Student Name: _____ **Roll No:** _____ **Section:** _____

Lab No. 05

Lab 05 – Introduction to Inheritance and Multi-Inheritance

Objectives:

- Understanding the concepts of inheritance.
- Parent and Child Classes
- Overriding Parents Methods
- Multiple-Inheritance
- Super() function

1. Accessing the Variables inside the Class

As we have practiced many times. We can access the attributes inside the parent class or we can change the value of the parameter during the creation of object. Here we have the Revise Exercise to teach how we can change the variable values from age and name.

Revise Exercise 1: Write a class of Boy and Girl, each Boy or Girl classes have name, age and gender as parameters. The objects then can change the values inside it.

```
class Boy:
    name =""
    gender ="Male"
    age = 20

class Girl:
    name =""
    gender ="Female"
    age = 18

Irtiza = Boy()
Ahmed = Boy()
Arisha = Girl()
Ayesha = Girl()

print("My name is : " + "Irtiza")
Irtiza.gender
print(Irtiza.gender)
Irtiza.age = 21
print(Irtiza.age)
```

Student Name: _____ **Roll No:** _____ **Section:** _____

```
print("My name is : " + "Ahmed")
Ahmed.gender
print(Ahmed.gender)
Ahmed.age = 22
print(Ahmed.age)

print("My name is : " + "Arisha")
Arisha.gender
print(Arisha.gender)
Arisha.age = 19
print(Arisha.age)

print("My name is : " + "Ayesha")
Ayesha.gender
print(Ayesha.gender)
Ayesha.age =18
print(Ayesha.age)
```

Exercise 2: Write a class for an arbitrary POS, where we can calculate the number of initiations of each object and can also calculate once to object is deleted.

```
class POS:

    counter = 0

    def_init_(self):
        type(self).counter += 1

    def_del_(self):
        type(self).counter -= 1

if___name___== "_main_":
    x1 = POS()
    print("Number of instances: : " + str(POS.counter))
    x2 = POS()
    print("Number of instances: : " + str(POS.counter))
    x3 = POS()
    print("Number of instances: : " + str(POS.counter))
    del x2
    print("Number of instances: : " + str(POS.counter))
    del x1
    print("Number of instances: : " + str(POS.counter))
```

Student Name: _____ **Roll No:** _____ **Section:** _____

Object-oriented programming creates reusable patterns of code to curtail redundancy in development projects. One way that object-oriented programming achieves recyclable code is through inheritance, when one subclass can leverage code from another base class.

In this lab will go through some of the major aspects of inheritance in Python, including how parent classes and child classes work, how to override methods and attributes, how to use the super() function, and how to make use of multiple inheritance.

2. What Is Inheritance?

Inheritance is when a class uses code constructed within another class. If we think of inheritance in terms of biology, we can think of a child inheriting certain traits from their parent. That is, a child can inherit a parent's height or eye color. Children also may share the same last name with their parents.
Classes called child classes or subclasses inherit methods and variables from parent classes or base classes.
We can think of a parent class called Parent that has class variables for last_name, height, and eye_color that the child class Child will inherit from the Parent.
Because the Child subclass is inheriting from the Parent base class, the Child class can reuse the code of Parent, allowing the programmer to use fewer lines of code and decrease redundancy.

3. Parent Classes

Parent or base classes create a pattern out of which child or subclasses can be based on. Parent classes allow us to create child classes through inheritance without having to write the same code over again each time. Any class can be made into a parent class, so they are each fully functional classes in their own right, rather than just templates.

Let's say we have a general Bank_account parent class that has Personal_account and Business_account child classes. Many of the methods between personal and business accounts will be similar, such as methods to withdraw and deposit money, so those can belong to the parent class of Bank_account. The Business_account subclass would have methods specific to it, including perhaps a way to collect business records and forms, as well as an employee_identification_number variable.

Similarly, an Animal class may have eating() and sleeping() methods, and a Snake subclass may include its own specific hissing() and slithering() methods.

Student Name: _____ **Roll No:** _____ **Section:** _____

Let's create a Fish parent class that we will later use to construct types of fish as its subclasses. Each of these fish will have first names and last names in addition to characteristics.

We'll create a new file called fish.py and start with the_init_() constructor method, which we'll populate with first_name and last_name class variables for each Fish object or subclass.

Exercise 3: Create a class Fish with swim() and swim_backwards() methods as most of the fishes cannot swim backward.

```
class Fish:
    def_init_(self, first_name, last_name="Fish"):
        self.first_name = first_name
        self.last_name = last_name

    def swim(self):
        print("The fish is swimming.")

    def swim_backwards(self):
        print("The fish can swim backwards.")
```

We have added the methods swim() and swim_backwards() to the Fish class, so that every subclass will also be able to make use of these methods.

