UCL

INST0004 Programming 2

Lecture 05: Building Class Structures, Algorithms and Pseudocode

Module Leader:
Dr Daniel Onah

2023-24


UCL

UCL

Copyright Note

Important information to adhere to

Recap of Previous Lecture 4

Recap of previous lecture

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

- We looked at basic concept of object-oriented programming
- We discussed - classes and objects creation
- we discussed the concept **data abstraction**
- We discussed the concept of constructor in programming
- We looked at **inheritance** and **polymorphism**
- We discussed various Python methods to achieve object-oriented programming paradigm

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, building classes, pseudocode and algorithms. At the end of the lecture, you should be able to:

- learn more on the basic concepts of building classes
- understand more about static variables, static and class methods
- understand how to test class in isolation
- understand class decorator
- understand the concepts of pseudocode in programming
- understand the concepts of program algorithms
- understand the implementation of an algorithm using pseudocode
- understand various Python methods to achieve object-oriented programming

Creating class and static variable

How are static variables used in a class

In Python programming, all objects created share either a class or a static variable. For different objects, non-static variables or instance variables vary in their functionalities. Class variables have values assigned to them in the class declaration, while instance variables or non-static variables have values assigned to them within the methods.

Characteristics of an object:

- State - The attributes of an object represents its state. It also reflects the properties of an object.
- Behavior - The method of an object represents its behavior.
- Identity - Each object must be uniquely identified and allow interacting with the other objects.

Class variable

How do we identify class variable

Sometimes, the value of a variable might belong solely to the class and not any other object of that class. **Such variables are known as a *class variable***. Class variable are sometimes referred to as **static variable**. Class variables are declared in a program when they are needed for a general purpose functions. For example, let's say we want to assign bank account numbers sequentially and we want the constructor to construct the first account with **101**, the second as **102**, and so on. *How do we write our program to do this?*. In order to solve this problem, we will need to create a single value variable, for example called ***assignedNumber*** that would be solely declared as a property of the class and not of any object of the class. It is significant to note that class variables are declared at the same level as methods. But in contrast, ***instance variables are created in the constructor of a class***.

Class variable

How do we identify class variable

Let's look at an example to help us understand this more.

```
#!/usr/bin/env python
"""
A program to demonstrate class variable
"""
# defind the class
class BankAccount:
    # a class variable is declared at the same level as a method or
    constructor
    _assignedNumber = 100 # class variable

    # define a constructor
    def __init__(self):
        self._accountBalance = 0 # instance variable
        BankAccount._assignedNumber = BankAccount._assignedNumber + 1
        self._accountNumber = BankAccount._assignedNumber
```

Class variable

How do we identify class a variable

In the previous program, every BankAccount has its own ***_accountBalance*** and also its own ***_accountNumber*** instance variables or attributes, but there is only a single copy of the *_assignedNumber* variable declared at the class level as class variable. The variable is stored in a separate location and outside any **BankAccount** objects declared in the program.

A final note: A class variable belongs to the class for which it is declared and **not to any instance of the class**

Class variable

Important information

Note that we reference the **class variable** by using the class name to call the variable name such as **BankAccount._assignedNumber**. Similar to instance variables or attributes, the class variables should always be declared with the concept of **encapsulation** that is, it should be declared **private** *to ensure that the methods of other classes do not change their values*. However, we can declare class constants (constant variable) as public within our program. For example, our **BankAccount** class can have or define a public constant value such as for example:

```
class BankAccount:
    OVERDRAFT_PAYMENT = 200.10
```

In this case, methods from any class can update and refer to such a constant by calling **BankAccount.OVERDRAFT_PAYMENT**.

Building a class - the *"self"* keyword

The significance of the *self* in a class

The ***self*** keyword as used in a class represents the instance of the class. By using the **self** we can *access the attributes and methods of the class in Python*. This *binds the attributes or instance variables with the given arguments*. The reason you need to use ***self***, is because Python **does not use** the **@syntax** or the ***this*** keywords as used in ***Java Programming*** and other programming languages to refer to **instance attributes**. The ***self*** keyword allow us to **refer to the** instance variables or attributes within our Python program construct.

