M110: Python Programming

Meeting #10

Classes and Object-Oriented Programming

AOU
الجامعة العربية المفتوحة
Arab Open University
Lebanon

Prepared by Dr. Ahmad Mikati

Topics

- Procedural and Object-Oriented Programming
- Classes
- Working with Instances
- Techniques for Designing Classes
- Inheritance

Procedural Programming

Procedural programming: is a method of writing software. It is a programming practice centered on the procedures or actions that take place in a program.

Procedural programming is in short writing programs made of functions that perform specific tasks

- Procedures typically operate on data items that are separate from the procedures
- Data items commonly passed from one procedure to another
- Focus: to create procedures that operate on the program’s data

Object-Oriented Programming (1 of 2)

Object-oriented programming: Whereas procedural programming is centered on creating procedures (functions), object-oriented programming (OOP)is focused on creating objects

Object: An object is a software entity that contains data and procedures.

An object’s ***data attributes*** are simply variables that reference data.

The procedures that an object performs are known as methods.

An object’s **methods** are functions that perform operations on the object’s data attributes.

The object is, conceptually, a self-contained unit that consists of data attributes and methods that operate on the data attributes.

- Data is known as *data attributes* and procedures are known as *methods*.
  - Methods perform operations on the data attributes.

Object-Oriented Programming (2 of 2)

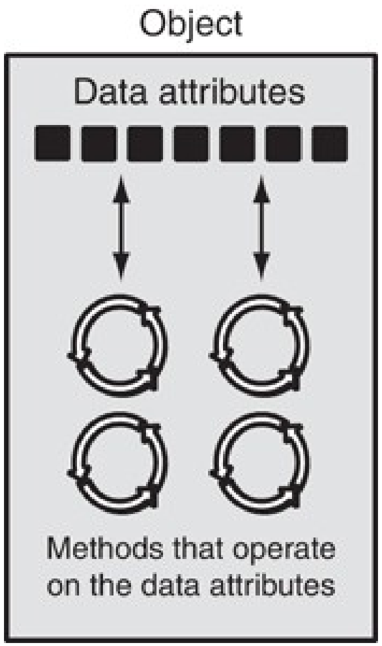


Figure 1: An object contains data attributes and methods

Classes

Before an object can be created, it must be designed by a programmer. The programmer determines the **data attributes** and **methods** that are necessary, then creates a *class*.

- **Class**: is a code that specifies the **data attributes** and **methods** of a particular type of object. A class is considered as a factory to create objects.
  - Like a blueprint* of a house. The blueprint itself is not a house but is a detailed description of a house.
  - A Python class uses variables (attributes) to define data fields and methods to define behaviors.
- **Instance**: is an **object** created from a class
  - Like a specific house built according to the blueprint.
  - There can be many instances of one class. Each house is a separate instance of the house described by the blueprint.

* blueprint :a design plan or other technical drawing

Classes

A way of thinking about the difference between a class and an object is to think of the difference between a cookie cutter and a cookie. While a cookie cutter itself is not a cookie, it describes a cookie. The cookie cutter can be used to make several cookies, as shown in Figure 3. Think of a class as a cookie cutter, and the objects created from the class as cookies.

So, a class is a description of an object's characteristics. When the program is running, it can use the class to create, in memory, as many objects of a specific type as needed.

Each object that is created from a class is called an *instance* of the class.

