

Programming Foundations in Python

Prepared by Dr. Hamada I. AbdulWakel

Lecture 8



The background features a series of vertical panels with rounded ends. The leftmost panel shows a person's hands typing on a laptop. The subsequent panels contain various white line-art icons on a semi-transparent background: a graduation cap, a play button, a target with an arrow, an open book, a magnifying glass, and a group of people. The main title is positioned to the right of these panels.

Object-Oriented Programming

Lecture 8

Prepared by
Dr. Hamada I. AbdulWakel

Lecture

08

RECAP: LECTURE 7

- The function default arguments.
- The inner and high-order functions.
- The anonymous functions
- The use of modules.

Lecture

08

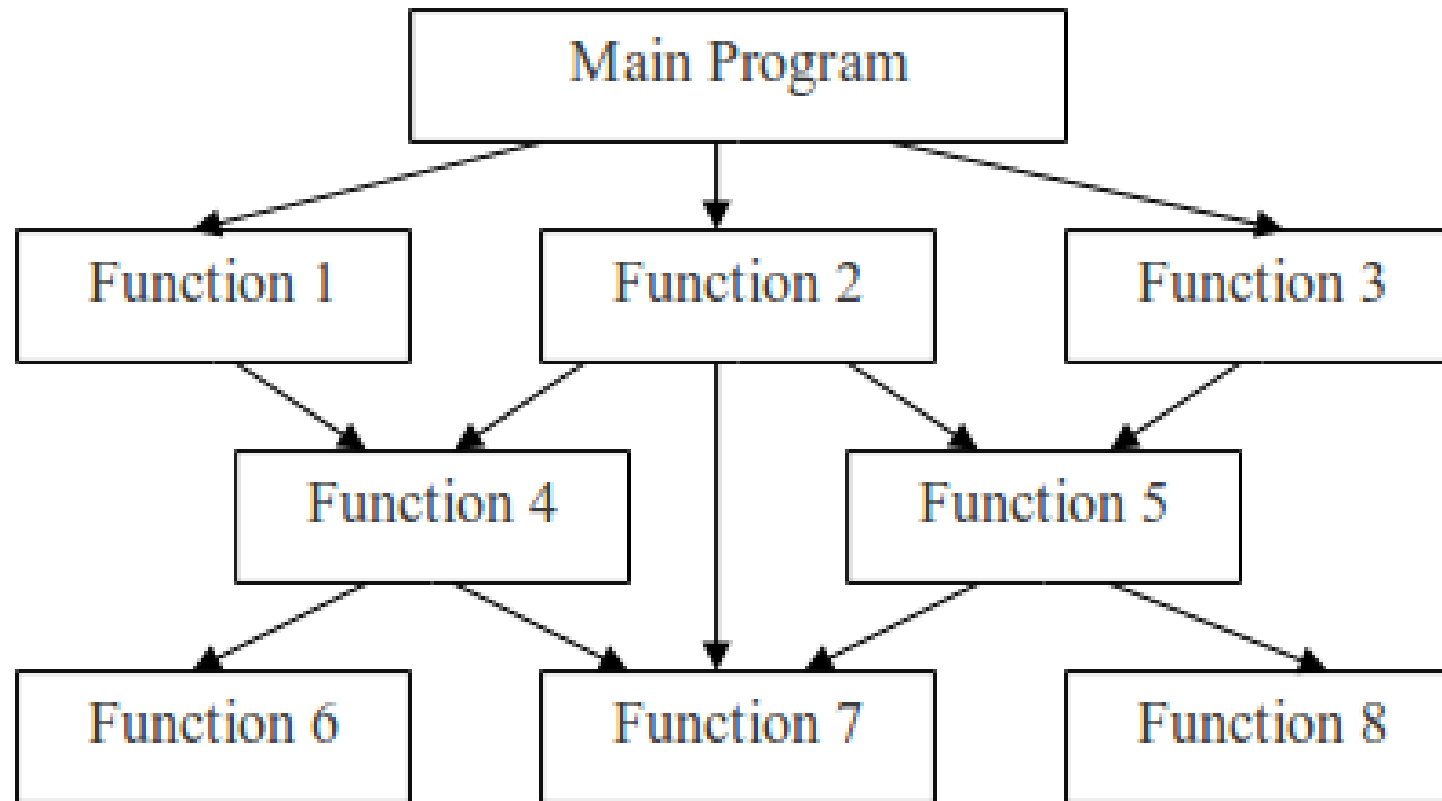
LECTURE MAIN POINTS

- Understanding the Procedural-Oriented Programming.
- The meaning of Object-Oriented Programming.
- The building blocks and principles of OOP.
- Difference between a constructor, instance variables, and methods for a class of objects.

