UCL

INST0004 Programming 2

Lecture 04: Principles of Object-Oriented Programming

Module Leader:
Dr Daniel Onah

2023-24


UCL

Copyright Note

Important information to adhere to

Recap of Previous Lecture 3

Recap of previous lecture

Let's remind ourselves of last week's lecture.

- We looked at control structures in Python
- We discussed decisions - if statement
- we discussed using **flowchat** to illustrate decision making
- We discussed nested statement and conditional expression
- We looked at nested branches
- We discussed **while loop** structures
- We looked at sentinel control construct
- We looked at **for-loop** control structure

Learning Outcomes

The learning outcomes for the lecture

The objective of this week's lecture is to introduce the concept of object-oriented programming in Python. At the end of the lecture, you should be able to:

- understand the basic concepts of object-oriented programming
- understand classes creation
- understand objects creation
- understand concepts of data abstraction
- understand the concepts of constructor
- understand concepts of inheritance
- understand concepts of polymorphism
- learn various Python methods to achieve object-oriented programming paradigms

Introduction to Object-oriented Programming

Why do we need to care about OOP?

Object-oriented programming (OOP) as gain widespread acceptance among programmers in the development of software applications. Python programming language adheres to the concepts of object-oriented programming paradigms. The OOP in Python is related to the declaration of classes, objects and properties which are the core concepts of object-oriented programming. This lecture will introduce you to the various concepts of Object-oriented programming in Python such as *classes*, *objects*, *constructors*, *inheritance*, and *polymorphism*.

Introduction to Object-oriented Programming

Why do we need to care about OOP?

Object-oriented programming (OOP) is a unique approach of structuring program constructs **by grouping associated properties and functionalities into separate objects**. The core concepts of Object-oriented programming are classes and objects. A class is a blueprint for which objects are created. While objects are real entities and are able to execute or perform various actions or tasks. Data elements known also as properties constitute behaviours such as actions or functions, make up an object.

Procedural Programming

Why do we need to care about Procedural Programming?

Another popular programming paradigm is procedural programming. Procedural programming, structures a program like a **recipe** by providing a step-by-step **guide** or **instructions** such as in a *function and code blocks in a sequential order to complete a given task*. There are key difference between object-oriented programming and procedural programming:

- object-oriented programming create a modular part of a program (that is divides a program into smaller units) and this small units are known as objects. While procedural programming divides a program into small modular programs and each small program is known as a function.
- Object-oriented programming gives more importance to data rather than the **functions** or the **procedures**. While procedural programming does not concentrate or give importance to the data.

Importance of Object-oriented Programming

Why is OOP significant?

Python similar to other famous general-purpose programming languages such as C, Java etc, has been an object-oriented language. It enables us to develop applications in an object-oriented manner. **This allows us to easily build and used classes and objects in Python**. *When we use classes and objects to develop a software application product, we are using the approach of an object-oriented paradigm*. The object is then associated to real-world objects such as **computer**, **a home**, **a mobile phone** and in **general Internet-of-Things**, etc. By general principle definition, object-oriented programming paradigm emphasizes the development of modular reusable codes. This is a common method of resolving a given task by creating objects. The following are the main concepts of OOP paradigm: Class, Object, Method, Data Abstraction and Encapsulation, Inheritance and Polymorphism.

Object Class

What do we mean by objects-classes in OOP?

A set of objects can be described as a class object. An object is a logical entity in object-oriented programming with some unique field properties and methods. For example, in a Student class, this should have an **attribute** and **method**, such as name, gender, age, address, and course.

Object

What is an object?

An object is a self-contain entity with state and operations or actions. It can be an aircraft, a pen, a mobile device or anything else. However, generally in programming and in Python Programming, an object is everything with attributes and methods. When we define an object within a class, **the class must construct this object in order to assign memory to the object being defined**.

Method

What is a method?

In Python programming method is similar to a —function in behaviour. One of the main difference is that a method is associated to objects and classes. There are two major variations between a method and a function in Python programming. These are:

- The method is used implicitly for the object for which it is named or created for.
- Field or Data within the class is accessible to the method.

Encapsulation & Data Abstraction

What is encapsulation and data abstraction?

In object-oriented programming, encapsulation is very essential. Encapsulation simply means information hiding. **It is used to limit access to variables and methods**. *Encapsulation protects code and data from accidental modification or restrict code visibility from outside the class by wrapping them together in a single device.* abstraction *is a technique for hiding internal information and displaying only the functionalities of the program*. The term abstract **refers to the process of giving items names that capture the essence of what a function or a program does**. Both data abstraction and encapsulation, the terms are nearly interchangeable.