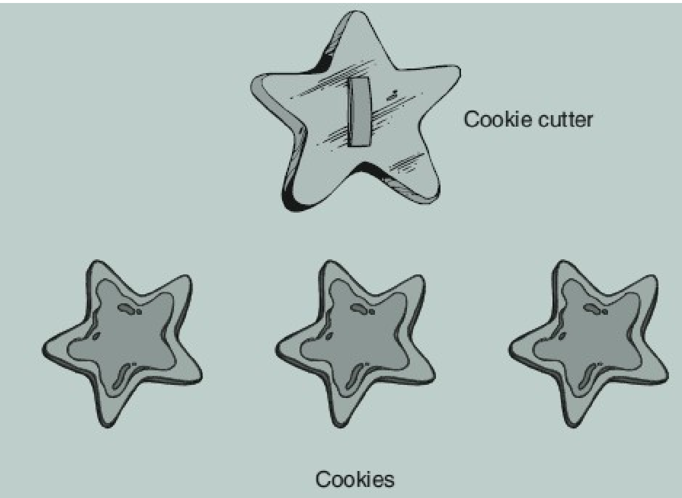


Figure 3 the cookie cutter metaphor

Classes

Instances have two characteristics: They have states and behaviors (an instance has attributes and methods attached to it)

Attributes represent its state, and methods represent its behavior.

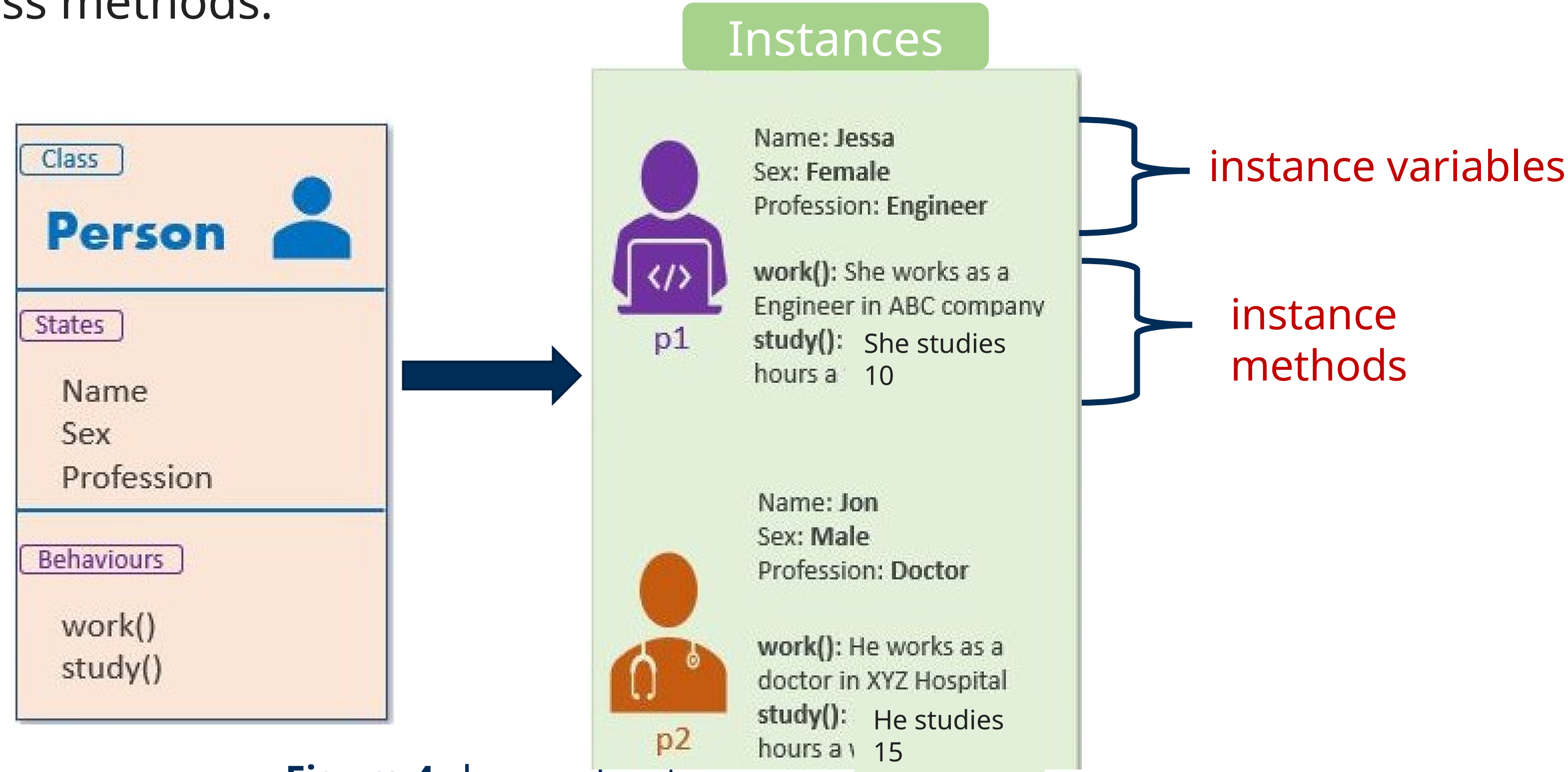
Using its methods, we can modify its state.

