Advanced Programming-Python (Module-3)

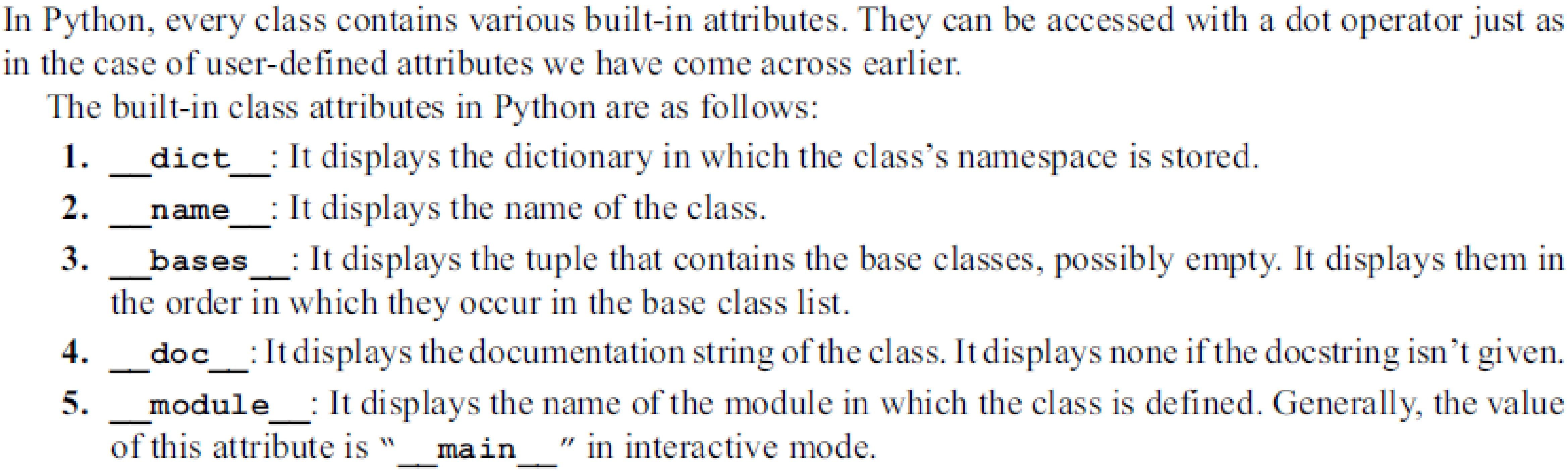
Presented by

Siddhanta Borah

Contents of Todays Discussion

- BUILT-IN CLASS ATTRIBUTES
- INHERITANCE
    - Multiple Inheritance
- METHOD OVERRIDING
- DATA ENCAPSULATION
- DATA HIDING

BUILT-IN CLASS ATTRIBUTES



In Python, every class contains various built-in attributes. They can be accessed with a dot operator just as in the case of user-defined attributes we have come across earlier.

The built-in class attributes in Python are as follows:

1. `__dict__`: It displays the dictionary in which the class's namespace is stored.
2. `__name__`: It displays the name of the class.
3. `__bases__`: It displays the tuple that contains the base classes, possibly empty. It displays them in the order in which they occur in the base class list.
4. `__doc__`: It displays the documentation string of the class. It displays none if the docstring isn't given.
5. `__module__`: It displays the name of the module in which the class is defined. Generally, the value of this attribute is "`__main__`" in interactive mode.

