UNIVERSITY OF ENGINEERING AND TECHNOLOGY PESHAWAR, JALOZAI CAMPUS

Lab 5: The __init__ Method

Lab Title: EE-271 "OOP & Data Structures Lab"

Time: 10 min/ Task

Lab Report Tasks:

1. Consider the complex number class.

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
```

   a. Define an object `x = Complex`
   b. Define an object `x = Complex(3.0)`
   c. Define an object `x = Complex(3.0, -4.5)`
   d. `Print (x.r, x.i)`
   e. Define an appropriate print method that can effectively print the complex number. Use it to print x.

Lab Work Tasks:

The __init__ method is run every time a new `Dog` object is created and tells Python what the initial **state**—that is, the initial values of the object's properties—of the object should be.

2. Consider the following class. When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance.

```
class MyClass:
    """A simple example class"""
    i = 12345
    def __init__(self):
      self.data = 11

    def f(self):
        return 'hello world'
```

   a. Let's define an instance of the class and print the data attribute of it.
   b. Let pass a value in the object instantiation step like `obj = MyClass(10).`
   c. Modify the __init__ method as:

```
    def __init__(self, dt):
      self.data = dt
```

   Now run the `obj = MyClass(10).`

3. Dog class

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

a. To instantiate an object, type the name of the class, in the original CamelCase, followed by parentheses containing any values that must be passed to the class's .__init__() method.

```
Run the following code.
miles = Dog("Miles", 4)
```

b. Make another object of the class Dog. The object's name is buddy while the dog's name is Buddy and age 9.
c. Print(`miles.name`)
d. Print(`miles.age`)
e. Let if you enter the age of miles wrong and you are interested in amending it with 7, which is the correct age. You can modify it. Please modify it. After modification print the age of miles and buddy.
f. Print all the class and instance attributes of both objects and observe.
g. Make a new instance and print its attributes.
h. Run: `miles = Dog("Miles")`
i. Run: `miles = Dog(4)`
j. `Run` Run: `Dog("Miles", 4)`

Hints:

i. In the body of the .__init__() method, there are two statements using the self variable. The first line, self.name = name, creates an instance attribute called name and assigns to it the value of the name variable that was passed to the .__init__() method. The second line creates an instance attribute called age and assigns to it the value of the age argument.
ii. Name and age are called instance attributes.
iii. After the Dog instances are created, you can access their instance attributes by using **dot notation**:
iv. The important takeaway here is that custom objects are mutable by default. Recall that an object is mutable if it can be altered dynamically. For example, lists and dictionaries are mutable, but strings and tuples are not—they are immutable.

4. Dog

```
class Dog:
    # Class Attribute
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

a. Make two objects of the Dog class.
b. Print the class attribute of both objects.
c. Run this code: `buddy.species == miles.species`
d. Modify the class attribute by Felis silvestris. Now print it for both instances.
e. Again Run this code: `buddy.species == miles.species`

Hints:

i. While instance attributes are specific to each object, **class attributes** are the same for all instances—which in this case is all dogs. In the next example, a class attribute called species is created and assigned the value "Canis familiaris":
ii. Class attributes are defined directly underneath the first line of the class and outside of any method definition. They must be assigned a value because they

are created on a class instance without arguments to determine what their initial value should be.

iii. The .__init__() method has three parameters, so why are only two arguments passed to it in the example?
When you instantiate a Dog object, Python creates a new instance and passes it to the first parameter of .__init__(). This essentially removes the self parameter, so you only need to worry about the name and age parameters.

5. Consider the complex number class.

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
```

f. Define an object `x = Complex`
g. Define an object `x = Complex(3.0)`
h. Define an object `x = Complex(3.0, -4.5)`
i. `Print (x.r, x.i)`
j. Define an appropriate print method that can effectively print the complex number. Use it to print x.

6. Consider the following class example.

```
class Totyota:
    def __init__(self,name,color,model):
        self.name=name
        self.color=color
        self.model=model

    def printdetails(self):
        print(f"Car Company is: {self.name},Car Color is :{self.color},Car model is {self.model} ")
```

a. Make an object of the class `company =Totyota('Corolla')`
b. Make an object of the class `company =Totyota('Corolla','Red')`
c. Make an object of the class `company =Totyota('Corolla','Red',2014)`
d. Print the details using the function `printdetails.`

7. Consider the following class example.

```
class Employee:
    def __init__(self,name,role,salary):
        self.name=name
        self.role=role
        self.salary=salary

    def printdetails(self):
        print( f"Employee name: {self.name} Employee role is :{self.role} Employee salary  is {self.salary} " )
```

a. Make an object of the class `Employee = Employee ('Ali')`
b. Make an object of the class `Employee = Employee ("Ali","Instructor)`
c. Make an object of the class `Employee = Employee ("Ali","Instructor",4000)`
d. Print the details using the function `printdetails.`

8. Understand the following code. The complete code is given just run and play with the code.