Since most of the fish we'll be creating are considered to be bony fish (as in they have a skeleton made out of bone) rather than cartilaginous fish (as in they have a skeleton made out of cartilage), we can add a few more attributes to the_init_() method:

Exercise 4: Extends the previous fish class with skeleton as bone or cartilage.

```
class Fish:
    def   _init__(self,    first_name,   last_name="Fish",   skeleton="bone",
eyelids=False):
        self.first_name = first_name
        self.last_name = last_name
        self.skeleton = skeleton
        self.eyelids = eyelids

    def swim(self):
        print("The fish is swimming.")

    def swim_backwards(self):
        print("The fish can swim backwards.")
```

Student Name: _____ **Roll No:** _____ **Section:** _____

Building a parent class follows the same methodology as building any other class, except we are thinking about what methods the child classes will be able to make use of once we create those.

4. Child Classes

Child or subclasses are classes that will inherit from the parent class. That means that each child class will be able to make use of the methods and variables of the parent class.
For example, a Goldfish child class that subclasses the Fish class will be able to make use of the swim() method declared in Fish without needing to declare it. We can think of each child class as being a class of the parent class. That is, if we have a child class called Rhombus and a parent class called Parallelogram, we can say that a Rhombus is a Parallelogram, just as a Goldfish is a Fish.
The first line of a child class looks a little different than non-child classes as you must pass the parent class into the child class as a parameter:

class Trout(Fish):
The Trout class is a child of the Fish class. We know this because of the inclusion of the word Fish in parentheses. With child classes, we can choose to add more methods, override existing parent methods, or simply accept the default parent methods with the pass keyword, which we'll do in this case:

```
class Trout(Fish):
    pass
```

We can now create a Trout object without having to define any additional methods.

Exercise 5: Extending the Fish class create a class of Trout which is child to its parent class (Fish).

```
class Trout(Fish):
    pass

terry = Trout("Terry")
print(terry.first_name + " " + terry.last_name)
print(terry.skeleton)
print(terry.eyelids)
terry.swim()
terry.swim_backwards()
```

We have created a Trout object terry that makes use of each of the methods of the Fish class even though we did not define those methods in the Trout child class. We only needed to pass the value of "Terry" to the first_name variable because all of the other variables were initialized.

Student Name: _____ **Roll No:** _____ **Section:** _____

Next, let's create another child class that includes its own method. We'll call this class Clownfish, and its special method will permit it to live with sea anemone:

Exercise 6: Add another class Clownfish inherited from Fish class.

```
class Clownfish(Fish):

    def live_with_anemone(self):
        print("The clownfish is coexisting with sea anemone.")
```

The output shows that the Clownfish object casey is able to use the Fish methods_init_() and swim() as well as its child class method of live_with_anemone().
If we try to use the live_with_anemone() method in a Trout object, we'll receive an error:
Output
terry.live_with_anemone()
AttributeError: 'Trout' object has no attribute 'live_with_anemone'
This is because the method live_with_anemone() belongs only to the Clownfish child class, and not the Fish parent class.

Child classes inherit the methods of the parent class it belongs to, so each child class can make use of those methods within programs.

5. Overriding Parent Methods

So far, we have looked at the child class Trout that made use of the pass keyword to inherit all of the parent class Fish behaviors, and another child class Clownfish that inherited all of the parent class behaviors and also created its own unique method that is specific to the child class. Sometimes, however, we will want to make use of some of the parent class behaviors but not all of them. When we change parent class methods we override them.

When constructing parent and child classes, it is important to keep program design in mind so that overriding does not produce unnecessary or redundant code.

We'll create a Shark child class of the Fish parent class. Because we created the Fish class with the idea that we would be creating primarily bony fish, we'll have to make adjustments for the Shark class that is instead a cartilaginous fish. In terms of program design, if we had more than one non-bony fish, we would most likely want to make separate classes for each of these two types of fish.

Student Name: _____ **Roll No:** _____ **Section:** _____

Sharks, unlike bony fish, have skeletons made of cartilage instead of bone. They also have eyelids and are unable to swim backwards. Sharks can, however, maneuver themselves backwards by sinking.
In light of this, we'll be overriding the__init__() constructor method and the swim_backwards()method. We don't need to modify the swim() method since sharks are fish that can swim. Let's take a look at this child class:

Exercise 7: Create a class Shark inherited from Fish class and now perform overridden method to its_init_() method and change the last_name to Shark.

```
class Shark(Fish):
    def_init_(self, first_name, last_name="Shark",
                skeleton="cartilage", eyelids=True):
        self.first_name = first_name
        self.last_name = last_name
        self.skeleton = skeleton
        self.eyelids = eyelids

    def swim_backwards(self):
        print("The shark cannot swim backwards, but can sink backwards.")

sammy = Shark("Sammy")
print(sammy.first_name + " " + sammy.last_name)
sammy.swim()
sammy.swim_backwards()
print(sammy.eyelids)
print(sammy.skeleton)
```

We have overridden the initialized parameters in the_init_() method, so that the last_name variable is now set equal to the string "Shark", the skeleton variable is now set equal to the string "cartilage", and the eyelids variable is now set to the Boolean value True. Each instance of the class can also override these parameters.

