UCL

INST0004 Programming 2

Lecture 07: Classes & Objects

Module Leader:
Dr Daniel Onah

2023-24


UCL

UCL

Copyright Note

Important information to adhere to

Summary of Previous Lecture

Recap of previous lecture

Re: Cap☺

In the last lecture we looked at ...

- basic concepts of recursion in Python
- understand how to create a recursive function to solve a factorial problem
- we discussed how to define recursive function for integer powers
- we discussed how to create a recursive function for a binary search problem
- we discussed how to define a recursive function to solve Tower of Hanoi task

Learning Outcomes

The learning outcomes for the lecture

The objective of this week's lecture is to introduce more of the concepts of classes and objects in Python programming. At the end of the lecture, you should be able to:

- understand **more of classes & objects**
- understand how to **implement a simple class**
- understand more on **constructors** structures
- understand how to **implement** methods
- understand how to **implement and test classes**

Object Decomposition

The idea behind object-oriented programming ...

Ideally, the easiest way to decompose your program tasks is to structure or break them into unit of modular (smaller) functions. Although, this is an excellent programming practice, however, experience shows that **this can be difficult to achieve from a large program perspective**. When you have a large program with so many functions, *this become challenging to understand and update especially when the functions have different complex operations*. In order to resolve this complexity, object-oriented paradigm was invented that uses collaborating objects. Each of the **object** *has its own unique set of data together with a set of methods that act upon the data.*

Object Decomposition

The idea behind object-oriented programming ...

You have already experienced this programming style when you used **strings**, **lists**, and **file objects**. Everyone of this objects has a set of **methods**. For example, we can use the **insert** or **remove** **methods** ***to operate on a list object***. When we develop an object-oriented program, we are *creating our own objects that describe important elements within the program application*. Lets look at an example of a **student class object**, within a student program, you might create ***Student*** and ***Objects***. Certainly, we should create methods for each of these objects.

Object Decomposition

The idea behind object-oriented programming ...

In Python programming, a class describes a set of objects with the same behaviour. By definition, a class is a blueprint for which objects are created. These objects though might have the same behaviour and structure, but they can still have their own **independent unique properties**.

Media Bakery.

Object Decomposition

The idea behind object-oriented programming ...

The ***str class*** in Python describes the behaviour of all strings. The class defines how a **string stores its characters**, *what methods can be used with strings*, and how the methods are implemented. In contrast to the **string class**, the ***list class*** describes the behaviour of objects that can be used to store a collection of elements' values. In list, each class defines a specific set of methods that you can use with its objects. For example, you can invoke the *upper()* method for a **string** class object. The upper() method is ideally a method for the string class.

```
"INST0004, Programming 1".upper()
```

Object Decomposition

The idea behind object-oriented programming ...

The list class has a different unique set of methods. For example if we try to invoke an **upper()** for a list of items, this would lead to **error** and it would be **illegal** *to say the least*, has the **list class** does not often have *upper()* method.

```
["INST0004, Programming 1"].upper()
```

However, the list class has a pop() method, and it would be perfectly legal if we call the **pop()** method on our list or collection of items. For example:

```
["INST0004, Programming 1"].pop()
```

Object State & Behaviour

What is the state & behaviour of an object ...

In Python programming, an object **encapsulates** state and behaviour, and has a unique identity.

An object's:

- behaviour determines which messages the object responds to, and how it responds to them
- state reflects cumulative effects on the object of its history
- identity distinguishes it from every other object

*An object's **state** can influence its **behaviour**, and its behaviour can change its **state**.*

Object State & Behaviour

What is the state & behaviour of an object ...

In Python programming, an object **encapsulates** state and behaviour, and has a unique identity.

Lets look at an objects from clear description:

- **state** is given by the value of its fields (sometimes referred to as attributes, members, instance variables)
- **behaviour** is defined by its methods and functions
- We can think of an object's identity as its **position in memory**

Object Oriented Paradigm

What is object oriented paradigm use for ...

In Python programming, object oriented Program does the following:

- Creates objects
- Requests that objects perform services by sending them messages, for example calling a function.

On receiving a message from a class object construct, an object takes control of execution. It might perform the following execution:

- Update its own state
- Send messages to other objects
- Once an object had dealt with a message, it returns control to the object that sent the message.

Classes

What is a class ...

