Aim and Objectives

The aim of this lab is to provide students with a solid understanding of **Object-Oriented Programming (OOP)** in Python, focusing on the creation and use of **classes** and **objects**, along with methods and attributes.
After successful completion of this lab, students should be able to:

- Demonstrate a clear understanding of how to define and instantiate **classes** and **objects** in Python.
- Implement Python programs that model simple real-world entities using basic **OOP** principles.
- Write and utilize methods to manipulate object data and represent object behavior in Python.
- Design simple programs that allow users to create, update, and interact with class objects based on given scenarios.

1 What is Object Oriented Programming?

Object-Oriented Programming (OOP) is a way of writing programs that tries to map things from the **real world** into code. In the real world, everything around you can be seen as an **object**; your phone, laptop, a car, or even a student.

Each of these objects has certain **properties** and **behaviors**. Think about your mobile phone; it could have properties such as the brand, model, color, etc., and behaviors such as the ability to send text messages, make calls, and browse the web.

In OOP, we use **classes** to represent these real-world objects in code. A class is like a ***blueprint*** that describes what an object is and what it can do. Then, we can create specific **objects** from that blueprint.

An example of creating different mobile phone objects using a mobile phone class is shown in Figure 1.

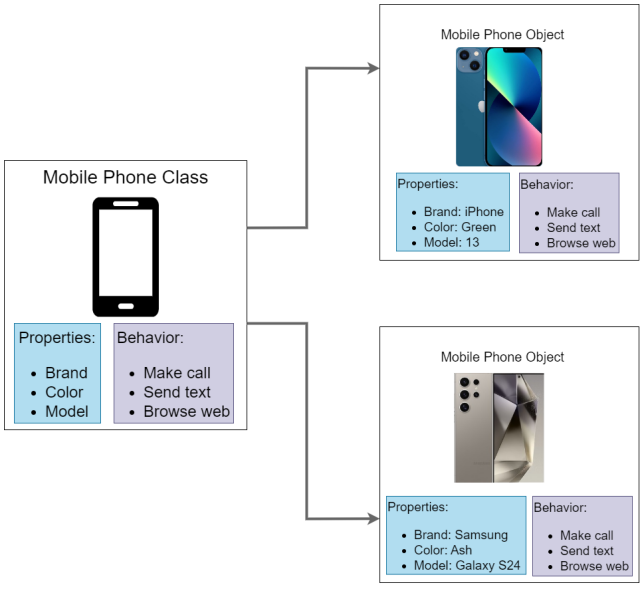

Figure 1: Creating different mobile phone objects using a mobile phone class

2 Classes in Python

Python supports object oriented programming, where we can create a **class** which is a blueprint for creating objects, along with its **attributes** (properties) and **methods** (behaviors).

1. Lets create a class named **Point** which has the **attributes** of a 2D Cartesian Point (x,y). The coding convention for naming Python classes is to use **upper camel case** for classnames, i.e: class names begin with a capital letter.

   We use the following syntax to define a **class**. At the initiation, we have to provide the **initial attributes**. Here, we use the origin(0,0) as the initial location.

```
class Point:
    x = 0
    y = 0
```

2. If we want to use the **class**, first we have to create an **object** from it.

```
point_1 = Point()
#Show initial value in x (0)
print(point_1.x)

#Show initial value in y (0)
print(point_1.y)
```

3. The examples above are classes and objects in their simplest form, and are not really useful in real life applications. To understand the meaning of classes, we have to understand the built-in ***__init()__*** function.

   All classes have a function called ***__init()__***, which is always executed first when the class is being initiated. We can use the ***__init()__*** function to assign values to object properties, or perform any operations that are necessary when the object is created:

```
# Python Classes
# CO1010 Lab 06

# Create the Class
class Point:
    # Initialize attributes
    def __init__(self,x_loc,y_loc):
        self.x = x_loc
        self.y = y_loc

point1 = Point(0,0)
print(point1.x)
print(point1.y)
```

   When we are referring to a property (attribute) or a behavior (method) of an object in a class, we have to use the keyword ***self*** to refer to the current object.