Creating class and static variable: Example

How are static variables used in a class

Let's look at an example to help us understand this better.

```
#!/usr/bin/env python
"""
A program to demonstrate the creation of class or static variables
"""
# creating class declaration
class Student:
    _tuition = 32000 # class or static variable
    def __init__(self, ID, name):
        self.ID = ID # instance or non-static variables
        self.name = name
# objects creation
student1 = Student(987656, "Danny")
student2 = Student(897645, "Elaine")
```

Creating class and static variable: Example

How are static variables used in a class

```
# accessing class variables using object name to display the object information
print("Student1 Tuition: ", student1._tuition) # displays 32000
print("Student2 Tuition: ", student2._tuition) # displays 32000
print("Student1 ID: ", student1.ID) # displays 987656
print("Student2 ID: ", student2.ID) # displays 897645
print("Student1 Name: ", student1.name) # displays Danny
print("Student2 Name: ", student2.name) # displays Elaine

# class variables can be accessed using class name also
print("Standard Students Tuition: ", Student._tuition) # displays 32000

# we can also change the tuition for student1 without affecting student2
student1._tuition = 36000
print("Student1 Tuition Changed: ", student1._tuition) # displays 36000
print("Student2 Tuition Unchanged: ", student2._tuition) # displays 32000
```

Creating class and static variable: Example

How are static variables used in a class

```
29 # now to change the tuition from all students, we can do this directly
       from the class
30 Student._tuition = 45000
31 print("Student2 Tuition Changed: ", student2._tuition) # displays 45000
```

Output

```
Student1 Tuition:  32000
Student2 Tuition:  32000
Student1 ID:  987656
Student2 ID:  897645
Student1 Name:  Danny
Student2 Name:  Elaine
Standard Students Tuition:  32000
Student1 Tuition Changed:  36000
Student2 Tuition Unchanged:  32000
Student2 Tuition Changed:  45000
```

Class method vs. Static Method

Looking at class method vs. static method

The **@classmethod** decorator is a built-in function decorator that evaluates after your function is specified. The outcome of that analysis casts a shadow over your function definition. The class is implicitly passed as the first argument to a class method, just as the instance is implicitly passed to an instance method.

Syntax

```
#!/usr/bin/env python
"""
A Syntax for defining and creating a class method
"""
class <Class_name>:
    @classmethod
    def <method_name>(cls, arg1, afr2, ...):
        # method body here
        return values
```

Class method vs. Static Method

Looking at class method vs. static method

Let's look at some ways we could described a class method properties:

- A class method is bound to the class rather than the class's object.
- The class parameter considers the class rather than the object instance. Therefore, the class method have access to the class state.
- A class method can change the state of a class that affects all of its instances. That is, it can change a class variable that affects all of its instances.

Instance variable in Constructor

Define instance variables or attributes inside a constructor

It is important to note that Python is a dynamic language in which all variables including instance variables or attributes are created at run time. You can create *instance variables inside a class method*. Sometimes, this is done if the constructor has not yet been created and we are using a Python default constructor. *Since constructor is invoked before any method can be called, any instance variable or attributes created within the constructor are also available and access by all class or static methods*. However, by convention, **creating instance variables in methods directly is not ideal in most cases**. This is because if you call a method on a newly created object without having to call the other helper method or methods this could lead to a **run-time error** if you attempt to update the value of the variable.

By convention, all instance variables should be created inside the constructor and we can call them within a class method or static method.

Class method vs. Static Method

Looking at class method vs. static method

An implicit first argument is not passed to a static method. the **@staticmethod** decorator is used for static methods.

Syntax

```
#!/usr/bin/env python
"""
A Syntax for defining and creating a class method
"""
class <Class_name>:
    @staticmethod
    def <method_name>(cls, arg1, afr2, ...):
        # Method body here
        return values
```

Class method vs. Static Method

Looking at class method vs. static method

The following are static method properties:

- A static method is one that is bound to the class rather than the class's object..
- A static method can't access or change the state of the class.
- It is present in a class. Because it is a class method, it makes sense for it to be present in a class.