A **class** in general term is a **blueprint** used for the Creation of objects. It describes aspects of related objects:

- Fields describe structural characteristics (**data**)
- Methods or functions describe behavioural characteristics (functionality)

Classes are used to model entities within a given problem domain. For example:

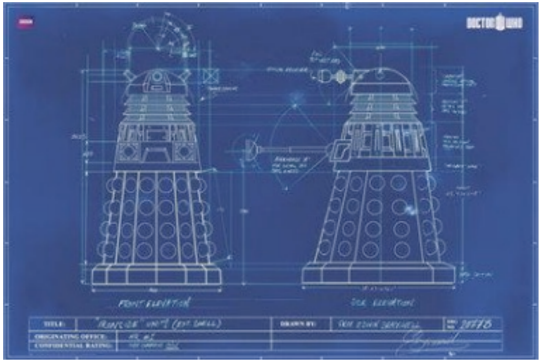
- A mechanic application system might have a **Mechanic** class
- A Hospital application system might have **Medical Doctor** and **Nurse** classes

Classes created in Python programming should be cohesive in nature. They should define **fields** and **methods or functions** with a logical connection.

Classes

What are classes ...

- In Python **objects are implemented through classes**.
- *A class is a specification of an object someone might build*.
  - Like a blueprint



Classes

What are classes ...

- In Python **objects are implemented through classes**.
- *A class is a specification of an object someone might build*.
  - Like a DNA

Classes

What are classes ...

- In Python **objects are implemented through classes**.
- *A class is a specification of an object someone might build*.
  - Like a Cookie Cutter

Classes

What are classes ...

- In Python **objects are implemented through classes**.
- *A class is a specification of an object someone might build*.
  - Like a Jelly Mould

Classes Example

What are classes ...

Let's look at a simple example of a circle class. This example would demonstrate the creation of circle class object given its colour and diameter.

```
1  #!/usr/bin/env python
2  """
3  A program to demonstrate creating a circle class
4  """
5  class circle:
6      # create a constructor
7      def __init__(self, colour, diameter):
8          self.colour = colour
9          self.diameter = diameter
10     # create an instance method
11     def display(self):
12         print("*** The class object features are ***")
13         print("Colour: " + str(self.colour))
14         print("Diameter: " + str(self.diameter))
```

Classes Example

What are classes ...

```
15  # creating object of the class
16 colour = " Blue"
17 diameter = 4.5
18
19 obj = circle(colour, diameter) # ensure you pass in the same number of
       arguments to the object (colour, diameter)
20
21 # calling the instance method using the object created i.e. 'obj'
22 obj.display()
```

Output

```
*** The class object features are ***
Colour:  Blue
Diameter: 4.5
```

Classes Example

Compute the area, circumference, diameter of a circle ...

Given a **radius** of a circle as **2.2**, calculate the diameter, the circumference and the area of the circle. Also, try requesting for user's input for the circumference to compute the radius of the circle.

```
1 #!/usr/bin/env python
2 """
3 A program to demonstrate calculating the diameter, circunference and
      area of a circle
4 """
5 import math
6 class circle:
7     # constructor
8     def __init__(self, radius):
9         self.radius = radius
```

Classes Example

Compute the area, circumference, diameter of a circle ...

```
10      # result display() function
11      def display(self):
12          diameter = circle.diameter(self)
13          circumference = circle.circumference(self)
14          area= circle.area(self)
15          print("The diameter of the circle is {} \nThe circumference of
        the circle is: {} \nThe area of the circe is: {}".format(round(
        diameter, 2), round(circumference, 2), round(area, 2)))
16          radius = circle.radius(self)
17          print("The radius of the circle used was: {}".format(radius))
18
19      # define diameter() function
20      def diameter(self):
21          return 2 * radius # formula = 2r
```

Classes Example

Compute the area, circumference, diameter of a circle ...

```
22      # define circumference() function
23      def circumference(self):
24          return 2 * math.pi * radius # 2PIr
25
26      # define area() function
27      def area(self):
28          return math.pi * pow(radius, 2) # PIr^2
29
30      # define radius() function
31      def radius(self):
32          circumference = circle.circumference(self)
33          return circumference/(2 * math.pi)  # r= C/2PI
34
35  # hardcode the radius value
36  radius = 2.2
```

Classes Example

Compute the area, circumference, diameter of a circle ...

```
37 # circumference = float(input("Enter circumference: ")) # circumference
       input
38 # create the object for the diameter of the circle
39 circle_obj_diam = circle.diameter(radius)
40 # create the object circumference of the circle
41 circle_obj_circ = circle.circumference(radius)
42
43 # create the object area of the circle
44 circle_obj_ar = circle.area(radius)
45
46 # create an object to display the result
47 circle_obj_disp= circle.display(radius)
48 # create the object radius of the circle
49 #circle_obj_rad = circle.radius(circumference) # this works with the
       circumference input
50 circle_obj_rad = circle.radius(circle_obj_circ)
```

Class Example: Output

Compute the diameter, circumference, area of a circle ...