In short, Every instance has the following properties:

- **Identity**: Every instance must be uniquely identified.
- **State**: An instance has an attribute that represents a state of an instance, and it also reflects the property of an instance.
- **Behavior**: An instance has methods that represent its behavior.

Classes

For example, if we design a class based on the states and behaviors of a Person, then States can be represented as instance variables and behaviors as class methods.


Class
Person
States
Name
Sex
Profession
Behaviours
work()
study()
Instances
p1
Name: Jessa
Sex: Female
Profession: Engineer
work(): She works as a Engineer in ABC company
study(): She studies hours a 10
p2
Name: Jon
Sex: Male
Profession: Doctor
work(): He works as a doctor in XYZ Hospital
study(): He studies hours a 15

instance variables

instance methods

Figure 4 class and instances in Python

Both instances (p1 and p2) are created from the same class, but they have different states and behaviors.

Class Definitions

In Python, class is defined by using the **class** keyword.
The syntax to create a class is given below.

```
class class_name:
    '''This is a docstring. I have created a new 
class'‘’
     <statement 1>
      <statement 2>
       .
       .
       <statement N>
```

To create a class, you write a *class definition*.

- *class_name:* It is the name of the class
  - Class names often start with uppercase letter
- *Docstring:* It is the first string inside the class with a brief description of the class.
- *statements:* Attributes and methods

Method definition is like any other python function definition

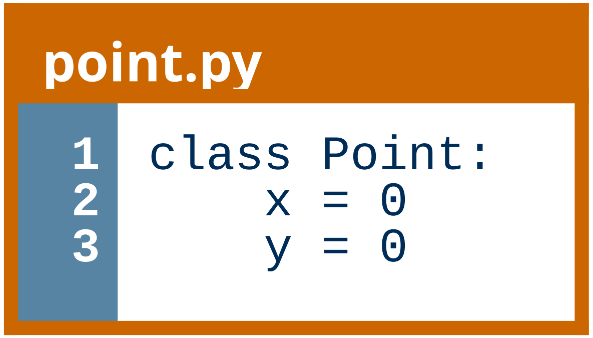
- ***self*** **parameter**: required in every method in the class – references the specific object that the method is working on.

Class Definitions- example

- Example:

```
class Point:
        x = 0
        y = 0

# main
p1 = Point()
p1.x = 2
p1.y = -5
```



point.py

```
class Point:
      x = 0
      y = 0
```

Using a Class

`import` **class**

- client programs must import the classes they use

point_main.py

```
from Point import *

# main
p1 = Point()
p1.x = 7
p1.y = -3
...

# Python objects are dynamic (can add fields
any time!)
p1.name = “Salim Hamad"
```

Object Methods

def **name**(***self***, **parameter, ..., parameter**):

statement(s)

- ***self*** *must* be the first parameter to any object method.
  - represents the "implicit parameter" (*this* in Java)
- Although you must specify *self* explicitly when defining the method, you don't include it when calling the method.
- *Python passes it for you automatically.*

Defining a method in class **Point:**

```
class Point:
  .....
  .....
   def translate(self, dx, dy):
      self.x += dx
      self.y += dy
   ...
```