Lecture

08

PROCEDURAL-ORIENTED PROGRAMMING (POP)



Lecture

08

POP: FEATURES

- The main concern in POP is **doing things**.
- Depend on the built-in and user-defined **functions**.
- Based on local and global variables in parameter passing.
- **Modularity**: means splitting the program into parts where the top-down design can be applied.

Lecture

08

POP: ADVANTAGES AND DISADVANTAGES

- Advantages:
 - Excellent for general-purpose programming.
 - The coded simplicity along with ease of implementation of compilers and interpreters
 - The program flow can be tracked easily.
- Disadvantages:
 - Poor modeling to real world problems.
 - It is not easy to maintain program if it is too complex.
 - Provide poor data security.
 - Not extensible programming paradigm.

Lecture

08

PRIMITIVE DATA TYPES

- **Primitive data types**, a.k.a. basic or fundamental data types, are the simplest and most basic building blocks in a programming language. It store single simple value.
- The primitive data types vary between programming languages, but some common ones include:
 - Integer, Short, and Long.
 - Floating-point, Double.
 - Character.
 - Boolean.
 - String.

Lecture

08

PRIMITIVE DATA TYPES PROBLEMS

- **Student Management System Case Study**
 - Suppose you want to store data for **5 students** for an academic institution.
 - Each student has name, ID, and level information.
 - How to use primitive data types to solve this?

```
stds_names = ["a", "b", "c", "d", "a"]  
stds_IDs   = [ 1,  2,  3,  4,  5 ]  
stds_levels = [ 1,  2,  3,  4,  1 ]
```

```
print("Student information are: ", stds_names[4], stds_IDs[0], stds_levels[0])
```

```
std1_name = "a"  
std1_ID   = 1  
std1_level = 1  
std2_name = "b"  
std2_ID   = 2  
std2_level = 2  
std3_name = "c"  
std3_ID   = 3  
std3_level = 3  
std4_name = "d"  
std4_ID   = 4  
std4_level = 4  
std5_name = "a"  
std5_ID   = 5  
std5_level = 1
```

PRIMITIVE DATA TYPES PROBLEMS

Cont'd

- **Student Management System Case Study**
 - Now, the academic institution has **100 registered students**.
 - **Each student** has name, ID, address, email, level, and registered courses information.
 - **Each course** has course name, course ID, course time, and course level information.
 - Primitive data types were not a problem for simple programs. But, for large scale and complex program it is not suitable at all.

Lecture

08

COMPLEX/CUSTOM DATA TYPES

- **Custom data types**, a.k.a. user-defined data types, allow developers to create structures that suitable for the specific needs of their applications.
- They offer several advantages in terms of **organization, abstraction, and functionality**.
- Complex data types help in representing **real world things/objects**.

Lecture

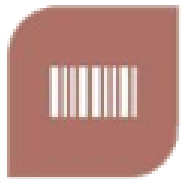
08

COMPLEX/CUSTOMER DATA TYPES

Cont'd



Customer



Product



Post



Name, address, payment method



Name, **description**, is it in stock?