Output

```
The diameter of the circle is 4.4
The circumference of the circle is: 13.82
The area of the circe is: 15.21
The radius of the circle is: 2.2
```

Public Interface & Encapsulation

What is a public interface and encapsulation ...

When designing a class, you start by specifying its public interface. The public interface of a class consists of all methods that a user of the class may want to apply to its objects. Let's consider a simple example. We want to use objects that simulate cash registers. A cashier who rings up a sale presses a key to start the sale, then rings up each item. A display shows the amount owed as well as the total number of items purchased. In our simulation, we want to call the following methods on a cash register object:

- Add the price of an item.
- Get the total amount owed, and the count of items purchased.
- Clear the cash register to start a new sale.

Public Interface & Encapsulation

What is a public interface and encapsulation ...

The method definitions and comments constitute the public interface of a class. The **data and the method bodies make up the private implementation of the class**. In order for us to understand and observe the behaviours of our methods or to see a method in action, we must first need to **construct an object**:

```
register = CashRegister()
    # Constructs a CashRegister object.
```

This statement defines the register variable and initializes it with a reference to a new **CashRegister** object. Once the object has been constructed, we are ready to **invoke a method**:

```
# The register object is use to invoke addItem method.
register.addItem(29.30)
```

Public Interface & Encapsulation

What is a public interface and encapsulation ...

Public interface of a class is the visibility of all set of functions or methods provided by a class, together with a **description of their behavior**. When you work with an object of a class, you do not know how the object stores its **data field**, or how the **methods** are implemented. You need not know how a **str object** organises a character sequence, or how a **list** stores its elements. *All you need to know is the public interface––which methods you can apply, and what these methods do.* **Encapsulation** is the process of providing a public interface, while hiding the implementation details such as the private constructs accessible only through special methods and class constructs.

Encapsulation

More on encapsulation ...

When you create your own classes using the concept of object oriented programming, you should use ***encapsulation*** to enforce information hiding. What this simply means for example is that, you would specify a set of **public methods** and hide the implementation details. The implementation details might be the various operations supporting the functionality of your program such as **attributes**, **methods**, **functions**, and so on. With the concept of encapsulation, other programmers or developers could use your program code successfully without knowing anything about the main implementation details, because this is already define as **invisible**, just as we are able to use the ***str*** and ***list*** classes without knowing their implementation details.

Encapsulation

More on encapsulation ...

When we develop our program for a long period of time, *it is a common place that the implementation details might change over-time*. This is to make our program objects to be more efficient and more capable to tolerate or withstand any fault or errors. This is when **encapsulation** becomes crucial to enabling these changes to our programs. Note that when the **implementation is hidden**, any future improvements of the program constructs does not affect the programmers or developers who use the class objects. *For example you can drive a car without knowing how the engine works*.

Important Points on Class and Objects

Recap on a class ...

A **class**:

- is an entity created at program development and execution time
- is a blueprint from which objects are instantiated
- defines a collection of fields or data points and functions or methods that its instantiation would have

Important Points on Class and Objects

Recap on an object ...

An **object**:

- is a run-time execution entity: it must be created from a class during program execution
- is an instance of a class
- stores a value for each field
- functions or methods can be invoked on an object

Instance Variable

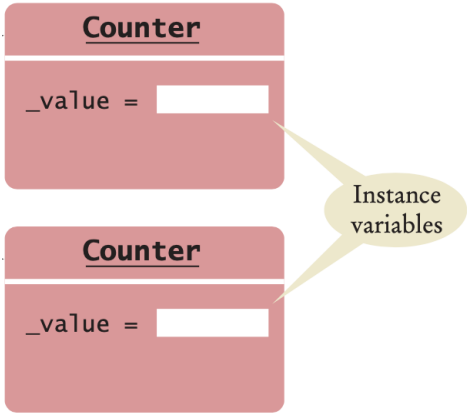
Class variables ...

An object stores its data in **instance variables**. **Instance variables** are variables that are declared **inside a class**. When creating a class, we have to decide which data we want each object to store. The object would need to have all the information required to call any method. For example, a **CashRegister** class with an object should be able to return the correct value using the **getTotal()** accessor method. The accessor or get method is a method that returns the value of any given data type. It does not modify the value but rather returns exactly whichever value that has been defined within the class program. While the mutator or set method is a method that modifies the value of an attribute. It is used to control the changes made to a variable (e.g. **setTotal()**). The **getTotal()** method must either store all entered prices or values and compute the total in the method call.