Class method vs. Static Method: Example

Looking at class method vs. static method

```
#!/usr/bin/env python
"""
A program to illustrate class and static methods
"""
# example of a class method and static method
from datetime import date
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # a class method to create a Studenr object by birth year.
    @classmethod
    def birthYear(cls, name, year):
        return cls(name, date.today().year - year)
```

Creating class and static variable: Example

How are static variables used in a class

```
16 # a static method to check if a Student is adult or not
17     @staticmethod
18     def isAdult(age):
19         return age > 18
20
21 # creating the objects
22 student1 = Student("Danny", 20)
23 student2 = Student.birthYear("Danny", 2003)
24
25 # display the age
26 print("The age of student1: ",student1.age)
27 print("The age of student2: ",student2.age)
28
29 # display the outcome of the check if the student is adult
30 print("Is student2 an adult: ", student2.isAdult(20))
```

Creating class and static variable: Solution

How are static variables used in a class

Output

```
The age of student1:  20
The age of student2:  20
Is student2 an adult:  True
```

Testing a Class

How do we test our class

It is very important to know the various ways you could test your class functionalities. We can test our class program as often as possible. Let's say we have two classes created, one of which is the class with the main object creation and the second is the test or driver class. *How do we implement the unit test of the modular class parts knowing fully well that this would become part of a larger program that users would interact with and store or retrieve data from*, and so on.

'If you win, say nothing. if you lose, say less' - **Paul Brown**

Testing a Class

How do we test our class

It is always a good practice to test our class in **isolation** before integrating this into our program. This testing in isolation outside a complete program is called **unit testing**. We can perform this test two ways:

- *using the Python interactive environment which provide you with access to* ***Python shell****.*
- *you can also test your class program by constructing an object, calling methods, and verifying that your get the expected return values from the return or print statements.*

Unit Testing a Class

How do we test our class

A unit test allow us to verifies whether a class works correctly in isolation, outside a complete program.

© Chris Fertnig/Stockphoto.

An engineer tests a part in isolation. This is an example of unit testing.

Unit Testing a Class

How do we test our class

Using the interactive testing approach is quick and convenient **but it has a drawback**. *When you find and correct an error, you would need to re-type in the tests again from the beginning in most cases*. Conventionally, as your *classes develop and become more complex*, you should write tester programs. **A tester program is a driver module that imports the class and contains statements to run methods of your class**. A tester program typically carries out the following steps:

- *Construct one or more objects of the class that is being tested*.
- *Invoke one or more methods*.
- *Print out one or more results*.
- *Print the expected results*.

Testing a Class

How do we test our class

Let's look at a simple interactive session that could test a class called **CashRegister**.

```
>>>from cashRegister import cashRegister
>>>reg = CashRegister()
>>>reg.addItem(1.95)
>>>reg.addItme(0.95)
>>>reg.addItme(2.50)
>>>print(reg.getCount())
>>>print(reg.getTotal())
```

Class decorator

What is a class decorator

Class decorators are potent and valuable tool in Python, which are used to change a function or class's behaviour. *Decorators allow us to to wrap another feature to expand its behaviour without changing it permanently*. Decorator can be defined as class using the **__call__** method. **If we want to create an object that acts like a function, the function decorator must return an object that acts like a function**, which is where the **__call__** method comes into action or use.

Class decorator: Example

What is a class decorator

```
#!/usr/bin/env python
"""
A program to demonstrate class decorator
"""
class Decorator:
    def __init__(self, function):
        print("This is constuctor ")
        self.function = function

    def __call__(self):
        # before function call code
        print("Before function call")
        self.function()

        # after function calls code
        print("After function call")
```

Class decorator: Example

What is a class decorator

```
# addition class decorator to the function
@Decorator
def function():
    print("Class decorator")
# function call
function()
```

Output

```
This is constuctor
Before function call
Class decorator
After function call
```

Recursive calls to functions

Implementation of recursive call and functions

A recursive call is a method in Python programming or problem solving which involves a function calling itself one or more times within the function body. This process of a function calling itself is know as **recursion**. Ideally, the recursive method returns the function call values. ***A function is defined as recursive , if its operation meets the recursion condition.*** As you know, in Python programming, we can call a function inside another function. In the same way, we can also call a function recursively inside its body (**recursion**).