User, **post text**, date

Lecture

08

OBJECT-ORIENTED PROGRAMMING

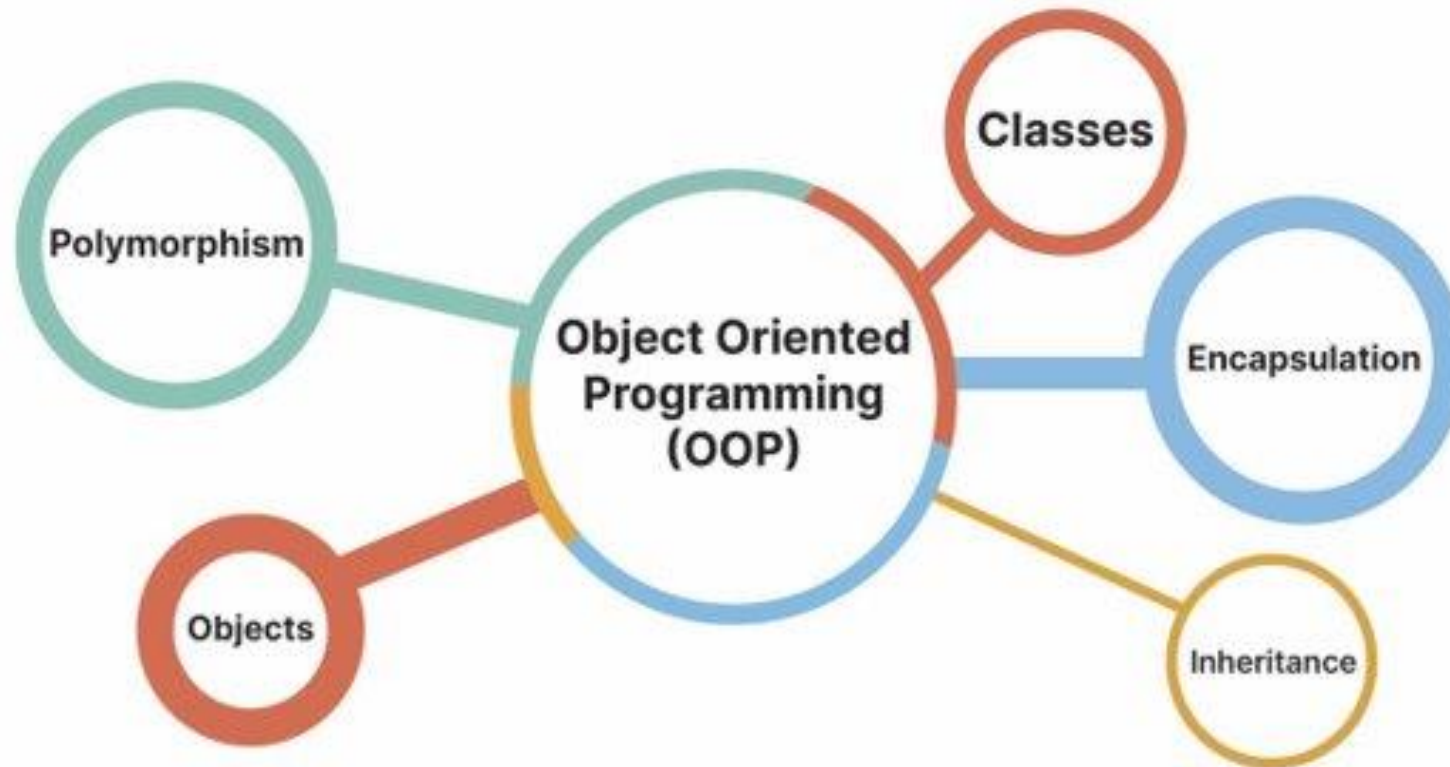
- Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects, which can encapsulate data and behavior.
 - Program is divided into classes and objects.
 - The main concern on **data**.
 - Strong modeling to real world problems.
 - It is easy to maintain program even if it is too complex.
 - Provide strong data security.
 - It is highly extensible programming paradigm.
 - Unit of programming is **class**.
 - Ex: Python, Java, C#.



Lecture

08

OOP: KEY CONCEPTS



Lecture

08

OOP: CLASS

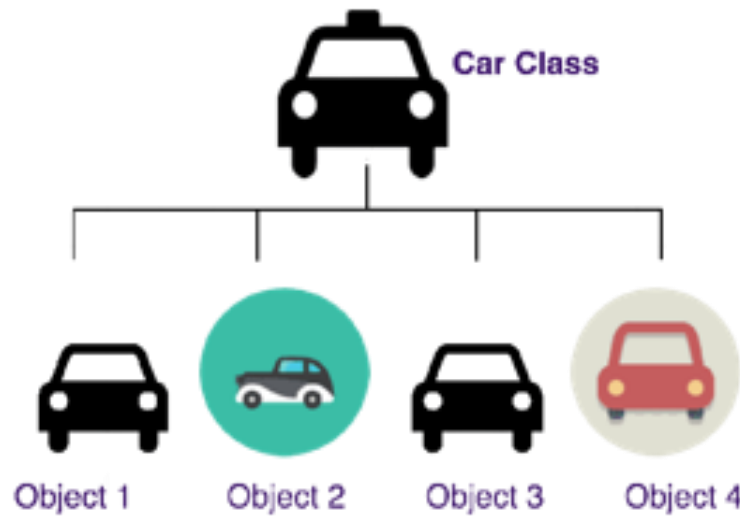
- **Classes:** the user-defined data types that act as the blueprint for individual objects, attributes and methods.
 - **Attributes:** are defined in the class template and represent the state of an object. Class attributes belong to the class itself.
 - **Methods:** are functions that are defined inside a class that describe the behaviors of an object.