Instance Variable

Class instance variables ... declared inside a class

Classes can have **common behavior**, but each of them can also have **different state**. Similarly, **objects of a class** can have their **instance variables** set to different values.



Class Method

Class methods ...

A **method** can be used to access the **instance variables** of the object on which it acts. Lets look at an example to help us understand this better.

```
def click(self) :
    self.value = self.value + 1
```

The **click method** advances the **value** instance variable by 1. A method definition is very similar to a function with these exceptions:

- A method is defined as part of a class definition. That is, ***methods are declared within the block of a class***.
- You can invoke a method by using the class name with a **.** keyword.
- The **first parameter variable** of a **method** is called **self**.

Class Method

Class methods ...

It is important to note that **instance variables** must be referenced within a method using the **self parameter** e.g. ***self.value***. Note how the **add** method increments the instance variable *value*. Which instance variable? The one belonging to the **object on which the method is invoked**. For example, consider the call:

```
concertCounter.click()
```

This call advances the **value** variable of the ***concertCounter*** object. No argument was provided to the click method even though the definition includes the *self* **parameter variable**. The *self* parameter variable refers to the object on which the **method** was invoked—**concertCounter** in this example. Note that when you create an instance method of a class you don't need to pass-in a parameter list. You just need to create this as: def **getValue(self)** using a reference to the instance method (self).

Class Method

Class methods ...

Let us look at the other methods of the **Counter class**. The **getValue** method returns the current value:

```
def getValue(self):
    return self.value
```

This method is provided so that users of the **Counter class** can find out how many times a particular *counter* has been **clicked**. A class user **should not directly access any instance variables**. Restricting access to instance variables is an essential part of **encapsulation**. This ***allows a programmer to hide the implementation of a class from a class user***. Note that using the **@staticmethod** decorator on a method would ensure that **self** *will never be implicitly passed-in*, ***like it normally would for class methods***. You could also use **@classmethod** decorator if you wish for other **subclasses** within your program *to override the class method*.

Class Method

Class methods ...

The **reset method** resets the counter class value:

```
def reset(self) :
    self.value = 0
```

In Python, you don't explicitly declare instance variables. Instead, when one first assigns a value to an instance variable, the **instance variable is created**. In our sample program, we call the **reset method** before calling any other methods, so that the **value** instance variable is created and initialized. Lets look at the complete **Counter class** and a **driver module** as provided to help us understand more about the counter class program.

Mutator Method

Class methods ...

If you observe the public interface of a class, it is useful to note that some of the methods are classified as ***mutators*** and ***accessors*** methods. A mutator method modifies the object on which it operates or acted upon. The mutator method is referred to as the **setter or set method**. Let's look at an example of **mutator** or **setter** method:

```
def setTotal(self, price):
    self._price = price
def setCount(self, count):
    self._count = count
```

Notice how we provided the attributes for **getter** and **setter** methods with an underscore (_). The attributes holding the **price** and **count** are **private** (**non-public**) because it has a **leading underscore** on its name, e.g. [*._price* and *._count*]. This allows us to ***enforce encapsulation*** and information hiding, so as to restrict users of our class modifying the values assigned to the **attributes** or **instance variables**.

Mutator Method

Class methods ...

For example, the **CashRegister class** program has two mutators: **addItem** and **clear**. As soon as you call or invoke any of these methods, the *object or its value would changed*. You can observe that change by calling the **getTotal** or **getCount** methods. These methods are the **getter or get method** that only return the modified object or value.

```
def addItem(self, price):
    self._totalPrice = self._totalPrice + price
def clear(self):
    self._totalPrice = 0.0
```

The setter method does the input transformation, converting the value assigned to the class attributes into its new or updated value.

Accessor Method

Class methods ...

An **accessor** method **queries the object for some information without changing it or modifying its attribute value**. the **accessor** method is referred to as the **getter or get method**. The **CashRegister** class has two accessors: **getTotal** and **getCount**. Applying either of these methods to a *CashRegister* object, simply returns a value and does not modify the object. For example, the following statement prints the current **total** and **count**:

```
def getTotal(self):
    return self._total
def getCount(self):
    return self._count
```

```
print(register.getTotal(), register.getCount())
```

Cash Register Class & Method

Example: Cash Register class and methods ...

```
#!/usr/bin/env python
"""
A program module define and demonstrate a CashRegister class.
"""
# A simulated cash register that tracks the item count and total amount.
class CashRegister:
    # Constructs a cash register with cleared item count and total.
    def __init__(self):
    # underscore is use to make the attribute private and enforces
    encapsulation
        self._itemCount = 0
        self._totalPrice = 0.0
    # Adds an item to this cash register.  # @param price of this item
    def addItem(self, price):
        self._itemCount = self._itemCount + 1
        self._totalPrice = self._totalPrice + price
```

