EXPERIMENT NO: 09

Title: Write a user define function to implement following features: Function without argument, Function with argument, Function returning value.

Prior Concepts: Basic python expressions, decision making and looping, data types such as lists, tuples, sets, dictionaries and its manipulation.

Objective: To understand the purpose of functions, its definition, calling and different types of functions based on arguments and return types.

Theory:

Functions are one of the core building blocks of Python programming, designed to improve code modularity, readability, and reusability. Functions help to break down complex problems into smaller, manageable pieces and reduce redundancy in code.

Basic Concepts of Functions

Definition of a Function: A function is defined with a unique name, followed by parentheses (). Inside the parentheses, you can specify parameters (inputs) that the function can use. The function's code block is indented under the function header, starting with the keyword def.

Types of Functions:

- **Built-in Functions:** These are predefined functions that come with Python, such as print(), len(), and type().
- **User-Defined Functions:** These are functions that programmers create to perform specific tasks.

Benefits of using a Function:

- **Reusability:** Functions allow you to write a block of code once and use it multiple times within the program.
- **Modularity:** Functions divide code into smaller, logical sections, making programs easier to read and debug.
- **Maintainability:** Functions make it easier to update or fix a section of code without affecting other parts of the program.
- **Abstraction:** By using functions, the details of an operation can be hidden. You only need to know the function’s purpose, not its internal code.

Defining a Function: To define a function in Python, you use the def keyword followed by:

- Function Name: A name you give to the function.
- Parentheses (): Inside the parentheses, you can specify any input parameters the function needs (or leave it empty if none are required).
- Colon: Indicates the start of the function body.

- Indentation: The code within the function is indented to mark it as part of the function.
- The return Statement: It sends a function's result back to the caller. When a return statement is reached, the function exits and the specified value is returned. If there's no return statement, the function returns None by default.

Syntax:

```
def function_name(parameters):
        # Code block to perform the task
        # You can include any operations or statements here
```

Examples:

#function without parameters and return statement
def say_hello():
        print("Hello! Welcome to learning Python functions.")
	#function with parameters and return statement
	def multiply (num1, num2):
	    return num1*num2

Calling a Function: Once a function is defined, you can call it anywhere in your code by using its name followed by parentheses, and any required arguments inside the parentheses. Taking above examples, we can call the function as:

Examples:

#calling a function without parameters
say_hello()
	#calling a function with paremeters
	result = multiply(5,3)
	print("The multiplication result=",result)

Scope of Variables

- **Local Scope:** Variables defined inside a function are local to that function and cannot be accessed outside of it.
- **Global Scope:** Variables defined outside all functions are global, meaning they can be accessed anywhere in the program.

Procedure

Step 1: Setting Up the Environment

1. Open your Python IDE or text editor.
2. Create a new Python file (your_script_name.py) to write the code.

Step 2: Writing the Python Code

Types of Functions by Argument and Return Value:

1. **Function without Arguments:** A function with no parameters simply performs its task without any arguments or parameters. It does not contain any return statement.

```
def area_of_rectangle():
   l=int(input("Enter length:"))
   b=int(input("Enter breadth:"))
   area= l*b

print("area of rectangle is = ",area)
 # Calling the function without arguments
 area_of_rectangle()
```

2. **Function with Return Value:** A functtion with a return statement sends a value back to the part of the program that called it.

```
#function without argument and with return type
def area_of_rectangle():
   l=int(input("Enter length:"))
   b=int(input("Enter breadth:"))
   area= l*b return area

# Calling the function and using the returned value
rectangle_area = area_of_rectangle()
print("The area of the rectangle is =", rectangle_area)
```

```
#function with arguments and return type
def area_of_rectangle(length, width):
      area = length * width return area

# Calling the function and using the returned value
rectangle_area = area_of_rectangle(7, 3)
print("The area of the rectangle is =", rectangle_area)
```

3. **Function with Arguments:** A function with arguments accepts data that can be used within the function. In Python, functions can accept different types of arguments to make them flexible and versatile.