4. Objects can also contain **methods**. Methods in objects are functions that belong to the object. Let us write a function named **display()** to display the location of a point.

```
# Python Class methods
# CO1010 Lab 06

# Create the Class
class Point:
    # Initialize attributes
    def __init__(self,x_loc,y_loc):
        self.x = x_loc
        self.y = y_loc

    # Display attributes
    def display(self):
        print("x:", self.x, "y:", self.y)
```

```
14
15 point1 = Point(0,0)
16 point1.display()
```

5. We can also use methods to **modify** attributes of objects. Lets implement a method called **change_loc()** to change the location providing the new coordinates.

```
1  # Python modify Class attributes
2  # CO1010 Lab 06
3
4  # Create the Class
5  class Point:
6      # Initialize attributes
7      def __init__(self,x_loc,y_loc):
8          self.x = x_loc
9          self.y = y_loc
10
11     # Display attributes
12     def display(self):
13         print("x:", self.x, "y:", self.y)
14
15     # Change attributes
16     def change_loc(self, x_new, y_new):
17         self.x = x_new
18         self.y = y_new
19
20 point1 = Point(0,0)
21 point1.display()
22 point1.change_loc(5,4)
23 point1.display()
```

6. Let's write another example. **EngStudent** class represents a student in the Engineering faculty. For this task, lets assume a student only has an E-Number and a name.

```
1  # Python Class
2  # CO1010 Lab 06
3
4  # Create the Class
5  class EngStudent:
6      # Initialize attributes
7      def __init__(self,stu_name,e_no):
8          self.name = stu_name
9          self.e_num = e_no
10
11     # Display attributes
12     def print_student_info(self):
13         print("Name:",self.name,", E-No:",self.e_num)
14
15     # Method for changing the name of a student
16     def change_name(self,new_name):
17         self.name = new_name
18
19      # Method for changing the E-No of a student
20     def change_enum(self,new_enum):
21         self.e_num = new_enum
```

7. Let's use the defined class to create a **EngStudent** objects.

```
1
2  student_1 = EngStudent("Chathura Gamage","E/22/107")
3
4  student_2 = EngStudent("Yasitha Rajapaksha","E/22/274")
5
6  # print student information
7  student_1.print_student_info()
8  student_2.print_student_info()
9
10 # change the E-No of student 1
```

```
student_1.change_enum("E/22/206")

# print student information
student_1.print_student_info()
```

8. We can also create a Python **list** to hold information of a list of **EngStudent objects**.

```

# Student dictionary holding names as keys and E-numbers as values
students = {
    "Isuru Pamuditha": "E/22/206",
    "Thilina Dissanayake": "E/22/001",
    "Shakthi Perera": "E/22/181"
}

# EngStudent objects list
eng_stu_list = []

# Create the list using the dictionary
for name, enum in students.items():
    eng_stu_list.append(EngStudent(name, enum))

# Check the type of the elements in the list
print("Type of elements in list ", type(eng_stu_list[0]))

# Print the objects in the list
for student in eng_stu_list:
    student.print_student_info()
```

9. More on OOP concepts can be found in the following link.

3 Practice Exercises

1. Implement a class named **ComplexNum** to store a complex number. Your class should support the following functionalities:

```
class ComplexNum:
    # Initialize attributes
    def __init__(self,real,imaginary):
        self.real = real
        self.img = imaginary

    # Change the real part
    def change_real(self,new_real):
    ------------------------------

    # Change the imaginary part
    def change_img(self,new_img):
    ----------------------------

    # Get the real part
    def get_real(self):
        return self.real

    # Get the imaginary part
    def get_img(self):
    -----------------

    # Calculate the absolute value of the complex number
    def absolute(self):
    ------------------

    # Display the complex value as a string (x + yi)
    def to_string(self):
    -------------------
```

2. Write a separate function named ***complex_add()*** which takes two complex numbers and return the addition (*Remember, the addition should also be a complex number*).

```
def complex_add(cx_num1,cx_num2):
--------------------------------

# Create two complex numbers
comp_n1 = ComplexNum(-1,2)
comp_n2 = ComplexNum(3,0)

# Compute and print the addition
comp_n3 = complex_add(comp_n1,comp_n2)
print(comp_n3.to_string())
```

```
OUTPUT
2 + 2i
```

3. Write a function named ***complex_mul()*** which takes two complex numbers and return the multiplication.

Lab Tasks

Each student is required to complete only **one question** based on their assigned group as follows:

- **Group A:** Library Book Management
- **Group B:** Gym Membership Management
- **Group C:** E-Wallet Management
- **Group D:** Parking Spot Management

Refer to the respective question and follow the provided instructions to complete your task.
After completion, show the code to an instructor, and get it marked within the lab hours.
Make sure to include meaningful comments in your Python scripts.

While peer-learning is highly encouraged in our labs, copying someone else's codes will earn you zero marks for all lab exercises.