Cash Register Class & Method

Example: Cash Register class and methods ...

```
16      # Gets the price of all items in the current sale.
17      def getTotal(self):
18          return self._totalPrice # @return the total price
19
20      # Gets the number of items in the current sale.  @return the item
        count
21      def getCount(self):
22          return self._itemCount
23
24      # Clears the item count and the total.
25      def clear(self):
26          self._itemCount = 0
27          self._totalPrice = 0.0
```

Cash Register Driver Class & Object

Example: Cash Register driver class ...

```
#!/usr/bin/env python
"""
This program is used to test the CashRegister class
"""
# import the cash register module
from CashRegister import CashRegister
class RegisterDriver:
    register = CashRegister()
    register.addItem(20.35)
    register.addItem(33.95)
    register.addItem(45.51)
    print("The count value is: %s "%register.getCount())
    print("The total sale is:   %.2f" % register.getTotal())
```

Cash Register: Solution

Cash register output

Output

```
The count value is: 3
The total sale is: $99.81
```

© James Richey/iStockphoto.

Counter Class & Method

Example: Class counter and methods program ...

```
#!/usr/bin/env python
"""
 A program to demonstrate a module that defines a Counter class.
"""
# Models a tally counter whose value can be incremented, viewed, or
    reset.
class Counter :
    # Gets the current value of this counter.
    def getValue(self):
        return self.value
    # Advances or increment the value of this counter by 1.
    def click(self):
        self.value = self.value + 1
    # Resets the value of this counter to 0.
    def reset(self):
        self.value = 0
```

Counter Class & Method

Example: Class counter and methods driver program ...

```
#!/usr/bin/env python
"""
This program is uased to test and demonstrate the Counter class
"""
# Import the Counter class from the counter module.
from counter import Counter
class CounterTest:
    tally = Counter() # create counter object
    tally.reset() # reset the value of the tally
    tally.click() # use the object to invoke the click method twice
    tally.click()
    result = tally.getValue() # tally object invoking the getValue()
    print("Value:", result)
    tally.click()
    result = tally.getValue()
    print("Value:", result)
```

Constructors

Class constructors ...

A **constructor** *defines and initialises the instance variables of a an object*. A **constructor** is automatically called whenever we create an **object**. To create an **instance of a class**, we use the name of the class as if it were a function together with any arguments required by the constructor. For example, to create an instance of ***Employee*** class, we use the command:

```
employee = Employee()
```

In the above construct, an object is created and the constructor of the **Employee** class is automatically called. The constructor has no parameter list when created in the *Employee* class, therefore this particular **constructor requires no arguments**.

Constructors

Class constructors ...

Constructors determine how instances are built. They are use to construct the attributes if they are complex types. It initialise the attributes or instance variables and check the constraints on parameter e.g. **range of values**.

Bob the builder, keith Chapman

Constructors

Class constructors ...

The following are common characteristics of a constructor:

- They are defined using the special method **name __init__**.
- They are used to initialise instance variables of an **object**.
- Every class has a constructor
- A constructor with no parameter is called a **default constructor**
- A class may have **multiple constructors** (this is called overloading)
- They are located at the beginning part of a class
- They are the **life-cycle** of a class
- They are **automatically called when an object is created or instantiated**.

Car Class Example using standard variables

Car class program with variables declared outside the class block ...

```
#!/usr/bin/env python
"""
A program to demontrate a car class
"""
class Car:
    def __init__(self, colour, model):
        self.colour = colour # self.colour is refering to the class
    attribute or instance variable colour
        self.model = model
    def display(self):
        print("Car model: ‘%s’, colour: ‘%s’ "%(model, colour))
colour = "Black" # declaring the variable within the program
model = "Ford"
carObject = Car(colour, model) # passing the variable as arguments
carObject.display() # display the object created, by using the object
    name to invoke the display() function or method
```

Car Class Example using instance variables

Car class program instance variables declared inside the class ...

```
#!/usr/bin/env python
"""
A program to demontrate a car class
"""
class Car:
    colour = "Black" # class instance variable
    model = "Ford" # class instance variable
    def __init__(self, colour, model):
        self.colour = colour # self.colour is refering to the class attribute or instance variable colour
        self.model = model
    def display(self):
        print("Car model: '%s', colour: '%s' "%(Car.model, Car.colour))
carObject = Car(Car.colour, Car.model)
carObject.display()
```

Car Class Example

Car class program...