3.1. **Positional Arguments:** These are the most common and straightforward type. The order of arguments in the function call must match the order in the function definition. If the arguments are given in the wrong order, it can lead to unexpected behavior or incorrect results.

```
def introduce(name, age):
        print("My name is ", name," and I am ", age," years old.")

# Calling with positional arguments
introduce("Rohan", 45)

#Drawback of positional argument
introduce(45,"Rohan")
```

3.2. Keyword Arguments: With keyword arguments, you can specify each argument by name, allowing them to be in any order. This enhances readability and is helpful when functions have many parameters.

```
def introduce(name, age):
        print("My name is ", name," and I am ", age," years old.")

#Calling with Keyword arguments
introduce(name="Rohan",  age=45)

#Drawback of positional argument overcome using keywords introduce(age=45,
name="Rohan")
```

3.3. Default Arguments: Default arguments are specified in the function definition, allowing the function to have pre-set values for certain parameters. If no value is provided during the function call, the default value is used.

```
def introduce(name, age=18):
        print(f"My name is {name} and I am {age} years old.")

# Calling with one argument (uses default age)
 introduce("Sam")
```

3.4. Variable-Length Arguments (args and kwargs): These are useful especially when you don't know in advance how many arguments you may need to pass to a function.

3.4.1. *args (Non-Keyword Arguments): *args allows you to pass a variable number of positional arguments to a function. When you use *args in a function, it gathers all extra positional arguments passed to the function into a tuple.

When to Use *args:

- When you want to write functions that can handle a varying number of arguments.
- When you don’t know how many arguments will be passed during the function call.
- When you need to pass a collection (like a list or tuple) as multiple arguments.

Syntax of *args: In the function definition,

*args is written with an asterisk (*) before the parameter name. Although args is a common convention, you can name it anything you like (e.g., *numbers, *items).

```
def my_function(*args):
        # args is a tuple of all positional arguments passed to the function
        print(args)
```

Example:

```
def sum_numbers(*numbers): total
        = 0
        for num in numbers: total +=
                num
        return total

print(sum_numbers(1, 2, 3))
print(sum_numbers(5, 10, 15, 20))
```

3.4.2. **kwargs (Keyword Arguments): These allows you to pass a variable number of keyword arguments (arguments with a key-value pair) to a function. When you use **kwargs, it collects these keyword arguments into a dictionary.

When to Use **kwargs:

- When you want to handle named arguments dynamically.
- When you don’t know which specific named arguments will be passed to the function.
- When you need to pass configuration or settings as named arguments.

Syntax of **kwargs: In the function definition, **kwargs is written with two asterisks (**) before the parameter name. kwargs is a common convention, but you can name it anything you like (e.g., **data, **options).

```
def my_function(**kwargs):
        # kwargs is a dictionary of all keyword arguments passed to the function
              print(kwargs)
```

Example

```
def user_profile(**kwargs):
   for  key,value in kwargs.items():
      print(key, value)
# Passing multiple keyword arguments
user_profile(name="Alice",        age=30,
location="New York")
```

```
def                    order_summary(smart_phone,
**specifications):
print(f"Buying          new: {smart_phone}")
for key, value in (specifications.items()):
   print(f"{key}: {value}")

order_summary("Samsung",          price=121999,
storage=256,RAM=12, color="Titanium Green")
```

Step 3: Running the Python Script

- Save your Python file.
- Run the script in your IDE or from the command line: python your_script_name.py
- Observe the outputs generated from each operation.

Conclusion: Thus, we have successfully studied functions and its types based on the parameters and return type.

Questions:

1. Write a function find_max that takes two numbers as arguments and returns the larger of the two. Example: find_max(10, 20) should return 20.
2. Create a function count_vowels that takes a string as an argument and returns the number of vowels (a, e, i, o, u) in that string. Example: count_vowels("Hello World") should return 3.
3. Create a function register_students that takes a class name as a required argument, a variable number of student names as *args, and optional details like teacher and room_number as **kwargs. The function should print:
   - The class name.
   - The list of students in the class.
   - Any optional details like teacher and room_number, if provided.

EXPERIMENT NO: 10

Title: Python program to demonstrate the use of constructors: default, parameterized constructors.

Prior Concepts: Basic python concepts, decision making and looping, data types such as lists, tuples, sets, dictionaries and its manipulation, functions and its types.

Objective: To understand Object Oriented Programming concepts, classes objects and constructors and its types.

Theory: Object-Oriented Programming (OOP) is a way of organizing code around "objects" rather than just functions and logic. These objects represent real-world entities or concepts, like a car, a student, or a book, each with its unique characteristics and actions.

