IQRA University

CSC 471 Artificial Intelligence

Lab# 02
Review of Functions and OOP in Python

Objective:

This experiment introduces the students to the concept of using functions and Object Oriented Programming in Python. This will help students better implement Artificial Intelligence concept using Python.

Name of Student: **Faraz Alam**

Roll No: **13948** **Sec.** **Saturday (12:00pm-14:00pm)**

Date of Experiment: **10/28/23**

Lab 02: Functions in Python and Review of OOP Concepts

Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions.

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword def followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

```
def functionname( parameters ):

   "function_docstring"
function_suite
return [expression]
```

Calling a Function

Defining a function gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is an example to call the printme() function.

```
# Function definition is here def

printme(str):

   "This prints a passed string into this

function"    print (str)    return

# Now you can call printme function printme("This is first
call to the user defined function!") printme("Again second
call to the same function")
```

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

```
# Function definition is here def

changeme( mylist ):

   "This changes a passed list into this function"    print

("Values inside the function before change: ", mylist)

mylist[2]=50    print ("Values inside the function after

change: ", mylist)    return


# Now you can call changeme function mylist =
[10,20,30] changeme( mylist ) print ("Values
outside the function: ", mylist)
```

```
Values inside the function before change:  [10, 20, 30]
Values inside the function after change:  [10, 20, 50]
Values outside the function:  [10, 20, 50]
```

```
# Function definition is here def

changeme( mylist ):

   "This changes a passed list into this function"    mylist =

[1,2,3,4] # This would assign new reference in mylist    print

("Values inside the function: ", mylist)    return


# Now you can call changeme function mylist =
[10,20,30] changeme( mylist ) print ("Values
outside the function: ", mylist)
```

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

Function Arguments

You can call a function by using the following types of formal arguments.

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

The Return Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return none. All the examples given below are not returning any value. You can return a value from a function.

```
# Function definition is here def

sum( arg1, arg2 ):

   # Add both the parameters and return

them."    total = arg1 + arg2    print

("Inside the function : ", total)    return

total

# Now you can call sum function total =
sum( 10, 20 ) print ("Outside the
function : ", total )
```

```
Inside the function :  30
Outside the function :  30
```

Object Oriented Programming

1. The __init__( ) Method

The __init__( ) method is profound for two reasons. Initialization is the first big step in an object's life; every object must be initialized properly to work properly. The second reason is that the argument values for __init__( ) can take on many forms. Because there are so many ways to provide argument values to __init__( ), there is a vast array of use cases for object creation. We take a look at several of them. We want to maximize clarity, so we need to define an initialization that properly characterizes the problem domain. Before we can get to the __init__( ) method, however, we need to take a look at the implicit class hierarchy in Python, glancing, briefly, at the class named object. This will set the stage for comparing default behavior with the different kinds of behavior we want from our own classes.
In this example, we take a look at different forms of initialization for simple objects (for example, playing cards). After this, we can take a look at more complex objects, such as hands that involve collections and players that involve strategies and states.
Python is a multi-paradigm programming language. Meaning, it supports different programming approach.
One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

Parrot is an object,

- name, age, color are attributes
- singing, dancing are behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself). In Python, the concept of OOP follows some basic principles:

1	***Inheritance***	A process of using details from a new class without modifying existing class.
2	***Encapsulation***	Hiding the private details of a class from other objects.
3	***Polymorphism***	A concept of using common operation in different ways for different data input.

Python Object Inheritance

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called child classes, and the classes that child classes are derived from are called parent classes. It's important to note that child classes override or extend the functionality (e.g., attributes and behaviors) of parent classes. In other words, child classes inherit all of the parent's attributes and behaviors but can also specify different behavior to follow. The most basic type of class is an object, which generally all other classes inherit as their parent. When you define a new class, Python 3 it implicitly uses object as the parent class. So the following two definitions are equivalent:

Exercise 1: Write a class of Dog, each dog must be of species type mammal. Each dog has its name and age. The class can have method for description () and sound () which dog produces. Create an object and perform some operations.

```
class Dog:


    # Class Attribute
species = 'mammal'

    # Initializer / Instance Attributes
def __init__(self, name, age):
        self.name = name
self.age = age

    # instance method
def description(self):
        return "{} is {} years old".format(self.name, self.age)
# instance method     def speak(self, sound):

        return "{} says {}".format(self.name, sound)

# Instantiate the Dog object razer
= Dog("Razer", 6)

# call our instance methods
print(razer.description())
print(razer.speak("Woof Woof"))
```

Exercise 2: Write a class of Dog, each dog must be of species type mammal. Each dog has its name and age. The class can have method for description () and sound () which dog produces. Now this time you need to create two sub classes of Dogs one is Bull Dog and other is Russell Terrier Create few objects and perform some operations including the inheritance.

```
# Parent class class
Dog:
    # Class attribute
species = 'mammal'

    # Initializer / Instance attributes
def __init__(self, name, age):
        self.name = name
self.age = age

    # instance method
def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Child class (inherits from Dog class)
class RussellTerrier(Dog):     def
run(self, speed):
        return "{} runs {}".format(self.name, speed)
# Child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)

# Child classes inherit attributes and #
behaviors from the parent class  thunder
= Bulldog("Thunder", 9)
print(thunder.description())

# Child classes have specific attributes
# and behaviors as well
print(thunder.run("slowly"))


spinter = Bulldog("Spinter", 12)
print(spinter.description()) print(spinter.run("fast"))


roger = RussellTerrier("Roger", 5)
```

Exercise 3: Extending question number 2, now we need to check that either the different dog classes and their objects link with each other or not. In this case we need to create a method to find either it's an instance of each other objects or not.

```
# Parent class class
Dog:
    # Class attribute
species = 'mammal'

    # Initializer / Instance attributes
def __init__(self, name, age):
        self.name = name
self.age = age

    # instance method
def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Child class (inherits from Dog() class)
class RussellTerrier(Dog):     def
run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child class (inherits from Dog() class)
class Bulldog(Dog):     def run(self,
speed):
        return "{} runs {}".format(self.name, speed)


# Child classes inherit attributes and
# behaviors from the parent class thunder
= Bulldog("Thunder", 9)
print(thunder.description())

# Child classes have specific attributes
# and behaviors as well
print(thunder.run("slowly"))

# Is thunder an instance of Dog()?
print(isinstance(thunder, Dog))


# Is thunder_kid an instance of Dog()?

thunder_kid = Dog("ThunderKid", 2) print(isinstance(thunder,
Dog))


# Is Kate an instance of Bulldog()  Kate
= RussellTerrier("Kate", 4)
print(isinstance(Kate, Dog))
```

```
# Is thunder_kid and instance of kate?
print(isinstance(thunder_kid, Kate))
print("Thanks for understanding the concept of OOPs")
```

NOTE:
Make sense? Both thunder_kid and Kate are instances of the Dog() class, while Spinter is not an instance of the Bulldog() class. Then as a sanity check, we tested if kate is an instance of thunder_kid, which is impossible since thunder_kid is an instance of a class rather than a class itself—hence the reason for the TypeError.

Student Exercise

The exercises to this lab are separately provided in the form of Jupyter Notebooks. Go through the provided notebooks and solve the exercises.