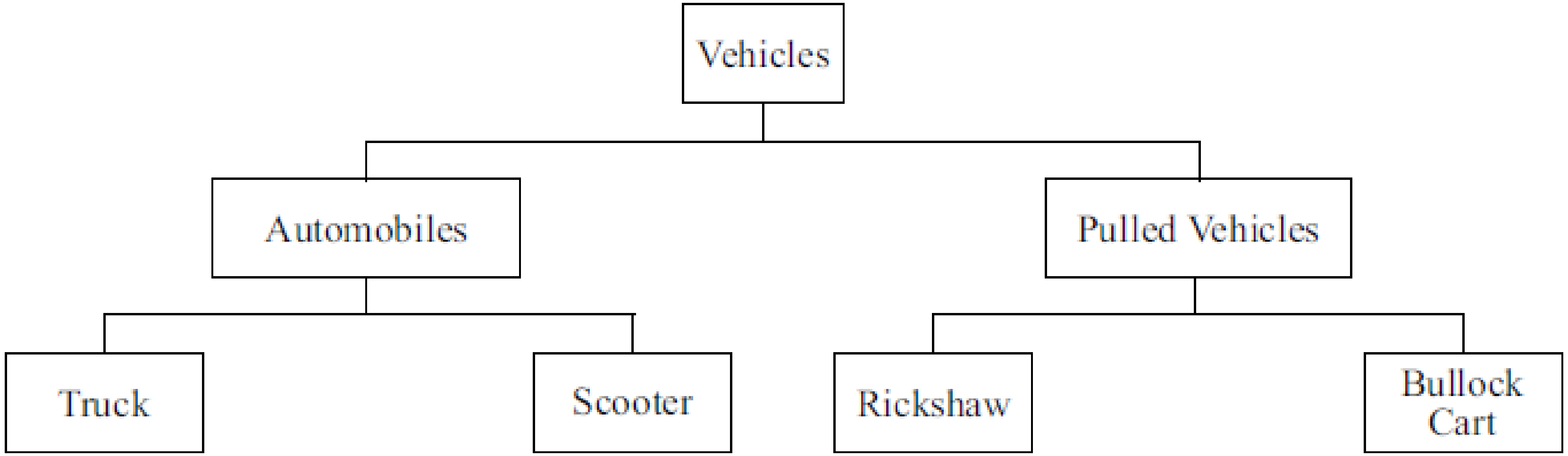
INHERITANCE

Inheritance is a very important concept in OOP. Inheritance, generally, means to acquire the features of something. In OOP, it means the reusability of code. It is the capability of a class to derive the properties of another class that has already been created.

Let us look at an example illustrated below:

- Vehicle is a class that is further divided into two subclasses, automobiles (driven by motors) and pulled vehicles (driven by men). Therefore, vehicle is the base class and automobiles and pulled vehicles are its subclasses. These subclasses inherit some of the properties of the base class vehicle.
- Truck and car are the subclasses of the class automobile that is the base class for them. They inherit some of the properties of base class automobiles. Similarly, the rickshaw and bullock cart are the subclasses of pulled vehicles that serves as the base class for them.

The main advantage of inheritance in the context of programming is that the code can be written once in the base class and then reused repeatedly in the subclasses.



Vehicles

Automobiles

Pulled Vehicles

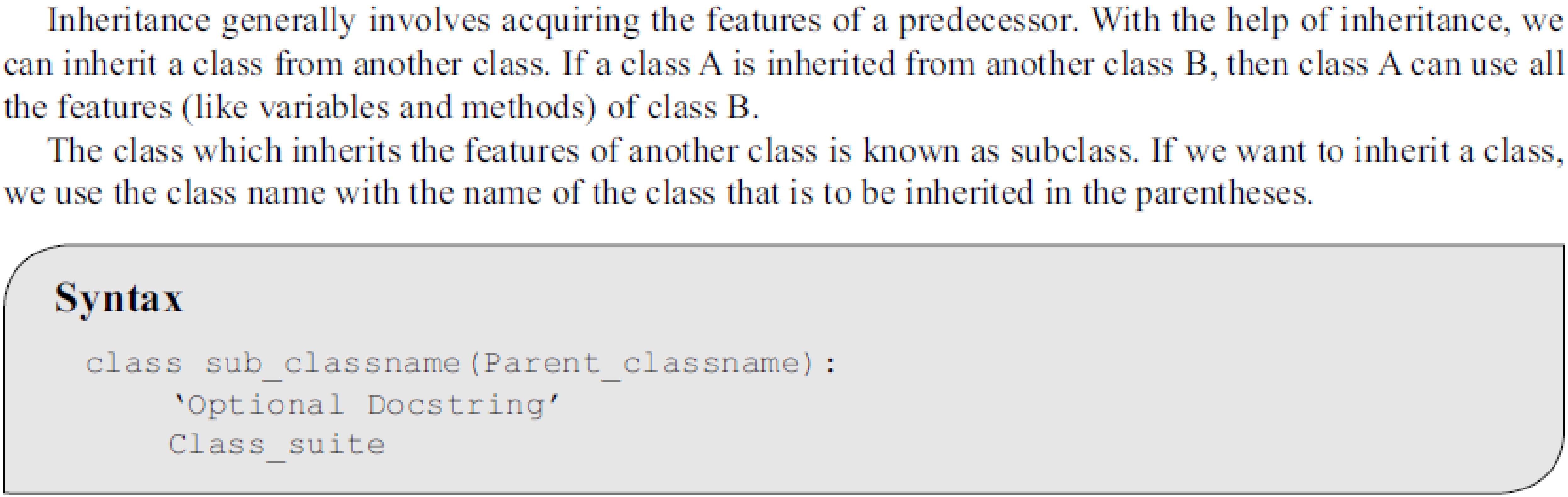
Truck

Scooter

Rickshaw

Bullock Cart

INHERITANCE (CONTD..)