Output

```
Car model: ‘Ford’ , colour: ‘Black’
```

A Pen Class Example using instance variables

Pen class program instance variables declared inside the class methods ...

```
#!/usr/bin/env python
"""
A program to create a pen class
"""
class MyPen:
    # Constructor method (this can also called an initializer)
    def __init__(self, colour):
        self.colour = colour

    # Instance method
    def display(self):
        print("Greetings from MyPen class!")

    # Instance method with parameter list from arguments
    def add_ink(self, ink):
        self.colour = ink
```

A Pen Class Example using instance variables

Pen class program instance variables declared inside the class methods ...

```
17      def getColour(self):
18          return self.colour
19
20  # Creating an instance of MyPen class
21  my_pen = MyPen("Blue")
22
23  # Calling the display method
24  my_pen.display()  # Output: Greetings from my pen class!
25
26  # Accessing, updating and modifying instance attributes
27  print("Initial colour:", my_pen.colour)  # Output: Initial colour: Blue
28  my_pen.add_ink("Black")
29  print("Updated colour:", my_pen.colour)  # Output: Updated colour: Black
30  # you can use a getter method to return the modified instance attribute
31  print("Updated colour:", my_pen.getColour())  # Output: Updated colour:
      Black
```

A Pen Class Example using instance variables

Pen class program instance variables declared inside the class methods ...

In the example program above.

1. We define a class called **MyPen**.
2. Within the Pen class, we define four methods:
   - The **__init__** method is the constructor and is automatically called when an instance of the class is created. It initializes the instance with a **colour** attribute.
   - The **display** method is a simple instance method that prints a message.
   - The **add_ink** method is an instance method that takes an argument **ink** and passed-in as parameter to the method and updates the **colour** attribute of the instance.
3. We create an instance of MyPen class called **my_pen** and pass the initial colour of "Blue" to the constructor.
4. We call the **display** method on the my_pen instance.
5. We access and modify the **colour** attribute using the **add_ink** method.
6. we can also retrieve the modified colour using the getter method **getColour()**.

Static method *Example*

Significance of static method in class ...

In Python programming, static method specifically belongs to the class and **not to any instance of that class**. That is, the functionality of static method ideally *do not depend on any class instance specific data or attributes*. ***Static methods are mainly use for a self contain piece of code or use to perform utility function operations without any need for class instance attributes***. Static methods are defined in Python using the decorator **@staticmethod**.

Static method *Example*

Significance of static method in class ...

```
#!usr/bin/env python
"""
A program to demonstrate static method
"""
class Person:
    # creating a static method
    @staticmethod
    def institution():
        print("University College London")
# call static method on the class, without creating object instance
Person.institution()# without need for class instance
```

Static method *Example*

Significance of static method in class ...

In the previous example, you noticed that we created a static method called **institution** using the decorator (*@staticmethod*). The static method belongs to **'Person** class. We can invoke or call this method using the class name i.e., **Person.institution** without the need to create an instance of the class object or utilising any attribute. Lets look at some characteristics of static methods:

- Static methods do not utilise or gain access to instance specific attributes or variables. For example using the **self** keyword to access the instance attributes i.e., **self.name**.
- static methods are able to access class-level attributes declared as class variables within the class.
- Ideally, they are mostly useful for helper functions that are related to the class, but not to any specific instance attributes or data field.
- Static method can make program code more readable by placing similar functions within the class's name-space in a logical order.

Static method *Example*

Significance of static method in class ...

```
#!usr/bin/env python
"""
A program to demonstrate static method
"""
class Person:
    name = "Danny Onah" # class variable
    # creating a static method
    @staticmethod
    def institution():
        print("University College London")
        # assigning and accessing class-level attribute or variable
        employeeName = Person.name
        print(employeeName)
# call static method on the class, without creating object instance
Person.institution()# without need for class instance
```

Static method *Example*

Significance of static method in class ...

Let's look at an example of a full program.

```
"""
A program to demonstrate static method
"""
class Person:
    name = "Danny Onah" # class variable
    # constructor
    def __init__(self, name):
        self.name = name
    def show(self):
         print(self.name)
```

Static method *Example*

Significance of static method in class ...

```
11      # creating a static method
12      @staticmethod
13      def institution():
14          print("University College London")
15          # accessing class-level attributes or variables
16          employeeName = Person.name
17          print(employeeName)
18  # call static method on the class, without creating object instance
19  Person.institution()# without need for class instance
20  PersonObj= Person("Daniel Onah") # creating instance of the class
21  PersonObj.show()
```

Class method *Example*

Significance of class method in class ...

