statements in the Python program are executed one after the other, in the order in which they are written. Function definitions do not alter the flow of execution of the program. When you call a function, the control flows from the calling function to the function definition. Once the block of statements in the function definition is executed, then the control flows back to the calling function and proceeds with the next statement. Python interpreter keeps track of the flow of control between different statements in the program.

When the control returns to the calling function from the function definition then the formal parameters and other variables in the function definition no longer contain any values.

Before executing the code in the source program, the Python interpreter automatically defines few special variables. If the Python interpreter is running the source program as a stand-alone main program, it sets the special built-in __name__ variable to have a string value "__main__". After setting up these special variables, the Python interpreter reads the program to execute the code found in it. All of the code that is at indentation level 0 gets executed. Block of statements in the function definition is not executed unless the function is called.

if __name__ == "__main__":
statement(s)

The special variable, __name__ with "__main__", is the entry point to your program. When Python interpreter reads the if statement and sees that __name__ does equal to "__main__", it will execute the block of statements present there.

When you code Python programs, it is a best practice to put all the relevant necessary calling functions inside the main() function definition.

The return statement terminates the execution of the function definition in which it appears and returns control to the calling function. It can also return an optional value to its calling function. n Python, it is possible to define functions without a return statement. Functions like this are called void functions, and they return None.

```
#Program to Check If a 3 Digit Number Is Armstrong Number or Not
user_number = int(input("Enter a 3 digit positive number to check for Armstrong number"))
def check_armstrong_number(number):
    result = 0
    temp = number
    while temp != 0:
        last_digit = temp % 10
        result += pow(last_digit, 3)
        temp = int(temp / 10)
          if number == result:
      print(f"Entered number {number} is a Armstrong number")
    else:
      print(f"Entered number {number} is not a Armstrong number")
def main():
  check_armstrong_number(user_number)
if __name__ == "__main__":
  main()
```

You can nest a function definition within another function definition. A nested function (inner function definition) can "inherit" the arguments and variables of its outer function definition. In other words, the inner function contains the scope of the outer unction. The inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function. The inner function definition can be invoked by calling it from within the outer function definition.

```
#Calculate and Add the Surface Area of Two Cubes. Use Nested Functions

def add_cubes(a, b):
  def cube_surface_area(x):
    return 6 * pow(x, 2)
  return cube_surface_area(a) + cube_surface_area(b)
def main():
  result = add_cubes(2, 3)
  print(f"The surface area after adding two Cubes is {result}")
if __name__ == "__main__":
  main()
```

Default Parameters

In some situations, it might be useful to set a default value to the parameters of the function definition. This is where default parameters can help. Each default parameter has a default value as part of its function definition. Any calling function must provide arguments for all required parameters in the function definition but can omit arguments for default parameters. If no argument is sent for that parameter, the default value is used. Usually, the default parameters are defined at the end of the parameter list, after any required parameters and non-default parameters cannot follow default parameters. The default value is evaluated only once.

Zelenskyy works in Data Analytics

Putin has interest in Internet of Things

```
def work_area(prompt, domain="Data Analytics"):
    print(f"{prompt} {domain}")
def main():
    work_area("Zelenskyy works in")
    work_area("Putin has interest in", "Internet of Things")
if __name__ == "__main__":
    main()
```

Scope and Lifetime of Variables

Python programs have two scopes: global and local. A variable is a global variable if its value is accessible and modifiable throughout your program. Global variables have a global scope. A variable that is defined inside a function definition is a local variable. The lifetime of a variable refers to the duration of its existence. The local variable is created and destroyed every time the function is executed, and it cannot be accessed by any code out-side the function definition. Local variables inside a function definition have local scope and exist as long as the function is executing.

It is possible to access global variables from inside a function, as long as you have not defined a local variable with the same name. A local variable can have the same name as a global variable, but they are totally different so changing the value of the local variable has no effect on the global variable. Only the local variable has meaning inside the function in which it is defined.

```
test_variable = 5
def outer_function():
    test_variable = 60
    def inner_function():
        test_variable = 100
        print(f"Local variable value of {test_variable} having local scope to inner function is displayed")
    inner_function()
    print(f"Local variable value of {test_variable} having local scope to outer func-tion is displayed ")
outer_function()
print(f"Global variable value of {test_variable} is displayed ")
```

Local variable value of 100 having local scope to inner function is displayed
Local variable value of 60 having local scope to outer function is displayed
Global variable value of 5 is displayed

*args and **kwargs

*args and **kwargs are special keyword which allows function to take variable length argument: Arbitrary Arguments, *args

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:

If the number of arguments is unknown, add a * before the parameter name:

def adder(*num): ** sum = 0** ** for n in num:** ** sum = sum + n** ** print("Sum:",sum)** **adder(3,5)** **adder(4,5,6,7)** **adder(1,2,3,5,6)**	**def my_function(*kids):** ** print("The youngest child is " + kids[2])** **my_function("Om", "Sai", "Ram")**
Sum: 8 **Sum: 22** **Sum: 17**	**The youngest child is Ram**

Python **kwargs

Python passes variable length non keyword argument to function using *args but we cannot use this to pass keyword argument. For this problem Python has got a solution called **kwargs, it allows us to pass the variable length of keyword arguments to the function.

In the function, we use the double asterisk ** before the parameter name to denote this type of argument. The arguments are passed as a dictionary and these arguments make a dictionary inside function with name same as the parameter excluding double asterisk **.

```
def intro(**data):
    print("\nData type of argument:",type(data))
    for key, value in data.items():
        print("{} is {}".format(key,value))
intro(Firstname="Sita", Lastname="Sharma", Age=22, Phone=1234567890)
intro(Firstname="John", Lastname="Wood", Email="johnwood@nomail.com", Country="Wakanda",
Age=25, Phone=9876543210)
```

Data type of argument: <class 'dict'> **Firstname is Pooja** **Lastname is Ray** **Age is 22** **Phone is 1234567890** **Data type of argument: <class 'dict'>** **Firstname is Rahul** **Lastname is Kar** **Email is itsme@nomail.com** **Country is India** **Age is 25** **Phone is 9876543210**	**In the above program, we have a function intro() with **data as a parameter. We passed two dictionaries with variable argument length to the intro() function. We have for loop inside intro() function which works on the data of passed dictionary and prints the value of the dictionary.**

Review Questions

1. Define function. What are the advantages of using a function?
2. Differentiate between user-defined function and built-in functions.
3. Explain with syntax how to create a user-defined functions and how to call the user-defined function from the main function.
4. Explain the built-in functions with examples in Python.
5. Differentiate between local and global variables with suitable examples.
6. Explain the advantages of *args and **kwargs with examples.
7. Demonstrate how functions return multiple values with an example.
8. Explain the utility of docstrings?
9. Write a program using functions to perform the arithmetic operations.
10. Write a program to find the largest of three numbers using functions.
11. Write a Python program using functions to find the value of ${}^{n}P_{r}$ and ${}^{n}C_{r}$.
12. Write a Python function named area that finds the area of a pentagon.
13. Write a program using functions to display Pascal's triangle.
14. Write a program using functions to print harmonic progression series and its sum till N terms.
15. Write a program using functions to do the following tasks:
    a. Convert milliseconds to hours, minutes and seconds.
    b. Compute a sales commission, given the sales amount and the commission rate.
    c. Convert Celsius to Fahrenheit.
    d. Compute the monthly payment, given the loan amount, number of years and the annual interest rate.