Recursive function have two main cases base case and standard case or recursive case.

Recursive calls to functions: Example

Implementation of recursive call and functions

```
#!/usr/bin/env python
"""
A program to demonstrate recursive function
"""
# A function to computer the factorial of a number
def fact(number):
    if number == 1: # base case
        return 1
    #standard case or recursive case
    else:
        return(number * fact(number - 1))
numb = 7
print("The factorial of", fact(numb) )
```

Output

```
The factorial of 7 is 5040
```

Recursive *factorial class: Example*

Implementation of recursive call in a class

```
#!/usr/bin/env python
"""
A program to demontrate recursive factorial class
"""
class Factorial:
    def factorial(self, number):
        if(number < 0):
            return print("Factorial does not required negative entry.")
        # base case
        elif(number == 0):
            return 1

        # standard case
        else:
        # calling the factorial method recursively using self keyword
            return ((number * self.factorial(number-1)))
```

Recursive *factorial class: Example*

Implementation of recursive call in a class

```
17 # take user input
18 number = int(input("Enter number: "))
19 # creating the object
20 factorialObj = Factorial()
21
22 factorial_calculation = factorialObj.factorial(number)
23 print("The factorial of {}! is = {}".format(number,
       factorial_calculation))
```

Recursive *factorial class: Example*

Implementation of recursive call in a class

```
#!/usr/bin/env python
"""
A program to demontrate recursive class
"""
class Factorial:
    def factorial(self, number):
        if(number < 0):
            return print("Factorial does not required negative entry.")
        # base case
        elif(number == 0):
            return 1

        # standard case
        else:
        # using the class name to invoke the factorial method
            return (number * Factorial.factorial(self, number-1))
```

Recursive *factorial class: Another Example*

Implementation of recursive call in a class

```
17 # take user input
18 number = int(input("Enter number: "))
19 # creating the object
20 factorialObj = Factorial()
21
22 factorial_calculation = factorialObj.factorial(number)
23 print("The factorial of {}! is = {}".format(number,
       factorial_calculation))
```

Programming *Conventional Facts*

General facts about program construction

Normally, before we write a program to solve a particular problem, **it is essential to have clear and thorough understanding of the the problem and carefully planned approach to solving the problem**. Ideally, when writing a program, *it is equally essential to understand the types of building blocks that are readily available and to consider using conventional* **program-construction principles**, such as modularity of the program constructs, control flow structures and so on.

Think before you write your program. Read before you think. - Originally paraphrase from **"Think before you speak. Read before you think"** by **Fran Lebowitz**

Pseudocode

What is a pseudocode

Pseudocode is an artificial and informal language that helps programmers develop **algorithms**. Pseudocode consists of ***descriptions of executable statements*** that are executed when the program has been converted from pseudocode to Python programming constructs. In this section, we would introduce you to pseudocode examples that would provide a useful insight knowledge of developing algorithms that would be converted to **Python programs**. Pseudocode is similar to everyday English; it is convenient and user-friendly, although it is not an actual computer programming language. Pseudocode programs are not executed on computers. Rather, *pseudocode helps the programmer **"plan"** a program before attempting to write it in a programming language, such as Python*. In this section, we provide some examples of how **pseudocode** can be used effectively in developing Python programs.

Pseudocode

What is a pseudocode

Pseudocode ***is an informal instructional steps for performing a given tasks***. This are made up of **natural language understanding or description of a given task**. *The steps are written in non-computer understandable language, but you would soon learn how to compute and formulate them in Python.* **Note that, there are no formal restriction or strict requirements for pseudocode formulation because it is read by humans and not by any computer program**. Here are example of pseudocode. Let's see how we could use pseudocode to describe decisions and repetitions.