Calling a method:

```
...
>>> p1.translate(1.5,2)
```

Calling Methods

- The user can call the methods of an object in two ways:
  - (the value of self can be an implicit or explicit parameter)

1) **object.method**(**parameters**)

or

2) **Class.method**(**object**, **parameters**)

- Example:

p1.translate(1, 5)

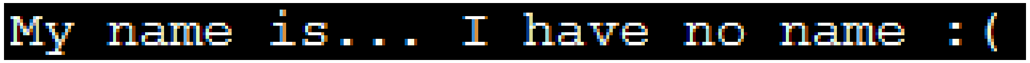
Point.translate(**p**, 1, 5)

Class Methods: **Example**

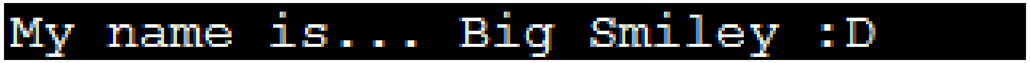
```
class Person:
   name = "I have no name :("
   def sayName(self):
      print("My name is...", self.name)

def main():
   aPerson = Person()
   aPerson.sayName()
   aPerson.name = "Big Smiley :D"
   aPerson.sayName()

main()
```



```
My name is... I have no name :(
```



```
My name is... Big Smiley :D
```

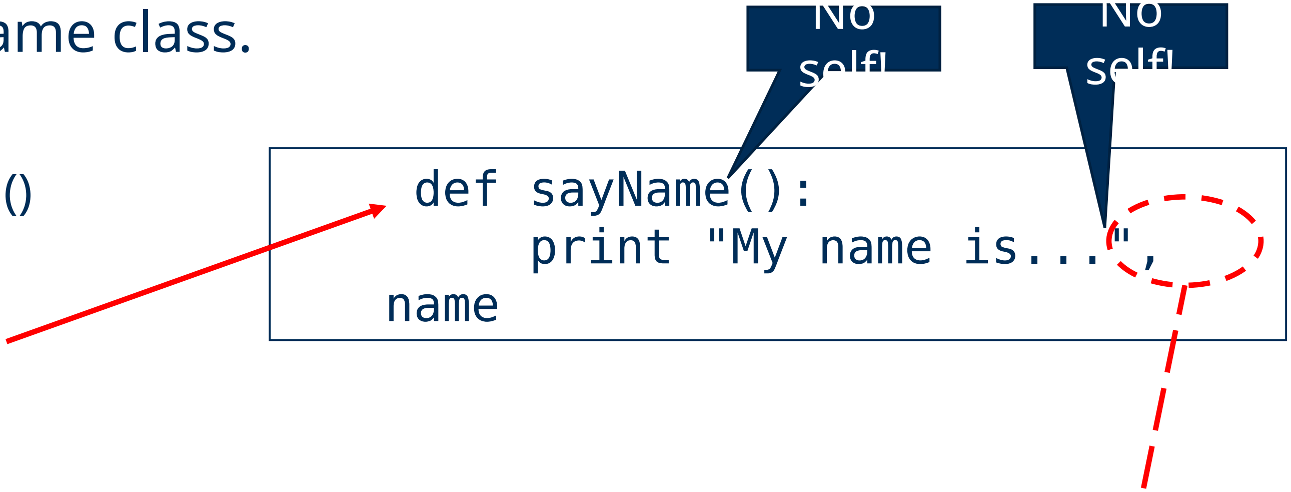
What Is The ‘Self’ Parameter

- **Reminder:** When defining/calling methods of a class there is always at least one parameter.
- This parameter is called the ‘**self**’ reference which allows an object to access attributes inside its methods.
- ‘**self**’ is needed to distinguish the attributes of different objects of the same class.
- Example:

bassem = Person()

lisa = Person()

lisa.sayName()



```
    def sayName():
        print "My name is...",
name
```

Whose name is this?
(This won’t work!)