Inheritance

What is the purpose of inheritance?

The most fundamental component of object-oriented programming is inheritance, which simulates the process of inheritance in real-life scenarios. Inheritance is a phenomenon where the child-object inherits **all the features of the parent object including the properties and behaviours**. *Inheritance simply means a class can inherit all the properties, methods and behaviours of another class and still have its own unique properties*. The new class that inherit from the parent class is known as a derived class or a child class while the parent class whose properties were acquired or inherited is also known as the base class. The phenomenon of inheritance, ensures that the program code can be reused in other applications.

Facts about Inheritance

The main fact about inheritance is that the child class acquires the features of the parent class and on its own develop unique features **based on the performance or functionalities required for the program to function properly**.

Polymorphism

What is polymorphism?

Polymorphism is a term in programming that simply means the ability of a given task to be executed in different ways or forms. Polymorphism uses a single type of entity for example; a method, an operator or an object that could be presented in multiple ways in various scenarios. For example, we have an operator **add or plus sign (+)**, which can perform as **adding two integers or different kinds of values together** and the same operator can also be used to concatenate strings together.

Facts about Polymorphism

Polymorphism is made up of two words poly and morph. The word *poly* means many, multiple or more than one, and the word *morphs* stands for various forms or stages.

Object-oriented vs Procedural Programming

What are the main differences?

The following table illustrate the core difference between object oriented programming and procedural programming.

S/N	Object-oriented Programming	Procedural Programming
1	Object-oriented programming is a problem-solving method that employs the use of objects to perform a computational operation	Procedural programming uses a sequence of instructions to perform a step-by-step computational operation.
2	It is a simulation of a real-world scenario. As a result OOPs can be used to easily solve real-life or world problems	It does not simulate the real-world. It is based on a set of step-by-step instructions that broken into smaller components known as functions.

Object-oriented vs Procedural Programming

What are the main differences?

The following table illustrate the core difference between object oriented programming and procedural programming.

S/N	Object-oriented Programming	Procedural Programming
3	It makes the execution to be simple and maintains the program easily	When working with procedural programming, it can be challenging to keep track of the program codes as the project becomes larger..
4	Python, Java, C, .Net are examples of object-oriented programming languages	C, Fortran, Pascal, VB are examples of procedural languages.

Object-oriented vs Procedural Programming

What are the main differences?

The following table illustrate the core difference between object oriented programming and procedural programming.

S/N	Object-oriented Programming	Procedural Programming
5	It allows data to be encapsulated and hidden. As a result it makes the program and code safer than procedural languages. In object-oriented programming, private information cannot be access from anywhere without permission.	Procedural languages lack proper method for data binding, therefore, they are insecure.

Classes and Objects

Relationship between classes and objects

A class is a blueprint for which we create objects.When an object is created, this holds memories and becomes visible or in existence within a class. For example, lets assume a class to be an entity or virtual model for building a structure. The building or architectural structure will then comprise information about the rooms, gardens, dinning area, balcony and so on. Based on the information, several house can be developed. As a result, the **architectural structure** can be classified as a **class or a blueprint** for which other buildings could be constructed. Therefore, we can build many houses as possible using the same blueprint.

An object is said to be an instance of a class. The process of constructing an object within a class is known as instantiation. In the example above, the **individual houses developed from the blueprint are the objects of the class**.

Defining a Class

Creating and building a class

In Python a class can be defined by using s the keyword **class**, followed by the class name. The following is the **syntax** for building and creating a class:

Syntax

```
#!/usr/bin/env python
"""
A Syntax for defining and creating a class
"""
class <Class_Name>
# data members
# member functions
```

Defining a Class

Creating and building a class

A class declares all its attributes in a **new local namespace**. The data members (such as fields) or functions may be defined as **attributes**. The class also contains a unique attributes that start with double underscores __. For example, __doc__ returns the class's docstring. We can use the following ***__doc__*** return statement to access the **docstring**. When we define a class, a new class object with the same name is generated. We can use this new class object to access the various attributes of the class and may even create new objects of that class.