- *If* $total\_sale\_1 \leq total\_sale\_2$
- *While the balance is less than* $50,000$
- *For each value in the sequence*

Pseudocode

What is a pseudocode

A carefully prepared **pseudocode program** can be converted easily to a corresponding Python program. In many cases, this is done simply by replacing the pseudocode statements with their **Python equivalent constructs**.

Let's see how we could write a pseudocode to describe how a value is change or set.

- *total cost = purchase price + operation cost*
- *Multiply value of the balance by 0.05*
- *Display the current balance*

Pseudocode

What is a pseudocode

Let's see how we could write a pseudocode to indicate which statements should be selected or repeated.

For each car:

- *operating cost = 10 x annual fuel cost*
- *total cost = purchase price + operating cost*

Here, the indentation indicates that both statements should be executed for each car.

Let's indicate results with statements such as:

- *Choose car2.*
- *Report year of manufacturing as the answer.*

Pseudocode

What is a pseudocode

The wording is not important, most important factor is that the pseudocode describes a sequence of steps that is:

- Unambiguous
- Executable
- Terminating

The step sequence is unambiguous ***when there are precise instructions for what to do at each step and the next step***. A step can only be executable ***when in practice this can be carried out successfully in real-life***. For example if the interest rate for sequence of bank accounts is not defined or set, there is no way we could know the exact percent for the interest rate to calculate. A sequence of steps is terminating ***if it will eventually come to a logical end***.

Algorithm

What is an algorithm

Algorithm is a sequence of instructional steps that is unambiguous,executable program written in a logical manner to solve a given task. ***The existing of an algorithm is essential for all programming task***. ***Ideally, we will need to define and describe algorithm sufficiently before we start writing our program code***. Any computing problem can be solved by executing a series of actions in a **specified order**. An algorithm is a *procedure* for solving a problem in terms of:

- actions to be executed and
- the order in which these actions are to be executed.

Algorithm control structures

What is an algorithm control structure

Let's look at an example to **demonstrate a specific order in which routine actions could be executed**. Consider the **"rise-and-shine"** algorithm followed by an **executive officer** for getting out of bed and going to work:

- (1) *Get out of bed*
- (2) *take off pajamas*
- (3) *take a shower*
- (4) *get dressed*
- (5) *eat breakfast*
- (6) *carpool to work*

This routine gets the executive to work on-time to make any critical decisions.

Algorithm control structures

What is an algorithm control structure

However, ***suppose that the same steps are performed in a slightly different order as follows***:

- (1) *Get out of bed*
- (2) *take off pajamas*
- (3) *get dressed*
- (4) *take a shower*
- (5) *eat breakfast*
- (6) *carpool to work*

In this case, our executive officer would show up for work soaking wet. Specifying the order in which statements are to be executed in a computer program is called **program control**.

Algorithm - *Example*

Example of an algorithm

Classic Strawberry Eton Mess

for Wimbledon

Ingredients:

500g strawberries
400ml double cream
3 ready-made meringue nests, crushed
sprigs of fresh mint, to garnish

Method:

Purée half the strawberries in a blender.
Chop the remaining strawberries, reserving four for decoration.
Whip the double cream until stiff peaks form, then fold in the strawberry purée, chopped strawberries and crushed meringue.
Spoon equal amounts of the mixture into 4 dishes.
Garnish with the remaining strawberries and a sprig of mint.

Algorithm - *Example*

Example of an algorithm

Blueberry Banana Pancakes

Serves: 4; Prep Time: 10 minutes; Cook Time: 10-15 minutes

INGREDIENTS

- Nonstick cooking spray
- 1 cup whole-wheat flour
- 1/2 cup all-purpose flour
- 2 T sugar
- 2 t baking powder
- 1/4 t salt
- 3 very ripe medium bananas
- 1 cup nonfat milk
- 1 egg
- 1 t vanilla extract
- 1 1/2 cup frozen blueberries (do not defrost)
- 4 T maple syrup

NUTRITION SCORE PER SERVING:
(3 pancakes plus 1 tablespoon maple syrup): 384 calories, 6% fat (3 g; < 1 g saturated), 83% carbs (80 g), 11% protein (11 g), 8 g fiber, 203 mg calcium, 3 mg iron, 401 mg sodium

Algorithm - *Pancake*

Example of a pancake algorithm

Let's look at a step-by-step preparatory direction to prepare fruit pancake.