```
class Circle:
    def __init__(self,radius):
```

```
        self.radius=radius
```

Now make some object.

```
circle1=Circle(4)
circle2=Circle(8)
circle3=Circle(6)
```

a. Print radius using dot operator.

```
print("Circle1 radius is ",circle1.radius)
print("Circle 2 radius is",circle2.radius)
print("Circle3 radius is ",circle3.radius)
```

b. Print the object directly like `print(Circle1)`.

c. Define a method that can print radius by directly calling by the circle object.

9. Consider the following class definition.

```
class Point:
    pass
```

a. Make two instances named p1 and p2.

b. Now, we have a basic class, but it's fairly useless. It doesn't contain any data, and it doesn't do anything. What do we have to do to assign an attribute to a given object?
It turns out that we don't have to do anything special in the class definition. We can set arbitrary attributes on an instantiated object using the dot notation:

```
p1 = Point()
p2 = Point()
p1.x = 5
p1.y = 4
p2.x = 3
p2.y = 6
```

c. Try different names for objects, instead of p1 and p2. Also, use different attribute names.
Note:

- The value (here 5, 4, 3, and 6) can be anything: a Python primitive, a built-in data type, or another object. It can even be a function or another class!

10. The Python initialization method is the same as any other method, except it has a special name, `__init__`.
Consider the point class below.

```
class Point:
    def move(self, x, y):
        self.x = x
        self.y = y
    def reset(self):
        self.move(0, 0)
```

Modify the above class by adding __init__ method.

```
class Point:
    def __init__(self, x, y):
        self.move(x, y)
    def move(self, x, y):
        self.x = x
        self.y = y
    def reset(self):
        self.move(0, 0)
```

a. Constructing an instance/ object using `point = Point(3, 5)`

b. Print the attributes.

11. Consider

```
# point.py

class Point:
    def __init__(self, x, y):
        print("Initialize the new instance of Point.")
        self.x = x
        self.y = y

    def print(self):
        return f"{type(self).__name__}(x={self.x}, y={self.y})"
```

a. Run the following code:

```
point = Point(21, 42)
print(point)
```

b. The __init__ can be called by the instance and pass value directly. Run the following code.

```
point.__init__(34, 45)
point.x
point.y
print(point)
```

12. Rectangle class

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

a. Define an object by passing the parameters and the print them. For printing use the instance variable and dot operator.

b. Returning some thing from the __init__ method. Run the following code and observe the result.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        return 42


rectangle = Rectangle(21, 42)
```

c. Now run the following code and observe the result.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        return None
```

```
rectangle = Rectangle(21, 42)
```

d. In .__init__(), you can also run any transformation over the input arguments to properly initialize the instance attributes. For example, if your users will use Rectangle directly, then you might want to validate the supplied width and height and make sure that they're correct before initializing the corresponding attributes:
Run the following code and observe the result while supplying different combinations of input. Furthermore, to enhance the exception further is highly recommended.

```
class Rectangle:
    def __init__(self, width, height):
        if not (isinstance(width, (int, float)) and width > 0):
            raise ValueError(f"positive width expected, got {width}")
        self.width = width
        if not (isinstance(height, (int, float)) and height > 0):
            raise ValueError(f"positive height expected, got {height}")
        self.height = height


rectangle = Rectangle(-21, 42)
```

13. Optional arguments allow you to write classes in which the constructor accepts different sets of input arguments at instantiation time. Which arguments to use at a given time will depend on your specific needs and context.
Consider the following code.

```
class Greeter:
    def __init__(self, name, formal=False):
        self.name = name
        self.formal = formal

    def greet(self):
        if self.formal:
            print(f"Good morning, {self.name}!")
        else:
            print(f"Hello, {self.name}!")
```

a. In the following example, you create an informal_greeter object by passing a value to the name argument and relying on the default value of formal. You get an informal greeting on your screen when you call .greet() on the informal_greeter object.

```
informal_greeter = Greeter("Pythonista")
informal_greeter.greet()
```

b. Similarly, in the following example, you use a name and a formal argument to instantiate Greeter. Because formal is True, the result of calling .greet() is a formal greeting.

```
formal_greeter = Greeter("Pythonista", formal=True)
formal_greeter.greet()
```

14. Default arguments

i. Consider the point class below.

```
class Point:
    def __init__(self, x, y):
        self.move(x, y)
    def move(self, x, y):
        self.x = x
        self.y = y
    def reset(self):
        self.move(0, 0)
```

Modify the above class by modifying the argument list of __init__ using default arguments.

```
class Point:
    def __init__(self, x=0, y=0):
        self.move(x, y)
    def move(self, x, y):
        self.x = x
        self.y = y
    def reset(self):
        self.move(0, 0)
```

a. Make an instance of the class point, `point = Point(3, 5).` Print the point.

Note:

- The keyword argument syntax appends an equals sign after each variable name. If the calling object does not provide this argument, then the default argument is used instead.