Defining a Class: Example

Creating and building a class

```
#!/usr/bin/env python
"""
A program to demonstrate creating a class
"""
# creating a class
class Student:
    "This is an example of a student class"
    age = 21    # declare age attribute
    def fun(self):    # defining a function
        print("INST0004 Python Programming")
print(Student.age) # display the age
print(Student.fun) # call the function
print(Student.__doc__) # display docstring
```

Defining a Class: Solution

Creating and building a class

Output

```
21
<function Student.fun at 0x102e50790>
This is an example of a student class
```

Defining a Class: Example

Creating and building a class

```
#!/usr/bin/env python
"""
Program to illustrate creating a class
"""
# Creating a student class
class Student:
    studentID = 15687293 # creating attribute of ID
    studentName = "Danny" # creating attribute of name
    def show(self): # define a function
        print("Student ID: ", self.studentID, "\n" "Student Name:", self
    .studentName)# creating the objects
# creating an object
studentObj = Student()
# calling a member function
studentObj.show()
```

Defining a Class: Solution

Creating and building a class

Output

```
Student ID:  15687293
Student Name: Danny
```

Defining a Class: Interpretation

Creating and building a class

In the above example, we created a Student class with two fields: **studentID** and **studentName**. The class also contains a function definition known as **show**(), which can illustrate the student's information.

The **self** variable is used as a reference variable to the current class object. This is always declared as the first argument within a function definition. However, using **self** in function call is optional.

The self-parameter accesses the class variables and corresponds to the current instance of the class. Instead of self, we can use anything, but it must be the first parameter to be declared within the class function.

Creating an Object

How do we create object?

In order to use any class attributes in another class or method, we must first instantiate them. We instantiate a class be invoking or calling the class name. Lets look at an example to help us understand the syntax for instantiating or creating a class instance (or object):

Syntax

```
#!/usr/bin/env python
"""
A Syntax for creating an object
"""
 <object-name> = <class-name>(<arguments>)
```

Creating an Object: Example

How do we create object?

Lets look at an example of how to create **objects** of a Book class.

```
#!/usr/bin/env python
"""
A program to demonstrate creating of a book class objects
"""
# creating a class called Book
class Book:
    # defining attributes
    title = "Learning Python Programming the Easy Way"
    author = "Daniel Onah"
    publicationYear = 2023
    # creating function
    def show(self):
        print("Book Title: ", self.title)
```

Creating an Object: Example

How do we create object?

```
14          print("Author: ", self.author)
15          print("Publication Year: ", self.publicationYear)
16  # object creation
17  Book1 = Book()
18  Book2 = Book()
19  # Accessing the book class attributes and methods using the objects
20  print(Book1.title)
21  print("----------------")
22  Book2.show()
```

Defining a Class: Solution

Creating and building a class

Output

```
Learning Python Programming the Easy Way
--------------------------------------------------------
Book Title:  Learning Python Programming the Easy Way
Author:  Daniel Onah
Publication Year:  2023
```

Deleting Object's Properties

How do we delete object and its properties? - deleting class attribute

We can delete the properties of a **class object** or the entire object using the del keyword. Lets look at an example to help us understand how to delete some **properties of objects and even the object itself**.

```
1  #!/usr/bin/env python
2  """
3  A program to demonstrate deleting object and properties
4  """
5  # Deleting properties of a class object
6  class Book:
7      # defining attributes
8      title = "Learn Python Programming the Easy Way"
9      author = "Daniel Onah"
10     publicationYear = 2023
```

Deleting Object's Properties

How do we delete object and its properties? - deleting class attribute

```
    def show(self):    # creating function
        print("Book Title: ", self.title)
        print("Author: ", self.author)
        print("Publication Year: ", self.publicationYear)
# object creation
Book1 = Book()
# Accessing class attribute and method using objects
print("-----------------------------------------------")
Book1.show() # use the object to invoke the function
# deleting a property (title)
del Book1.title
print("------------After deletion--------------------")
Book1.show()
```

Defining a Class: Solution

Creating and building a class

Output

```
Book Title:  Learn Python Programming the Easy Way
Author:  Daniel Onah
Publication Year:  2023
------------After deletion--------------------
Traceback (most recent call last):
File "<string>", line 21, in <module>
   del Book1.title
AttributeError: title
```

As we deleted the title property associated to the Book, the program throws an attribute error: title, when we try to invoke the title using the Book1 object.

Deleting Object: Example

Deleting class object

```
#!/user/bin/env python
"""
A program to demonstrate object delection
"""
# Example of object deletion
# creating a class book
class Book:
    # defining attributes
    title = "Learn Python Programming the Easy Way"
    author = "Daniel Onah"
    publicationYear = 2023
    # creating function
    def show(self):
        print("Book Title: ", self.title)
        print("Author: ", self.author)
        print("Publication Year: ", self.publicationYear)
```

Deleting Object: Example

Deleting class object

```
17 # Object creationg
18 Book1 = Book()
19 # Accessing class attribute and method using objects
20 print("---------Before Delection of Object----------------")
21 Book1.show()
22 # deleting object Book1
23 del Book1
24 print("----------------After Deletion of Object------------")
25 Book1.show()
```

Deleting Object: Solution

Creating and building a class

Output

```
---------Before Delection of Object----------------
Book Title:  Learn Python Programming the Easy Way
Author:  Daniel Onah
Publication Year:  2023
----------------After Deletion of Object------------
Traceback (most recent call last):
File "<string>", line 25, in <module>
   Book1.show()
NameError: 'Book1' is not defined. Did you mean: 'Book'?
```

As we deleted the entire object "Book1" all properties associated to the Book1 has been removed from the program, so as soon as we request to view the object in line 13, the program throws a NameError: Book1, that Book1 is no longer defined

Data abstraction

What is data abstraction?