Directions

- Preheat your over to 250 degrees F. Coat a griddle or large nonstick skillet with cooking spray and preheat.
- In a medium bowl, combine both flours, sugar, baking powder and salt.
- Mix well with a fork and set aside.
- In a large bowl or food processor, mash the bananas fruit until mushy. Add milk, egg and vanilla and mix or process until blended.
- And dry ingredients to the banana fruit mixture and mix or process until just blended (tiny lumps should still appear; do not over-mix or pancakes will be tough).

Algorithm - *Pancake*

Example of a pancake algorithm

- Ladle 3 tablespoons of the banana fruit pancakes batter onto hot griddle for each pancake. Top each with 1/2 tablespoons of blueberries.
- When bubbles appear around the edges of the banana blueberries pancakes, after about 2-3 minutes, flip and cook 1 minute.
- Transfer the banana blueberry pancakes to a warm plate and keep warm in a 250 degree F oven while you cook remaining pancakes.
- Serve the healthy banana blueberry pancakes with maple syrup over top.
- Enjoy the delicious results of your new healthy cooking pancakes with fruit recipe with family and friends!

Conceptual Theory of Algorithm Formulation

Progressive logic of developing algorithm

With the vast improvement of computational power, researchers, scientists, programmers and software developers and so on, **have gain more insights in using special mathematical or statistical algorithmic models to analyse and retrieve relevant information from datasets**. Nowadays, the ease of data collection and the availability of open source computation tools provide innovative methods of analysing data, these processes and techniques of collecting and analysing data is this way is collectively called **data science**. A branch of **data science** with the ability to **find patterns in vast amounts of data** is known as data mining. Using special mathematical tools we can find clusters of related data points. Some statistical methods can provide classification-algorithm for grouping different data into different categories.

Conceptual Theory of Algorithm Formulation

Scenario

Let's take a look at a popular recommendation algorithm scenario. A business identify different groups of customers from their past shopping behaviour and then make business decision by proving specific purchase recommendation in form of similar items for each group. Data mining techniques and algorithms can be used to provide this recommendation. Data mining is also useful for identifying abnormal behaviour, which might be detecting sign of fraud or an imminent danger to a person or property. Another powerful part of data science and Artificial Intelligence is Machine Learning. **Machine Learning provide us with the opportunity to build systems that loosely mimic the processing power of the human brains**. Machine Learning models can be trained with data or patterns that have been **pre-categorised**. After the ***training phase***, ***machine learning algorithms*** are able to **identify and recognise similar patterns**.

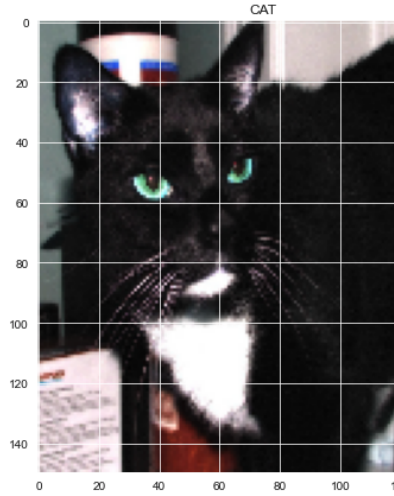
Conceptual Theory of Algorithm Formulation

Scenario

For example we can build a Neural Networks (NNs) or Convolutional Neural Networks (CNNs) on thousands of photos of **fish**, and then on a similar number of **cats**. *The machine learning or deep learning models can subsequently recognise with great reliability a given picture of as fish or cat and make a reliable prediction*.

