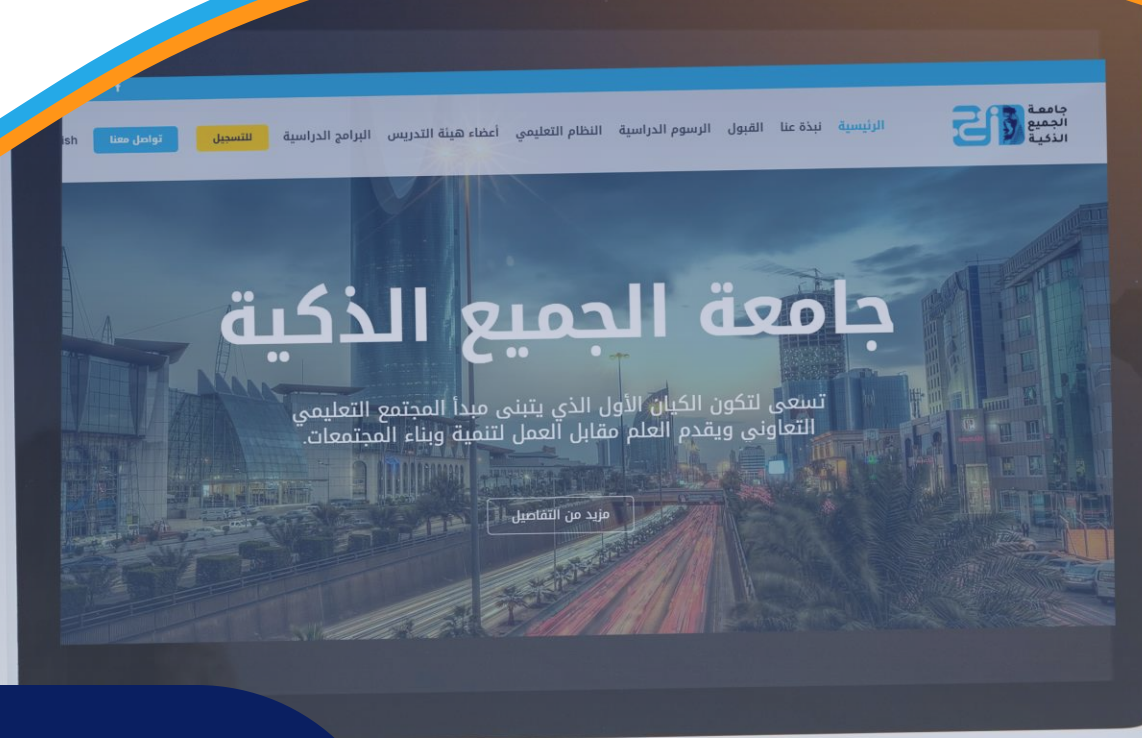




جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY



Object-Oriented Programming
Prepared by Dr. Hamada I. AbdulWakel
Lecture 8

CYB550
Programming Foundations In
Python

- Know what the meaning of Object-oriented programming.
- Explain the behavior of classes and objects.
- Define a constructor, instance variables, and methods for a class of objects.
- List the methods, including their parameters and return types, that realize the behavior of a class of objects.



- Program is divided into a series of statements, could contain functions.
- The main concern is on doing things.
- Poor modeling to real world problems.
- It is not easy to maintain program if it is too complex.
- Provide poor data security.
- Not extensible programming paradigm.
- Unit of programming is function.
- Ex: Pascal, C, Basic, Fortran.



Lecture 8

Procedural-Oriented Programming

Cont'd





- Suppose we need to model the students registered in a set of courses.
- There are more students in the classroom.
- Each student has name, age, address, and course name.
- You need to define!!!!



- It store single simple value.
- It was not a problem for a simple programs.
- Because of development and the need for complex programs, the need for group variables is required.
- Grouping variables together makes it easier.





- Program is divided into classes and objects.
- The main concern on data.
- Strong modeling to real world problems.
- It is easy to maintain program even if it is too complex.
- Provide strong data security.
- It is highly extensible programming paradigm.
- Unit of programming is class.
- Ex: Python, Java, C#.



- Classes: are **user-defined data types** that act as the blueprint for individual objects, attributes and methods.
- Objects: are **instances** of a class created with specifically defined data.

Lecture 8

Building Blocks of OOP

Cont'd





- Attributes: are defined in the class template and represent **the state of an object**. Class attributes belong to the class itself..
- Methods: are functions that are defined inside a class that describe **the behaviors of an object**.

Lecture 8

Main Principles of OOP





- Encapsulation: all important information is contained inside an object and only select information is exposed. The implementation and state of each object are privately held inside a defined class.
- Abstraction: Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code.

- Inheritance: Classes can reuse code from other classes. Relationships and subclasses between objects can be assigned, enabling developers to reuse common logic while still maintaining a unique hierarchy.
- Polymorphism: Objects are designed to share behaviors and they can take on more than one form.

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

```
class ClassName:  
    # body
```

- Example:

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

- The properties that all objects **must have** are defined in a method called `__init__()`.
- Every time a new object is created, `__init__()` sets **the initial state** of the object by assigning the values of the object's properties.
- The `__init__()` can take any number of parameters, but the first parameter will always be a variable called **self**.

- Example:

class Student:

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

- `self.name` creates an attribute called `name` and assigns to it the value of the `name` parameter.

- The class attributes are attributes that have **the same value** for all class instances.
- Class attributes are defined directly **beneath the first line of the class name**.
- You can define a class attribute by assigning a value to a variable name **outside** the `__init__()` function.