The Self Parameter: A Complete Example

```
class Person:
   name = "I have no name :("
   def sayName(self):
      print("My name is", self.name)
def main():
   lisa = Person()
   lisa.name = "Lisa Haddad, pleased to meet you."
   bassem = Person()
   bassem.name = "Bassem Hassan, who are you???!!!"
   lisa.sayName()
   bassem.sayName()

main()
```

Recap: Accessing Attributes & Methods

- **Inside the class definition** (inside the body of the class methods)
  - Preface the attribute or method using the ‘**self**’ reference

```
class Person:
    name = "No-name"
    def sayName(self):
        print("My name is", self.name)
```

- **Outside the class definition**
  - Preface the attribute or method using the **name of the reference** used when creating the object.

```
def main():
    lisa = Person()
    bart = Person()
    lisa.name = "Lisa Haddad, pleased to meet you."
```

Class Definitions: Initializing The Attributes

- Classes have a special method that can be used to initialize the starting values of a class to some specific values.
- This method is automatically called whenever an object is created.
- **Format**:

Two underscores without spaces between them

```
class <Class name>:
    def __init__(self, <other parameters>):
    <body of the method>
```

- **Example**:

```
class Person:
  name = ""
  def __init__(self):
     self.name = "No name"
```

This design approach is consistent with many languages

Class Definitions

- **Initializer method**: automatically executed when an instance of the class is created
  - Initializes object’s data attributes and assigns self parameter to the object that was just created
  - Format: **def __init__ (self):**
  - Usually, the first method in a class definition

```
class Point:
                def __init__(self, ax, ay):
                    self.x = ax
                    self.y = ay
```

```
class Coin:
    # The _ _init_ _ method initializes the sideup data attribute with 'Heads’.
    def _ _init_ _(self):
        self.sideup = 'Heads'
```

Class Definitions

- To create instances of a class, you call the class using class name (and pass in whatever arguments its __init__ method accepts).
  - Format: ***My_instance = Class_Name()***
- To call any of the class methods using the created instance, use dot notation
  - Format: ***My_instance.method()***
  - Because the self parameter references the specific instance of the object, the method will affect this instance
    - Reference to self is passed automatically

Initializing The Attributes Of A Class

- Because the ‘**init()**’ is a method, it can also be called with parameters which are then used to initialize the attributes.
- **Example**:

```
# Attribute is set to a default in the class definition and then the
# attribute can be set to a non-default value in the init() method.
# (Not standard Python but a common approach with many languages)
class Person:
    name = "Default name" # Create attribute here
    def __init__(self, aName):
        self.name = aName
```

OR

```
# Create the attribute in the init() method. (Approach often used in Python).
class Person:
    def __init___(self, aName):
        self.name = aName  # Create attribute here
```

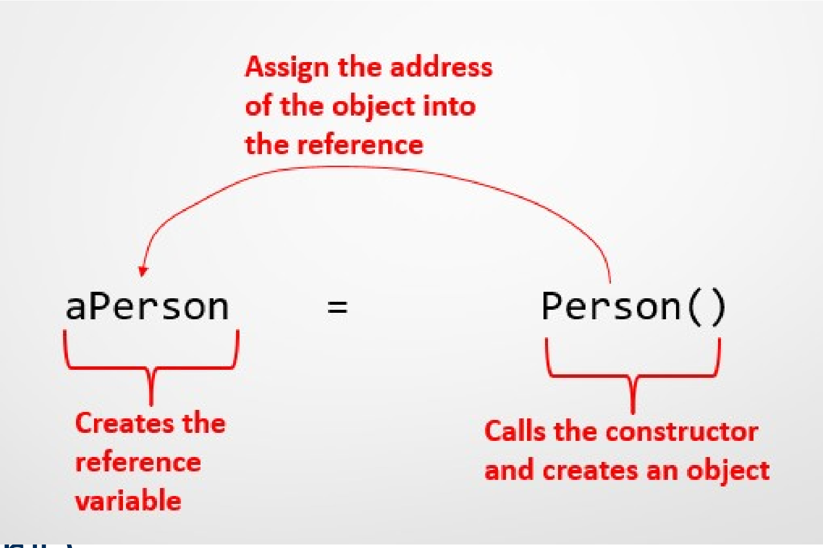
Using The “Init()” Method-Example

```
class Person:
   name = “No name"

   def __init__(self, aName):
       self.name = aName

def main():
   aPerson = Person(“Jamal Nader”)
   print(aPerson.name)

main()
```