Lecture

08

OOP: OBJECT

- **Objects:** are **instances** of classes and represent real-world entities. They encapsulate data (attributes) and behavior (methods) related to those entities.

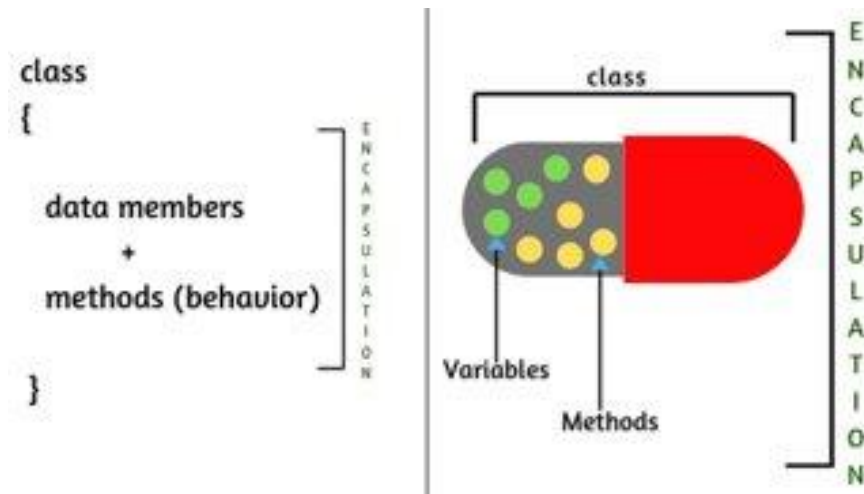


Lecture

08

OOP: ENCAPSULATION

- **Encapsulation** is the bundling of data (attributes) and the methods (functions) that operate on that data into a single unit (class). It restricts access to the internal details of an object, providing a protective barrier around the object's state.



Lecture

08

OOP: INHERITANCE AND POLYMORPHISM

- **Inheritance:**
 - Allows a class to inherit the properties and behaviors of another class.
 - This promotes code **reuse and the creation of a hierarchy of classes**.
- **Polymorphism:** Objects are designed to share behaviors (methods) and they can take on **more than one form**.

Lecture

08

CLASS DEFINITION

- All class definitions start with the `class` keyword, which is followed by the name of the class and a **colon**.
- Any code that is **indented** below the class definition is considered part of the class's body.

```
class className :  
    #class body
```

CLASS DEFINITION

Cont'd

- Example: empty class

```
class Student:  
    pass          #indicating where code will eventually go  
  
print(Student)
```

Lecture

08

INSTANCE ATTRIBUTES

- Instance attributes are the **properties** that all objects must have and defined inside the method called `__init__()`.
- When a new object is created, `__init__()` sets **the initial state** of the object by assigning the values of the object's properties.
- If you change the attributes of an instance, it won't affect other instances.
- The `__init__()` can take any number of parameters, but the first parameter will always be a variable called **self**.

INSTANCE ATTRIBUTES

Cont'd

- Example:

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

- Both **name** and **age** are called instance attributes.

Lecture

08

CLASS ATTRIBUTES

- The class attributes are properties that have **the same value** for all class instances.
- Class attributes are defined directly **under the line of the class name**, outside the `__init__()` function.
- You can access the class attribute via **instances** of the class or via the **class name**.

CLASS ATTRIBUTES

Cont'd

- Example:

```
class Student:  
    course = "OOP in Python"  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

- The **course** property is called class attribute and has the "OOP in Python" value for every student object.

Lecture

08

DEALING WITH INSTANCE AND CLASS ATTRIBUTES

- When you access an attribute via an **instance of the class**, Python searches for the attribute in **the instance attribute list**.
- If the instance attribute list doesn't have that attribute, Python continues looking up the attribute in **the class attribute list**.
- However, if you access an attribute via **the class name**, Python directly searches for the attribute in **the class attribute list**.

DEALING WITH INSTANCE AND CLASS ATTRIBUTES

Cont'd

- Example:

```
class Student:  
    age = 10  
    def __init__(self):  
        self.age = 20
```