Group A: Library Book Management

1. Write a Python class named **Book** to represent a book in a library. Assume that the library only holds the title of the book, author, and the number of available copies (non-negative value).

```
class Book:
    # Initialize the attributes
    def __init__(self, title, author, copies):
    ----------------------------------

    # Display the attributes
    def get_info(self):
    ------------------------------------

    # Method to borrow a book if sufficient number of copies are available. Reduce the number
     of copies by one if borrowing is possible.
    def borrow(self, copies):
    ------------------------------------

    # Method to return a book to the library. Increase the number of copies by one.
    def return_book(self, copies):
    ------------------------------------

```

```
book = Book("To Kill a Mockingbird", "Harper Lee", 5)

book.get_info()
book.borrow(2)
book.get_info()
book.borrow(10)
book.get_info()
book.return_book(2)
book.get_info()
```

```
OUTPUT:
Book Title: To Kill a Mockingbird, Author: Harper Lee, Available Copies: 5
Book Title: To Kill a Mockingbird, Author: Harper Lee, Available Copies: 3
ERROR: Not enough copies to borrow.
Book Title: To Kill a Mockingbird, Author: Harper Lee, Available Copies: 3
Book Title: To Kill a Mockingbird, Author: Harper Lee, Available Copies: 5
```

2. Create a function named **compare_author()** which compares the authors of two Book objects and prints a message if they are the same.

```
def compare_authors(source_book, target_book):
    ----------------------------------------

book1 = Book("To Kill a Mockingbird", "Harper Lee", 5)
book2 = Book("Go Set a Watchman", "Harper Lee", 10)
book3 = Book("1984", "George Orwell", 8)

compare_authors(book1, book2)
compare_authors(book1, book3)
```

```
OUTPUT:
Both books are written by the same author: Harper Lee
Books are written by different authors: 'Harper Lee' and 'George Orwell'
```

Group B: Gym Membership Management

1. Write a Python class named **Member** to represent a gym member's account. Assume that the gym account only holds the name of the member, gym identification number, and the number of remaining sessions (non-negative value).

```
class Member:
    # Initialize the attributes
    def __init__(self, name, id_num, sessions):
    -----------------------------------

    # Display the attributes
    def get_info(self):
    -----------------------------------

    # Method to increase the number of sessions.
    def add_sessions(self, sessions):
    -----------------------------------

    # Method to reduce the number of sessions if there are sufficient number of sessions
    remaining. If number of sessions is insufficient, print an error message.
    def use_sessions(self, sessions):
    -----------------------------------

member = Member("Amali Perera", 50123, 15)

member.get_info()
member.add_sessions(5)
member.get_info()
member.use_sessions(20)
member.get_info()
member.use_sessions(5)
member.get_info()
```

```
OUTPUT:
Member Name: Amali Perera, ID Num: 50123, Remaining Sessions: 15
Member Name: Amali Perera, ID Num: 50123, Remaining Sessions: 20
Member Name: Amali Perera, ID Num: 50123, Remaining Sessions: 0
ERROR: Insufficient sessions.
Member Name: Amali Perera, ID Num: 50123, Remaining Sessions: 0
```

2. Create a function named **transfer_sessions()** which transfers sessions from one member to another. Sessions cannot be transferred if the source member doesn't have enough sessions remaining.

```
def transfer_sessions(source_member, target_member, sessions):
    ------------------------------------------------------

member1 = Member("Amali Perera", 50123, 15)
member2 = Member("Sahan Wijesinghe", 4291, 10)

transfer_sessions(member1, member2, 5)

member1.get_info()
member2.get_info()
```

```
OUTPUT:
Member Name: Amali Perera, ID Num: 50123, Remaining Sessions: 10
Member Name: Sahan Wijesinghe, ID Num: 4291, Remaining Sessions: 15
```

Group C: E-Wallet Management

1. Write a Python class named **Wallet** to represent an e-wallet account of a user. Assume that the wallet only holds the name of the user, identification number, and the current balance (non-negative value).

```
class Wallet:
    # Initialize the attributes
    def __init__(self, name,id_num, balance):
    ------------------------------------

    # Display the attributes
    def get_info(self):
    ------------------------------------

    # Method to add money to the account (balance should increase).
    def add_money(self, amount):
    ------------------------------------

    # Method to withdraw money from the account if the balance is sufficient (balance should
    decrease).
    def spend_money(self, amount):
    ------------------------------------

wallet = Wallet("Ishan Perera", 12345, 5000.00)

wallet.get_info()
wallet.add_money(1000)
wallet.get_info()
wallet.spend_money(8000)
wallet.get_info()
wallet.spend_money(2000)
wallet.get_info()
```

```
OUTPUT:
Wallet Holder: Ishan Perera, ID Number: 12345, Balance: 5000.00
Wallet Holder: Ishan Perera, ID Number: 12345, Balance: 6000.00
ERROR: Insufficient balance.
Wallet Holder: Ishan Perera, ID Number: 12345, Balance: 6000.00
Wallet Holder: Ishan Perera, ID Number: 12345, Balance: 4000.00
```