Assign the address of the object into the reference

aPerson = Person()

Creates the reference variable

Calls the constructor and creates an object

Output:
Jamal Nader

Person Class Example

Write a Python class Person that represents a person with the attributes **name**, **age**, and **profession**. The class should have an initializer method __init__() to initialize the instance variables. It should also have two instance methods: **show**() to display the name, age, and profession of the person, and **work**() to display the person's name and profession.

The program should also:

- Prompt the user to enter their name, age, and profession.
- Create an object of the Person class using the entered values.
- Call the show() and work() methods on the object to display the person's details and profession, respectively.

Person Class-Solution

```
class Person:
    def __init__(self, aname, anage, aprofession):
        # data members (instance variables)
        self.name = aname
        self.age = anage
        self.profession = aprofession

    # Behavior (instance methods)
    def show(self):
        print('Name:', self.name, '\tAge:', self.age, '\tProfession:',
self.profession)

    # Behavior (instance methods)
    def work(self):
        print(self.name, 'is working as a/an', self.profession)

myname = input("Enter the name: ")
myAge = input("Enter the Age: ")
myProfession = input("Enter the profession: ")

p = Person(myname, myAge, myProfession)
p.show()
p.work()
```

Computation class Example

You are requested to write a python program that does the following:

a. Create **Computation class** with a default constructor (without parameters) allowing to perform various calculations on integers numbers.

b. Create a method called **Factorial()** in the above class which allows to calculate the factorial of an integer, n.

c. Create a method called **Sum()** in the above class allowing to calculate the sum of the first n integers 1 + 2 + 3 + .. + n.

d. Instantiate the class, prompt the user to enter an integer, and write the necessary statements to test the above methods.

Computation class - Solution

```
class Computation:
    def __init__ (self):
        self.n=0
# --- Factorial ------------
    def factorial(self, n):
        j = 1
        for i in range (1, n + 1):
            j = j * i
        return j
# --- Sum of the first n numbers ----
    def sum (self, n):
        j = 0
        for i in range (1, n + 1):
            j = j + i
        return j
x= Computation()
n=int(input("Enter an integer: "))
print("The factorial of the number",n, "is:",x.factorial(n))
print("The sum from the number",n, " to 1 is:",x.sum(n))
```

Class Attributes vs Instance Attributes in Python

Class attributes are the variables defined directly in the class that are shared by all objects of the class.

Instance attributes are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.

Class Attribute	Instance Attribute
Defined directly inside a class.	Defined inside a constructor using the self parameter.
Shared across all objects.	Specific to object.
Accessed using class name as well as using object with dot notation, e.g. classname.class_attribute or object.class_attribute	Accessed using object dot notation e.g. object.instance_attribute
Changing value by using: classname.class_attribute = value will be reflected to all the objects.	Changing value of instance attribute will not be reflected to other objects.