```
test = Test()  
print(test.age) # 20  
print(Test.age) # 10
```

Lecture

08

INSTANTIATING OBJECTS

- You can instantiate a new object by typing the name of the class, followed by opening and closing parentheses.

`ObjectName = ClassName( [arguments if exist] )`

- After you create the instances, you can access their instance attributes using **dot notation**.
- The custom objects are **mutable** by default.
- You can delete the instance or its attribute using the keyword **del**.

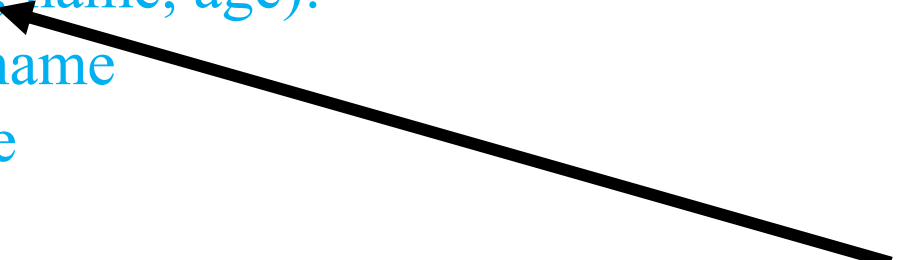
INSTANTIATING OBJECTS

Cont'd

- Example:

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

`std1 = Student("Ahmed", 20)` → The object **std1** is automatically passed where `self` is referring to the object **std1**



INSTANTIATING OBJECTS

Cont'd

- Example:

```
std1 = Student("Ahmed", 20)
std2 = Student("Mohamed", 15)
print(std1 is std2)
print(std1.name)
std1.age = 25
print(std1.age)
del std1.age
print(std1.age)
del std1
```

Lecture

08

INSTANCE METHODS

- Methods are functions defined **inside** a class and can only be called/accessed from an instance of that class.
- Like `__init__()`, an instance method's first parameter is **always the self** argument.
- When there are two instance methods with **the same name** and **different implementation**, the second version of the method overridden the first one when calling it.

INSTANCE METHODS

Cont'd

- Example:

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def displayStudentInfo(self):
        return f'{self.name} is {self.age} years old'

s1 = Student("Ahmed", 15)
print(s1.displayStudentInfo())
```

INSTANCE METHODS

Cont'd

- Example:

```
class Student:
    def display(self):
        print(self.name)
    def display(self):
        return f" The student's name is {self.name}"

s1 = Student("Ahmed", 15)
print(s1.display())
```

Lecture

08

SPECIAL INSTANCE METHODS

- This kind of methods overriding the functionality implementation of a built-in functions in Python to be suitable for custom classes.
- Like `__init__` method, a special instance method is always beginning and ending with **double underscores** `__`.
- The `__str__()` method is used to change what printed when passing an instance to the `print()` method.
- The `__len__()` method which is called when the built-in function `len()` is called, with the object being the argument.

SPECIAL INSTANCE METHODS

Cont'd

- Example:

```
class Student:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return f"The student name is {self.name}"
    def __len__(self):
        return len(self.name)
s1 = Student("Ahmed")
print(s1); print( len(s1) )
```

Lecture

08

EXAMPLE

- Create a class called "**Distance**" that has two instance attributes called "**feet**" and "**inches**" and has the following methods:
 - `__init__()`: that initializes the distance objects using the default arguments.
 - `getDistance()`: that takes the distance information from the user.
 - `addDistances()`: that adds two different distance and return the sum of their feets and inches.

EXAMPLE

Cont'd

- Code:

```
class Distance:
    def __init__(self, feet = 0, inch = 0.0):
        self.feet = feet
        self.inch = inch
    def getDistance(self):
        f = int(input("Enter the distnace feet value: "))
        self.feet = f
        inc = float(input("Enter the distnace feet value: "))
        self.inch = inc
```

EXAMPLE

Cont'd

- Code:

```
class Distance:
    def addDistances(self, otherObj):
        temp = self.__class__()
        temp.feet = self.feet + otherObj.feet
        temp.inch = self.inch + otherObj.inch
        if temp.inch >= 12.0:
            temp.feet += 1
            temp.inch -= 12.0
        return temp
```

EXAMPLE

Cont'd

- Code:

```
d1 = Distance()
print(d1.feet)
d1.getDistance()
print(d1.feet)
d2 = Distance(2, 1.5)
d3 = d1.addDistances(d2)
print(d3.feet, d3.inch)
```

Lecture

08

QUESTIONS FOR ONLINE DISCUSSION

- Write a class, Student, which has three properties, including a 6-digit student ID, a name, and a list of exam scores. The constructor takes the student name as the first argument, followed by an optional number of additional arguments representing the examination scores. The Student class provide two methods, including average, which takes no argument and returns the average score of the student, and __str__, which formats the student information as '[ID\t name\t scores]', where scores is a tab-seperated list of floating-point numbers, accurate to two digits after the decimal point.



Thank You