Let's look at the class method example above, always remember that the class method is basically the method of the class **Person**. We can call this method by using the class name and the name of the method i.e., **Person.institution()** in the same manner as static method without creating an instance of the class. The difference between the static method and the class method is that:

- *you can easily call the class method on an instance of the class and this will still have access to the class-level attributes.*
- class methods are often use when you want to perform operations that are related to the class itself. such as for example:
  - creating instances of the class with specific requirements or functionalities
  - getting access to class-level attributes or variables
  - performing any class wide related computation.

Class method *Example*

Significance of class method in class ...

Let's look at an example of a class method program.

```
#!usr/bin/env python
"""
A program to demonstrate class method
"""
class Person:
    staffName = "Danny Onah" # class variable
    #staffDept = "Information Studies"
    # constructor
    def __init__(self, name):
        self.name = name
```

Class method *Example*

Significance of class method in class ...

```
11      # creating a class method
12      @classmethod
13      def institution(name):
14          print("University College London")
15          # accessing class-level attributes or variables
16          print(f"Name: {name.staffName}")
17          #print(f"Department: {dept.staffDept}")
18
19  # call class method on the class, without creating object instance
20  Person.institution()# without need for class instance or attribute
```

Class method *Example*

Significance of class method in class ...

```
21 # we can also call our class method on an instance,
22 # this will still access the instance attributes
23 print("*** Instance attributes using object ***")
24 PersonObj = Person("Danny Onah") # with constructor
25 #PersonObj = Person()# without constructor
26 PersonObj.institution()
```

Coin Class

Creating Coin class objects ...

The **Coin** class defines the **Coin** problem set:

- This class tells us how to create one or more coin objects
- Each coin object stores the most recent flip outcome
- Describes how a coin flip works ...

The **RunCoin** class

- Create one or more Coin objects
- Flips the coin
- Test and displays flip outcomes

Question: How do we add another coin and flip that?

Coin Class Example

Coin class program ...

```
#!/usr/bin/env python
"""
A program to demonstrate a coin class
"""
import random
class Coin:

    global HEADS, TAILs, face
    HEADS = 0
    TAILS = 1

    def __init__(self):
        Coin.flip(self) # invoke the flip() method using the name of the
     class
```

Coin Class Example

Coin class program ...

```
14      def flip(self):
15          face = (int)(random.randint(0, 1))# this output any number
        between 0 and 1
16          #print(face) # uncomment to see the random number generated
17          if (face == HEADS):
18              return "Heads"
19          else:
20              return "Tails"
```

Coin Class Example

Coin class program ...

```
#!/usr/bin/env python
"""
A program to demonstrate using the coin class as a helper class to
    create the objects
"""
from Coin import Coin
class RunCoin:
    def display(self):
        print("Initially: ", Coin.flip(self))
        #Coin.flip(self) # this does not add much. you can try
    uncommenting this and see what happens
        print("Now: ", Coin.flip(self))

coinObject = Coin()
coinObject.flip()
coinObject = RunCoin.display(coinObject)
```

Employee Class Example

Employee class program ...

Lets look at an example of an employee record system. We first describe the functionalities. In the program we want to define constructor method **__init__** in the employee class. The **job** attribute is set to *None*. Note that within the **Person** class we provided as an employee job to be **'mgr'** string. But, this does not really matter as our program would only use this string when we set the employee as a manager. In the **increment** customization, we run the original **__init__** in Employee by calling through the class name. This allows us to initialize the object's state using the attributes or data field information. You would notice that the call that created the Person **Joe**, did not passed in the job title, because we have modified the constructor without the job name ***def __init__(self, name, pay)***.

Employee Class Example

Employee class program ...

```
#/usr/bin/env python
"""
A program to demonstrate an exmployee record
"""
class Employee:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def increment(self, percent): # increment
        self.pay = int(self.pay * ( 1 + percent))
    def __repr__(self):
        return `[Employee: %s, %s]'%(self.name, self.pay)
```

Employee Class Example

Employee & Person class program ...

```
#/usr/bin/env python
"""
A program to demonstrate an exmployee record - using Employee program
    file
"""
# import Employee class to inherit the features of the class
from Employee import Employee
class Person(Employee):
    def __init__(self, name , pay):  # Redefine constructor
        Employee.__init__(self, name, 'mgr', pay) # Run original with '
    mgr'
    def increment(self, percent , bonus=.10):
        Employee.increment(self, percent + bonus)
```