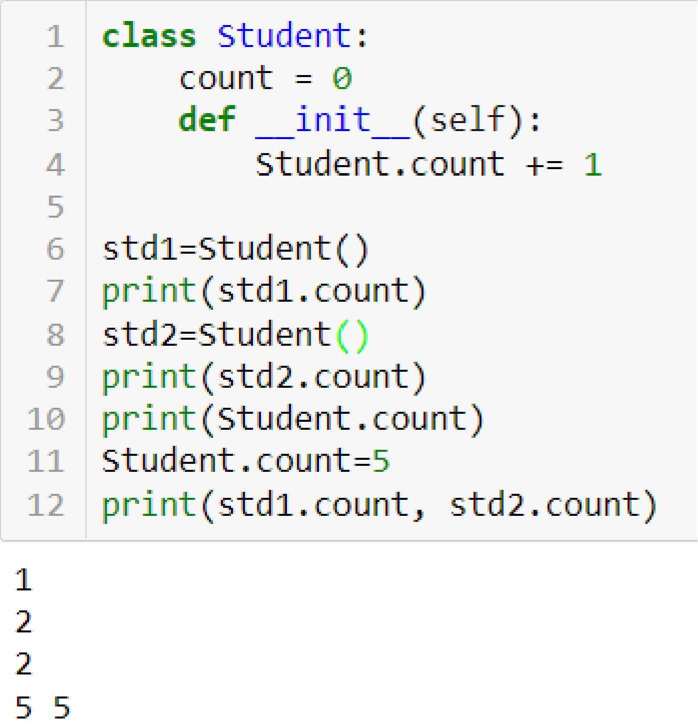
Class Attributes vs Instance Attributes in Python

The following example demonstrates the use of class attribute *count*.

```
class Student:
   count = 0
   def __init__(self):
      Student.count += 1
```

```
std1=Student()
print(std1.count)
std2=Student()
print(std2.count)
print(Student.count)
Student.count=5
print(std1.count, std2.count)
```

In this example, *count* is an attribute in the Student class. Whenever a new object is created, the value of count is incremented by 1.

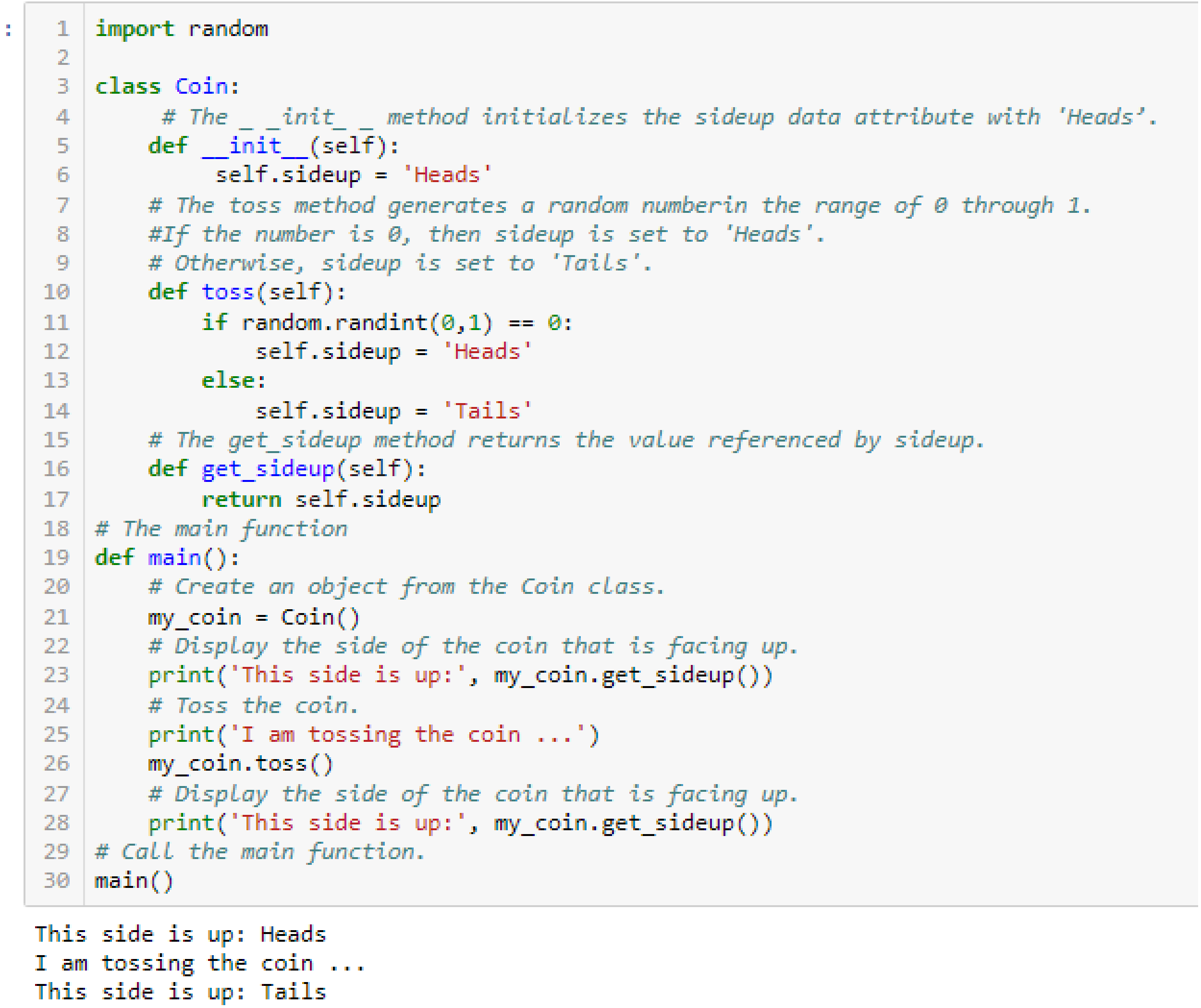


```
class Student:
    count = 0
    def __init__(self):
        Student.count += 1

std1=Student()
print(std1.count)
std2=Student()
print(std2.count)
print(Student.count)
Student.count=5
print(std1.count, std2.count)
```

```
1
2
2
5 5
```