Data abstraction is an important feature in object-oriented programming. This is the process of hiding Python data by prefixing the attribute so they could be hidden with a double underscore (__). In this case, the attribute will then be hidden from outside access through the object after performing the data abstraction on the class object. We will see an example to help us understand the concept of data abstraction better. the data abstraction allows us to hide information and make them not to be available or accessible outside the class.

Data Abstraction: Example

How do we create data abstraction class

Let's see an example of **data abstraction** in Python programming.

```
#!/usr/bin/env python
"""
A program to demonstrate the concept of data abstraction
"""
class Student:
    __n = 0;
    def __init__(self):
        Student.__n+=1 # no space in the assignment operator
    def show(self):
        print("Total Number of Students: ", Student.__n)
# creating student objects
student1 = Student()
student2 = Student()
student3 = Student()
```

Data Abstraction: Example

How do we create data abstraction class

```
15 # display the objects
16 student1.show()
17 student2.show()
18 # This will generate error because n is hidden from outside class
19 print(student1.__n)
```

Data Abstraction: Solution

How do we create data abstraction class

Output

```
Total Number of Students:  3
Total Number of Students:  3
----------------After Deletion of Object------------
Traceback (most recent call last):
File "<string>", line 19, in <module>
   print(student1.__n)
AttributeError: 'Student' object has no attribute '__n'
```

Constructors

What are constructors?

Constructor is a special type of function used to initialize the class's instance members. In **Java and C++, constructors have the same name as that of the class**. Constructors in Python are handled differently. When we create an object of a class, the constructor function executes it. There are two basic constructors in Python; parameterized constructors and non-parameterized constructors. In Python, constructors are defined or created using the __init__() method. The constructor in Python, takes the **self** keyword as a first argument, allowing access to the class's **attributes** and **methods**. Depending of the definition of the constructor (**__init__()**), we can pass any number of arguments when constructing the class object. Within the constructor body, we usually set the class attributes. In Python (and in **Programming generally**), every class must have a default constructor if one has not be defined already.

Constructors: Example

creating objects and accessing attributes

```
#/usr/bin/env python
"""
A program to create a class book using a constructor
"""
# creating a class book
class Book:
    # defining constructor
    def __init__(self, name, author, year):
        print("Constructor example") #  initializing the book attributes
        self.name = name
        self.author = author
        self.year = year
    # creating the book detail function
    def book_detail(self):
        print("Book Name: ", self.name)
        print("Author: ", self.author)
        print("Year: ", self.year)
```

Constructors: Example

creating objects and accessing attributes

```
18 # object creating and passing arguments to constructor
19 Book1 = Book("Python Programming", "Danny", 2023)
20 Book2 = Book("Java Programming", "Daniel", 2020)
21
22 # Accessing class attribute and method using the book objects
23 print("+++++++++++++++ Output are ++++++++++++++++++")
24 Book1.book_detail()
25 Book2.book_detail()
```

Constructor: Solution

creating objects and accessing attributes

Output

```
Constructor example
Constructor example
++++++++++++++ Output are ++++++++++++++++++
Book Name:  Python Programming
Author:  Danny
Year:  2023
Book Name:  Java Programming
Author:  Daniel
Year:  2020
```

Constructors: Example

creating objects and accessing attributes

```
#/usr/bin/env python
"""
A programme to demonstrate counting number of objects in a class
"""
# A program for counting number of objects in a class
class ObjectCounts:
    # Defining data member
    count = 0
    # Defining constructor
    def __init__(self):
        ObjectCounts.count += 1
# creating objects
Object1 = ObjectCounts()
Object2 = ObjectCounts()
Object3 = ObjectCounts()
# display the number of objects created
print("The number of objects created is: ", ObjectCounts.count)
```