Conceptual Theory of Algorithm Formulation

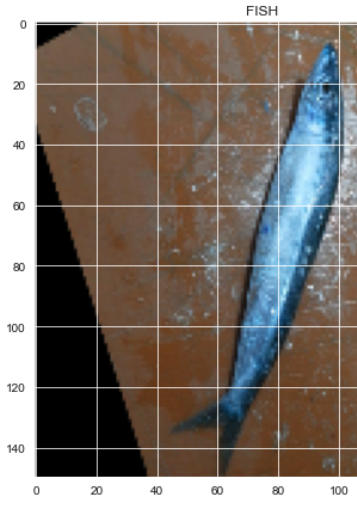
Scenario



We can create a pattern to identify a cat using the tail.

Conceptual Theory of Algorithm Formulation

Scenario



We can create a pattern to identify a fish using the scale.

Conceptual Theory of Algorithm Formulation

Scenario

This process and technique might not sound impressive at first. ***But think of how you might program such a task, given the special features of the objects (cats and fish)***. How would you develop or formulate an algorithm that can differentiate between a cat and a fish using only statement such as "if the object has a **tail** predict cat, otherwise predict fish". In this case, we will be using the features for cats **(tail)** and features for fish **(scale)**. A certain amount of algorithms and programming constructs are required to develop this logic fully.

Insertion Sort Algorithm

The swapping scenario

Insertion sort algorithm similar to **selection sort** also *splits the list into two halves*: the sorted sub-list and the unsorted sub-list, and starts with all items in the unsorted sub-list. Insertion sort, however, *does not search the unsorted sub-list for the next item to insert into the sorted sub-list, instead it takes the first item in the unsorted sub-list and repeatedly swaps it with the element to its left (starting with the last element in the sorted sub-list) until the next number to the left is smaller than the one being moved, or there are no more numbers to the left*. **This repeats until the unsorted sub-list contains no items**.

Insertion Sort Algorithm

The swapping scenario

©Mark Day (Midjourney)!

Insertion Sort Algorithm - *Example*

The swapping scenario

Using the list below, we will now see how insertion sort operates. Remember that the ^ **caret** or **circumflex** character symbol is used to denote the first item in the unsorted sub-list:

```
[5,1,12,-5,16,2,12,14]
 ^
```

The first step is to move the first element in the unsorted sub-list (5) into the sorted sub-list. As this is now the only element in the sorted sub-list, it is in the correct position so we don't need to perform any more actions.

Insertion Sort Algorithm - *Example*

The swapping scenario

We now repeat the steps of the algorithm. Firstly, we move the first item in the unsorted sub-list into the sorted sub-list and repeatedly swap it with the item on the left until it reaches the correct position.

```
[5,1,12,-5,16,2,12,14]
  ^
```

Insertion Sort Algorithm - *Example*

The swapping scenario

As you can see in the below, the number, 1 (the first item in the unsorted sub-list), was swapped with the item to the left (5, the last item in the sorted sub-list), this placed the number 1 before 5. As there are no more numbers to the left, the algorithm continues with the next number in the unsorted sub-list.

```
[5,1,12,-5,16,2,12,14]
   ^
[1,5,12,-5,16,2,12,14]
   ^
```

Insertion Sort Algorithm - *Example*

The swapping scenario

Because the first item in the unsorted sub-list (12) is larger than all items in the sorted sub-list, no swap occurs. The number is already in the correct position. There are numbers left in the unsorted sub-list, so the algorithm continues.

```
[5,1,12,-5,16,2,12,14]
      ^
```

Insertion Sort Algorithm - *Example*

The swapping scenario

Now, notice how the -5 (the first item in the unsorted sub-list) repeatedly swaps with the item to the left until it reaches the correct position.

```
[5,1,12,-5,16,2,12,14]
       ^
[5,1,12,-5,16,2,12,14]
         ^
[1,5,-5,12,16,2,12,14]
         ^
[1,-5,5,12,16,2,12,14]
         ^
[-5,1,5,12,16,2,12,14]
         ^
```

Insertion Sort Algorithm - *Example*

The swapping scenario

The algorithm continues in exactly the same way until the unsorted sub-list contains no elements.

```
[-5,1,5,12,16,2,12,14]
              ^
```

```
[-5,1,2,5,12,16,12,14]
                ^
```

```
[-5,1,2,5,12,12,16,14]
                  ^
```

Insertion Sort Algorithm - *Example*

The swapping scenario

At the end, as the unsorted sub-list contains no numbers, the algorithm terminates, and the list is sorted.

```
[-5,1,2,5,12,12,14,16]
```

Algorithm

The swapping scenario

Let's see how we could formalise the algorithm for this scenario.