Coin Class Example



```
 1  import random
 2  
 3  class Coin:
 4       # The _ _init_ _ method initializes the sideup data attribute with 'Heads'.
 5      def __init__(self):
 6           self.sideup = 'Heads'
 7      # The toss method generates a random numberin the range of 0 through 1.
 8      #If the number is 0, then sideup is set to 'Heads'.
 9      # Otherwise, sideup is set to 'Tails'.
10      def toss(self):
11          if random.randint(0,1) == 0:
12              self.sideup = 'Heads'
13          else:
14              self.sideup = 'Tails'
15      # The get_sideup method returns the value referenced by sideup.
16      def get_sideup(self):
17          return self.sideup
18  # The main function
19  def main():
20      # Create an object from the Coin class.
21      my_coin = Coin()
22      # Display the side of the coin that is facing up.
23      print('This side is up:', my_coin.get_sideup())
24      # Toss the coin.
25      print('I am tossing the coin ...')
26      my_coin.toss()
27      # Display the side of the coin that is facing up.
28      print('This side is up:', my_coin.get_sideup())
29  # Call the main function.
30  main()
```

```
This side is up: Heads
I am tossing the coin ...
This side is up: Tails
```

N.B: In line 1, we import the random module. This is necessary because we use the randint function to generate a random number

```
import random
class Coin:
     # The _ _init_ _ method initializes the sideup data attribute with
'Heads’.
    def __init__(self):
         self.sideup = 'Heads'
    # The toss method generates a random number in the range of 0
through 1.
    #If the number is 0, then sideup is set to 'Heads'.
    # Otherwise, sideup is set to 'Tails'.
    def toss(self):
        if random.randint(0,1) == 0:
            self.sideup = 'Heads'
        else:
            self.sideup = 'Tails'
    # The get_sideup method returns the value referenced by sideup.
    def get_sideup(self):
        return self.sideup
# The main function
def main():
    # Create an object from the Coin class.
    my_coin = Coin()
    # Display the side of the coin that is facing up.
    print('This side is up:', my_coin.get_sideup())
    # Toss the coin.
    print('I am tossing the coin ...')
    my_coin.toss()
    # Display the side of the coin that is facing up.
    print('This side is up:', my_coin.get_sideup())
# Call the main function.
main()
```

Summary

- This lecture covered:
  - Procedural vs. object-oriented programming
  - Classes and instances
  - Class definitions, including:
    - The self parameter
    - Data attributes and methods
    - initializer