The method swim_backwards() now prints a different string than the one in the Fish parent class because sharks are not able to swim backwards in the way that bony fish can.
We can now create an instance of the Shark child class, which will still make use of the swim() method of the Fish parent class:

The Shark child class successfully overrode the_init_() and swim_backwards() methods of the Fish parent class, while also inheriting the swim() method of the parent class.

Student Name: _____ **Roll No:** _____ **Section:** _____

When there will be a limited number of child classes that are more unique than others, overriding parent class methods can prove to be useful.

6. The super() Function

With the super() function, you can gain access to inherited methods that have been overwritten in a class object.

When we use the super() function, we are calling a parent method into a child method to make use of it. For example, we may want to override one aspect of the parent method with certain functionality, but then call the rest of the original parent method to finish the method.

In a program that grades students, we may want to have a child class for Weighted_grade that inherits from the Grade parent class. In the child class Weighted_grade, we may want to override a calculate_grade() method of the parent class in order to include functionality to calculate a weighted grade, but still keep the rest of the functionality of the original class. By invoking the super() function we would be able to achieve this.

The super() function is most commonly used within the__init__() method because that is where you will most likely need to add some uniqueness to the child class and then complete initialization from the parent.

To see how this works, let's modify our Trout child class. Since trout are typically freshwater fish, let's add a water variable to the_init__() method and set it equal to the string "freshwater", but then maintain the rest of the parent class's variables and parameters:

Exercise 8:

```
class Trout(Fish):
    def_init_(self, water = "freshwater"):
        self.water = water
        super()._init__(self)
```

We have overridden the __init__() method in the Trout child class, providing a different implementation of the __init__() that is already defined by its parent class Fish. Within the __init_() method of our Trout class we have explicitly invoked the_init_() method of the Fishclass.

Student Name: _____ **Roll No:** _____ **Section:** _____

Because we have overridden the method, we no longer need to pass first_name in as a parameter to Trout, and if we did pass in a parameter, we would reset freshwater instead. We will therefore initialize the first_name by calling the variable in our object instance.

Now we can invoke the initialized variables of the parent class and also make use of the unique child variable. Let's use this in an instance of Trout:

```
terry = Trout()

# Initialize first name
terry.first_name = "Terry"

# Use parent_init_() through super()
print(terry.first_name + " " + terry.last_name)
print(terry.eyelids)

# Use child_init_() override
print(terry.water)

# Use parent swim() method
terry.swim()
```

The output shows that the object terry of the Trout child class is able to make use of both the child-specific__init__() variable water while also being able to call the Fish parent__init__() variables of first_name, last_name, and eyelids.

The built-in Python function super() allows us to utilize parent class methods even when overriding certain aspects of those methods in our child classes.

7. Multiple Inheritance

Multiple inheritance is when a class can inherit attributes and methods from more than one parent class. This can allow programs to reduce redundancy, but it can also introduce a certain amount of complexity as well as ambiguity, so it should be done with thought to overall program design.
To show how multiple inheritance works, let's create a Coral_reef child class than inherits from a Coralclass and a Sea_anemone class. We can create a method in each and then use the pass keyword in the Coral_reef child class:

Student Name: _____ **Roll No:** _____ **Section:** _____

```
class Coral:

    def community(self):
        print("Coral lives in a community.")


class Anemone:

    def protect_clownfish(self):
        print("The anemone is protecting the clownfish.")


class CoralReef(Coral, Anemone):
    pass
```

The Coral class has a method called community() that prints one line, and the Anemone class has a method called protect_clownfish() that prints another line. Then we call both classes into the inheritance tuple. This means that Coral is inheriting from two parent classes.
Let's now instantiate a Coral object:

```
great_barrier = CoralReef()
great_barrier.community()
great_barrier.protect_clownfish()
```

The object great_barrier is set as a CoralReef object, and can use the methods in both parent classes. When we run the program, we'll see the following output:

Student Name: _____ **Roll No:** _____ **Section:** _____

Programming Exercise (Python)

Task 1: Discuss in detail what you understand by inheritance, Multi-inheritance, Multilevel inheritance and Super(), for all the exercises and tasks starting from task 2.

Task 2: Create class and sub classes for different types of frequent airline travelers with different connecting flights. Use concept of Multiple inheritance and Super().

Task 3: Create class and sub classes for differ types of umbrella, they have different styles, prints, sizes, for male, female, kids, they have different usage such as only for rain, for sun protection for snow. Etc. Use the concept of Multiple inheritance and supper() where necessary.
Classes, use case first then process.