- Example:

class Student:

```
    course = "OOP in Python"
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

- course name "OOP in Python" will assign to every student.

- Creating a new object from a class is called **instantiating an object**.
- You can instantiate a new object by typing the name of the class, followed by opening and closing parentheses.

- Example:

```
class Student:
```

```
    course = "OOP in Python"
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
std1 = Student("Ahmed", 20)
```

```
std2 = Student("Mohamed", 15); print(std1); print(std1 is std2)
```



- After you create the instances, you can access their instance attributes using **dot notation**.
- Although the attributes are guaranteed to exist, their values can be changed dynamically.
- The custom objects are **mutable** by default.
- You can delete the instance or its attribute using the keyword **del**.



- Example:

```
std1 = Student("Ahmed", 20)
print(std1.name)
print(std1.course)
std1.age = 25
print(std1.age)
del std1.age
print(std1.age)
del std1
```



- Functions that are defined **inside** a class and can **only be called** from an instance of that class.
- Like `__init__()`, an instance method's first parameter is always **self**.
- The instance methods of the same name with different implementation, the second implementation of the method prevails over the first one.

- Example:

```
class Student:
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
    def displayStudentInfo(self):
```

```
        return f"{self.name} is {self.age} years old"
```

```
s1 = Student("Ahmed", 15)
```

```
print(s1.displayStudentInfo())
```



- Example:

```
class Student:
```

```
    def display(self):
```

```
        print(self.name)
```

```
    def display(self):
```

```
        return f"The student name is {self.name}"
```

```
s1 = Student("Ahmed", 15)
```

```
print(s1.display())
```



- Like `__init__()`, a dunder method is always begin and ends with **double underscores** `__`.
- Used to override functionality for built-in functions for custom classes.
- The `__str__()` method is used to change what gets printed when print a special instance of the class.

- Example:

```
class Student:
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
    def __str__(self):
```

```
        return f"{self.name} is {self.age} years old"
```

```
s1 = Student("Ahmed", 15)
```

```
print(s1)
```





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

class Distance:

```
def __init__(self, feet = 0, inch = 0.0):
```

```
    self.feet = feet
```

```
    self.inch = inch
```

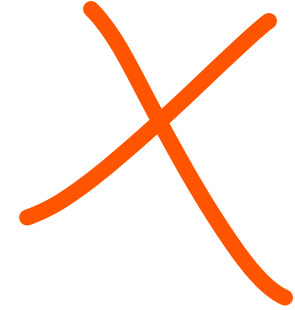
```
def getDistance(self):
```

```
    f = int(input("Enter the distnace feet value: "))
```

```
    self.feet = f
```

```
    inc = float(input("Enter the distnace feet value: "))
```

```
    self.inch = inc
```





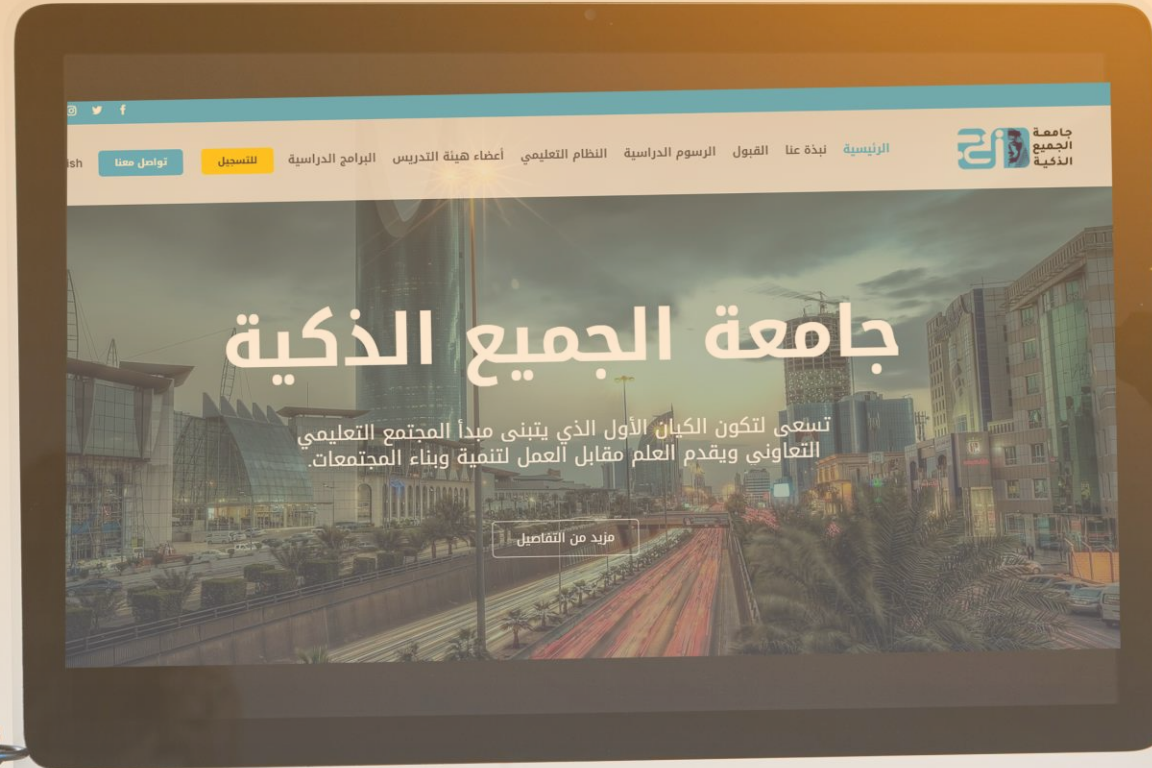
class Distance:

.....

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



```
d1 = Distance()  
print(d1.feet)  
d1.getDistance()  
print(d1.feet)  
d2 = Distance(2, 1.5)  
d3 = d1.addDistances(d2)  
print(p3.feet, p3.inch)
```



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

THANK YOU