Basic Concepts in OOP

- **Attributes:** Characteristics that belong to the object. In our Car example, attributes might be color, brand, and year.
- **Methods:** Actions that an object can perform. For a car, methods might include start and stop.

In OOP, we define attributes and methods inside the class to give each object specific characteristics and actions.

Key Benefits of OOP

- **Code Organization:** Grouping related data and functions together makes code easier to manage.
- **Reusability:** Once a class is defined, you can create multiple objects from it, each with its unique data.
- **Scalability**: Classes and objects make it easier to build and expand on code, especially in large projects.

In OOP, two primary components are classes and objects:

1. **Classes:** A class is like a blueprint for creating objects. Think of it as a template that defines the characteristics (attributes) and actions (methods) that an object of this type can have. For example, if we have a class Car, it might define that all cars have characteristics like color, brand, and year, and actions like start and stop.
   Example:

```
class Car:
    # Attributes and methods will be defined here later
```

2. **Objects:** An object is an actual instance created from a class. If Car is the blueprint, then an object would be a specific car, like a "Red Toyota 2020." When we create an object, we call it instantiating the class.
   Example:

```
my_car = Car()
```

Procedure:

Step 1: Setting Up the Environment

1. Open your Python IDE or text editor.
2. Create a new Python file (your_script_name.py) to write the code.

Step 2: Writing the Python Code

Constructor in Python: In Python, the constructor function is defined using the special method _init_(). This method is automatically called when an object is created from a class, so you don't need to call it manually. The word _init_stands for "initialize," as it's where we initialize the object's data.

Syntax:

```
class ClassName:
        def __init__(self, parameters): # Initialize
                the attributes self.attribute = value
```

self: The first parameter in _init____is always self, which represents the object being created. Using self allows us to refer to the object's attributes and methods from within the class.

Why is a Constructor Used?

- **To Initialize Object Attributes:** The constructor sets initial values for an object's attributes, ensuring it has the necessary data to function correctly from the start.
- **Automatic Setup:** Since _init_() is automatically called when an object is created, you don't have to remember to initialize every object manually. This ensures that every object starts with defined values.
- **Provide Flexibility with Parameterized Constructors:** Constructors can accept parameters, allowing objects to be customized at creation.

Types of Constructors

1. **Default Constructor:** A default constructor is a constructor with no parameters (except self). It initializes the object with default values.

Example:

```
class Person:
        def ____init____(self):
```

```
        # Default values for attributes
        self.name = "Unknown"
        self.age = 0

    def show_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Creating an object with the default constructor
person1 = Person()
person1.show_info()
```

2. **Parameterized Constructor:** A parameterized constructor accepts parameters, allowing you to set specific values for the object’s attributes when it’s created.

Example:

```
class Person:
    def ____init____(self, name, age):
        # Initialize attributes with provided values self.name = name
        self.age = age

    def show_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Creating objects with the parameterized constructor
person1 = Person("Rohan", 45)
person2 = Person("Riya", 15)

person1.show_info()
person2.show_info()
```

Example using both:

```
class Car:
    # Constructor to initialize attributes
    def __init__(self, color, brand, year):
        self.color = color       # Set the color attribute
        self.brand = brand       # Set the brand attribute
        self.year = year         # Set the year attribute
        # Method to display car details def
        show_details(self):
                print(f"This is a {self.color} {self.brand} from {self.year}.")

Creating an Object and Using the Constructor:
my_car = Car()
my_car1 = Car("Red", "Toyota", 2020)          # Object creation calls __init__()


my_car.show_details()
my_car1.show_details()
```

Step 3: Running the Python Script

- Save your Python file.
- Run the script in your IDE or from the command line: python your_script_name.py
- Observe the outputs generated from each operation.

Conclusion: Thus, we successfully studied class, objects, constructors and its types in python.

Questions:

1. Create a Book class with a default constructor that consists of a name, author, price. Write a method to display these details. Create an object of the book class using the default constructor and display its details.
2. Create a BankAccount class with a constructor that initializes account_number and balance. Set a default balance of 1000 and if a balance lower than 100 is provided, set it to 100. Create two BankAccount objects with different initial balances and print the details to observe the conditional balance assignment.