Employee Class Example

Employee & Person class program ...

```
11 if __name__ == '__main__':
12     danny = Person('Danny Onah', pay=500000)
13     elaine = Person('Elaine Pang', pay= 200000)
14     print(danny)
15     print(elaine)
16     print(danny.lastName(), elaine.lastName())
17     elaine.increment(.10)
18     print(elaine)
19     joe = Person('Joe Ken', 100000) # job detail not included (name)
20     joe.increment(.10)  # implied/set by class
21     print(joe.lastName())
22     print(joe)
```

Employee Class Example

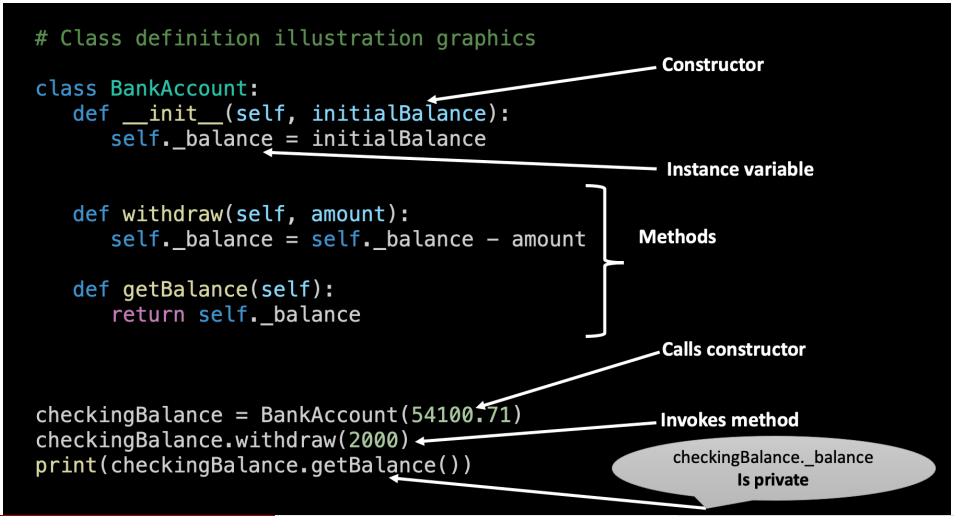
Employee & Person class program ...

Output

```
[Employee: Danny Onah, 500000]
[Employee: Elaine Pang, 200000]
Onah Pang
[Employee: Elaine Pang, 240000]
Ken
[Employee: Joe Ken, 120000]
```

Key Points

Key points to remember about classes...



Good Programming Tips

Good practice ...

All instance variables should be private and most methods should be public. Although most object-oriented languages provide a mechanism to explicitly hide or protect private members from outside access, **Python does not**. Instead, the designer of a class has to indicate which **instance variables and methods** are supposed to be private. It's then the ***responsibility of the class user not to violate the privacy***. *It is common practice among Python programmers to use names that begin with a* **single underscore for private instance variables and methods**.

Good Programming Tips

Good practice ...

The single underscore serves as a flag to the class user that those members are private. *You then must trust that the class user will not attempt to access these items directly.* This technique is recognized by documentation generator tools that **flag private instance variables and methods** in the documentation. You should always use **encapsulation**, in which all instance variables are private and are *only manipulated with methods*. *Typically, methods are public*. However, sometimes you have a method that is used only as a helper method by **other methods**. In that case, you should make the helper method private by using a name that begins with a **single underscore**.


Summary

Summary

Let's revise the concepts of today's lecture

In this lecture we discuss the following: Object Oriented programming is all about modularity ...

- Data + Methods = Object
- Classes provide a specification of objects
- Instances are real concrete realization of classes
- Use the keyword **class** to define classes
- Use **.** with **class name to access class functions and instance variables** in an object instance
- Python uses the special name **__init__** for the constructor because its purpose is to **initialize an instance of the class**

Summary

Let's revise the concepts of today's lecture

- Now your setter method has a true goal instead of just assigning a new value to the target attributes. It has the goal of adding **extra behavior** to the attributes by enforcing encapsulation using the *underscore symbol* before the class instance variable or attributes.
- The addition of this extra behaviour **makes the attribute private so they could not be modified from outside the class** and these can only be access through the **getter** and **setter** methods.
- We want our programs to observe object-oriented programming paradigm by enforcing encapsulation and using the **setter** and **getter** methods as the only means of ***modifying and retrieving information from the attributes***.

Further Reading

chapters to find further reading on the concepts

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

- Python for Everyone (3/e) : **By Cay Horstmann & Rance Necaise** - *Chapter 9 Objects and Classes*
- Learning Python (5^{th} Edition): **By Mark Lutz** - *Chapter 28 A More Realistic Example*
- 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*

Next Lecture 8

In week 8

Lecture 8: Inheritance and Polymorphism