Constructor: Solution

creating objects and accessing attributes

Output

```
The number of objects created is:  3
```

Non-parameterized Constructors

No need for value update

When we don't need to update the value or just to use only self as an argument, we use the non-parameterized constructor. Let's take a look at an example.

```
#/usr/bin/env python
"""
A program to demonstrate non-parameterized constructor
"""
# class creating
class Test:
    def __init__(self):    # defining the non-parametrized constructor
        print("Non parameterized constructor")
    def display(self, test):
        print("INST0004" , test)
Test1 = Test() # creating objects
Test1.display("Programming 2") # calling or invoking the function
```

Non-parameterized Constructor: Solution

No need for value update

Output

```
Non parameterized constructor
INST0004 Programming 2
```

parameterized Constructors

There is need to update value

The parameterized constructor has several parameter including the self. Let's see an example to help us understand how to construct this.

```
#/usr/bin/env python
"""
A program to illustrate parameterized constructor
"""
# class creation
class Test:
    def __init__(self, arg):  # Defining paramterized constructor
        print("Parameterized constructor")
        self.arg = arg
    def display(self, test):
        print("INST0004", self.arg, test)
object1 = Test("Programming 2") # creating objects and passing a
    parameter
object1.display("Module") # calling function with one argiment
```

Parameterized Constructor: Solution

There is need to update value

Output

```
Parameterized constructor
INST0004 Programming 2 Module
```

Default Constructor

Python interpreter provides default constructor automatically

The default constructor is used when no constructor is defined or included in a class construct. When a constructor is not declared or specified in Python program, the Python interpreter automatically declares a default **(non-visible)** constructor in the class program.

Default Constructor

Python interpreter provides default constructor automatically

```
#/usr/bin/env python
"""
A program to demonstrate the use of a default constructor
"""
# Default constructor example
class Employee: # Class declaration
    EmployeeNo = 897645
    EmployeeName = "Danny"
    # function definition
    def status(self):
        print("Employee Number: ", self.EmployeeNo, "\n", "Employee Name
    : ", self.EmployeeName)
object1 = Employee() # object creation
object1.status() # invoking function
```

Default Constructor: Solution

Python interpreter provides default constructor automatically

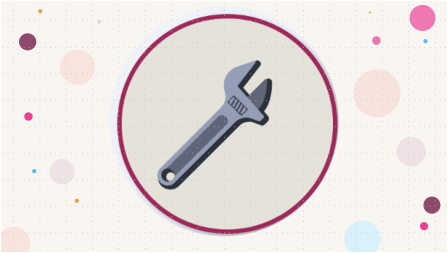
Output

```
Employee Number:  897645
Employee Name:  Danny
```

Multiple Constructors

Declaring multiple constructors in a class

In Python, we can also declare two or more constructor within a class. When we define two or more constructors with the same type in a single class, this is known as Constructor overloading. Here the objects and parameters of the two constructors will be different. For example, the **object1** might called the last constructor and does not have access to the first or second constructors. The key fact here is that if your class has many constructors and an object is created, the object can only access the last constructor. Let's see an example to help us understand how to define multiple constructors.



Multiple Constructor

Declaring multiple constructors in a class

```
#/usr/bin/env python
"""
A program to illistrate multiple constructors
"""
# Multiple constructor example
class DriverTest:
    # defining first constructor
    def __init__(self):
        print("First Constructor")
     # defining second constructor
    def __init__(self):
        print("Second Constructor")
     # defining third constructor
    def __init__(self):# output of the last constructor always display
        print("Third Constructor")
# creating an object
object1 = DriverTest()
```

Multiple Constructor: Solution

Declaring multiple constructors in a class

Output

```
Third Constructor
```

Note: Constructor overloading in NOT PERMITTED in Python


Summary

Summary

Let's revise the concepts of today's lecture

In this lecture we discuss the following:

- We discuss object-oriented programming
- We look at the difference between **object-oriented programming** and **procedural programming paradigm**
- We briefly discussed about **classes and objects**
- We looked at various concepts of object-oriented programming such as: inheritance, polymorphism
- We discussed about encapsulation, data abstraction and constructors.

Further Reading

chapters to find further reading on the concepts

You can read further this week's lecture from the following chapters:

- Python for Everyone (3/e) : **By Cay Horstmann & Rance Necaise** - *Chapter 9 Objects and Classes*
- Learning Python (5th Edition): **By Mark Lutz** - *Chapter 128 A More Realistic Example* - Focus on OOP
- Python Programming for absolute beginner (3rd Edition): **By Michael Dawson** - *Chapter 9 Object-Oriented Programming: The BlackJack Game*
- Python Crash Course - A hands-on, project-based introduction to programming (2nd Edition): **By Eric Matthes** - *Chapter 9 Classes*

UCL

Next Lecture 5

In week 5

Lecture 5: Building Class Structures, Algorithms and Pseudocode