1. ***Put all the elements of the list to be sorted in the unsorted sub-list***
2. ***Select the first element in the unsorted sub-list and move it to the end of the sorted sub-list***
3. ***Repeated swap the last element of the sorted sub-list with the element to its left, until the number to the left is smaller or the element is at the start of the sorted sub-list***
4. ***If the unsorted sub-list has elements in it, go to step 2, otherwise stop.***

Pseudocode

The swapping scenario

Instead of providing hints as you implement the algorithm, let's try to represent and interpret the algorithm using a **Pseudocode**. As you know, ***Pseudocode*** does not have a standard format like Python code structure, instead it's just a semi-structured method of representing an algorithm in a way that makes it easier to convert to proper code.

```
#!/usr/bin/env python
"""
Demonstrating Pseudocde for the insertion sort algorithm
"""
for i <- 1="" to="" length="" list="" j="" -="" i="" while="">  0 and
    list[j-1] > list[j]
    swap list[j] and list[j-1]
    j <- j="" -="" 1="" pre="">
```

Insertion Sort Algorithm - *Example*

The swapping scenario

```
#!/usr/bin/env python
"""
A program to demonstrate implementation of an insertion sort algorithm
"""
number = [5, 1, 12, -5, 16, 2, 12, 14] # list of eight numbers
# using for-loop and while-loop to iterate over the list
for i in range(1, len(number)):
    j = i
    while(j > 0 and number[j - 1] > number[j]):
        temp = number[j]
        number[j] = number[j - 1]
        number[j - 1] = temp
        j = j - 1 #j -= 1
print(number)
```

Insertion Sort Algorithm - *Class Example*

The swapping scenario

```
#!/usr/bin/env python
"""
A program to demonstrate implementation of an insertion sort algorithm
using a class, object and method
"""
class InsertionSort:
    def algorithm(self, number):
        for i in range(1, len(number)):
            j = i
            while(j > 0 and number[j - 1] > number[j]):
                temp = number[j]
                number[j] = number[j - 1]
                number[j - 1] = temp
                j = j - 1 #j -= 1
        return print(number)
```

Insertion Sort Algorithm - *Class Example*

The swapping scenario

```
17 number = [5, 1, 12, -5, 16, 2, 12, 14] # list of eight numbers
18 # create the object
19 insertionObj = InsertionSort()
20 insertionObj.algorithm(number)
```

Implementation of Insertion Sort Algorithm in Python

The swapping scenario interpretation

In this algorithm, the variable **'i'** acts as our **'position'** variable from before. It stores a pointer to the first item in the unsorted sub-list, *the only difference is that it is started at 1 instead of 0. The reason for this is that although all items are said to be in the unsorted sub-list when the algorithm starts, there is really no need to consider the first item as there is no item to the left of it to swap with.* **Therefore this psuedocode assumes the first element (element 0) is already in the sorted sub-list.** The variable **'i'** is slowly increased from **1** to the **number of elements in the list** *as each item is moved from the unsorted sub-list to the sorted sub-list and repeatedly swapped to the left until the element is in the correct place.* **'j'** is used to track the current position of the element in question, being reduced by **1 every time the element is swapped to the left.**


Summary

Summary

Let's revise the concepts of today's lecture

In this lecture we discuss the following:

- We looked at classes, objects, class and static methods
- discussed testing a class using **unit testing** approach.
- Decorators are powerful and valuable Python tool. They allow us to modify the behaviour of a function or a class.
- Pseudocode is an informal description of a sequence of steps for solving a problem.
- An algorithm for solving a problem is a sequence of steps that is unambiguous, executable, and terminating.

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 1 Introduction and 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 3 Branching, while Loops, and Program Planning: The Guess My Number Game, 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 6

In week 6

Lecture 6: Recursion in Python