2. Create a function named **transfer_money()** which transfers money from one wallet to another. Money cannot be transferred if the source wallet doesn't have enough.

```
def transfer_money(source_wallet, target_wallet, amount):
    ------------------------------------------------------

wallet1 = Wallet("Ishan Perera", 1234, 5000.00)
wallet2 = Wallet("Chamari Silva", 3124, 7000.00)

transfer_money(wallet1, wallet2, 3000)

wallet1.get_info()
wallet2.get_info()
```

```
OUTPUT:
Wallet Holder: Ishan Perera, ID Number: 1234, Balance: 2000.00
Wallet Holder: Chamari Silva, ID Number: 3124, Balance: 10000.00
```

Group D: Parking Spot Management

1. Write a Python class named **ParkingSpot** to represent a parking spot in a parking lot. Assume that the parking spot only holds the spot number and the availability status (True for available, False for occupied).

```
class ParkingSpot:
    # Initialize the attributes
    def __init__(self, spot_number, available, vehicle_num):
    ---------------------------------------------------------

    # Display the attributes
    def get_info(self):
    --------------------

    # Method to change the availability and vehicle number of a spot.
    def update_availability(self, status, vehicle_num):
    ---------------------------------------------------

spot = ParkingSpot(15, True, None)

spot.get_info()
spot.update_availability(False, 'QBQ-4533')
spot.get_info()
```

```
OUTPUT:
Spot Number: 15, Availability: Available, Vehicle Number: None
Spot Number: 15, Availability: Occupied, Vehicle Number: QBQ-4533
```

2. Create a function named `swap_availability()` that compares the status and vehicles in two spots and swaps them.

```
def swap_availability(spot1, spot2):
-------------------------------------

spot1 = ParkingSpot(15, True, None)
spot2 = ParkingSpot(16, False, 'ABW-1293')

swap_availability(spot1, spot2)

spot1.get_info()
spot2.get_info()
```

```
OUTPUT:
Spot Number: 15, Availability: Occupied, Vehicle Number: ABW-1293
Spot Number: 16, Availability: Available, Vehicle Number: None
```

Bonus Task: Creating a Game with Combat Simulation

*While this task completion is **optional**, if you complete the task and get it marked by an instructor, you will earn **extra marks** for the labs.*

1. Create a Python class named **Character** to represent a character in the game. Each character should have the following attributes:
   - name: Name of the character.
   - weapon: Weapon of the character.
   - energy: Energy level of the character.
   - life: Life level of the character.
   - x, y: Coordinates to represent the character's position on a 1000x1000 grid.
2. Add the below methods to manipulate the attributes:

```
class Character:
    # Initialize the attributes
    def __init__(self, name, weapon, energy, life):
    ------------------------------------------------

    # Method to change the character's name
    def change_name(self, new_name):
    --------------------------------

    # Method to change the character's weapon
    def change_weapon(self, new_weapon):
    -----------------------------------

    # Increase the character's energy based on the type of food consumed. Use a
    dictionary to map food types to energy levels. When the energy level reaches 100,
    increase the life by one.
    def increase_energy(self, food_type):
    -------------------------------------

    # Decrease the character's energy based on the number of shots taken. When the
    energy level decreases to zero, decrease the life by one.
    def decrease_energy(self, shots):
    ---------------------------------

    # Move left, right, forward, or backward by 10 units within the game space limited
    to a 1000 x 1000 square unit area.
    def move_left(self):
```

```
    ---------------------

    def move_right(self):
    ---------------------

    def move_forward(self):
    ---------------------

    def move_backward(self):
    ---------------------
```

3. Add the following two methods to simulate interactions between two characters:

```
# Attack another character, randomly decreasing their energy.
# (Hint: use decrease_energy(shots) function)
def attack(source, target):
---------------------------

# Use your creativity to simulate combat between two selected characters.
def combat(source, opponent):
-----------------------------
```

4. Create a Python class named **Game**. This class should manage a list of Character objects.

```
class Game:
    # Initialize an empty list to store characters
    def __init__(self):
    --------------------

    # Method to add a character to the game
    def add_character(self, character):
    -----------------------------------

    # Method to view all characters in the game
    def view_characters(self):
    --------------------------

    # Method to select a character from the list based on its index
    def select_character(self, index):
    ----------------------------------
```

5. Create a **Game** containing **5 Character objects** and simulate a **combat between two** of the Characters.

Submission

Please follow these steps for submission:
1. Complete the tasks and save all your Python scripts.
2. Compress all scripts into a single ZIP folder.
3. Rename the ZIP folder to `E22xxx.ZIP`, where `xxx` represents the last three digits of your student registration number.
4. Submit the ZIP folder to the link provided on the FEeLS course page.

Steps for Creating a ZIP File

If you are unfamiliar with how to create a ZIP file, please follow the steps below:
1. Select the all Python scripts.
2. Right-click on one of the selected files.
3. Choose **Send to → Compressed (zipped) folder**.
4. A new ZIP folder will appear. Rename this folder to `E22xxx.ZIP`, replacing `xxx` with the last three digits of your registration number.