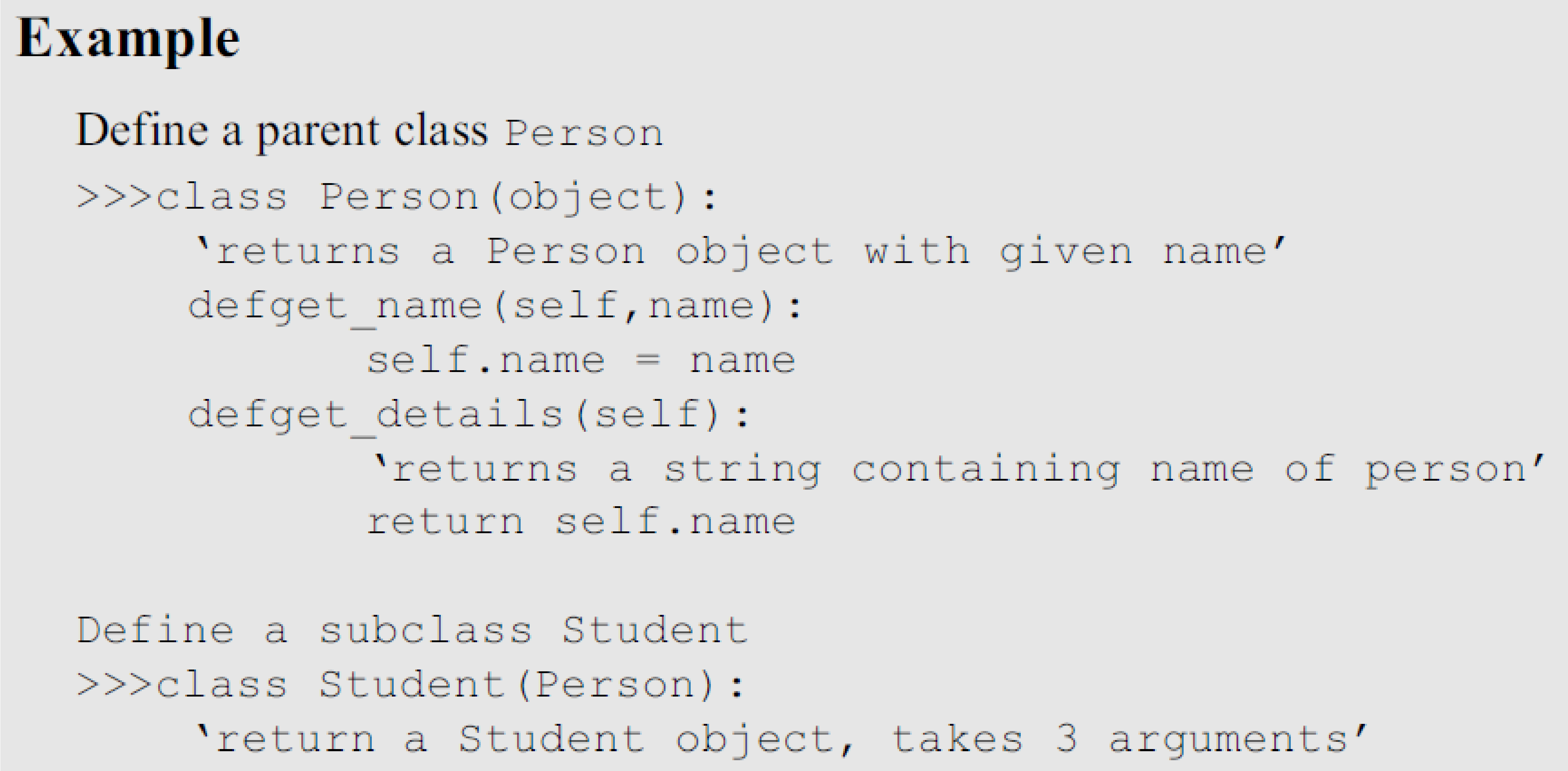
Inheritance generally involves acquiring the features of a predecessor. With the help of inheritance, we can inherit a class from another class. If a class A is inherited from another class B, then class A can use all the features (like variables and methods) of class B.

The class which inherits the features of another class is known as subclass. If we want to inherit a class, we use the class name with the name of the class that is to be inherited in the parentheses.

Syntax

```
class sub_classname(Parent_classname):
      'Optional Docstring'
      Class_suite
```

EXAMPLE-INHERITANCE



Example

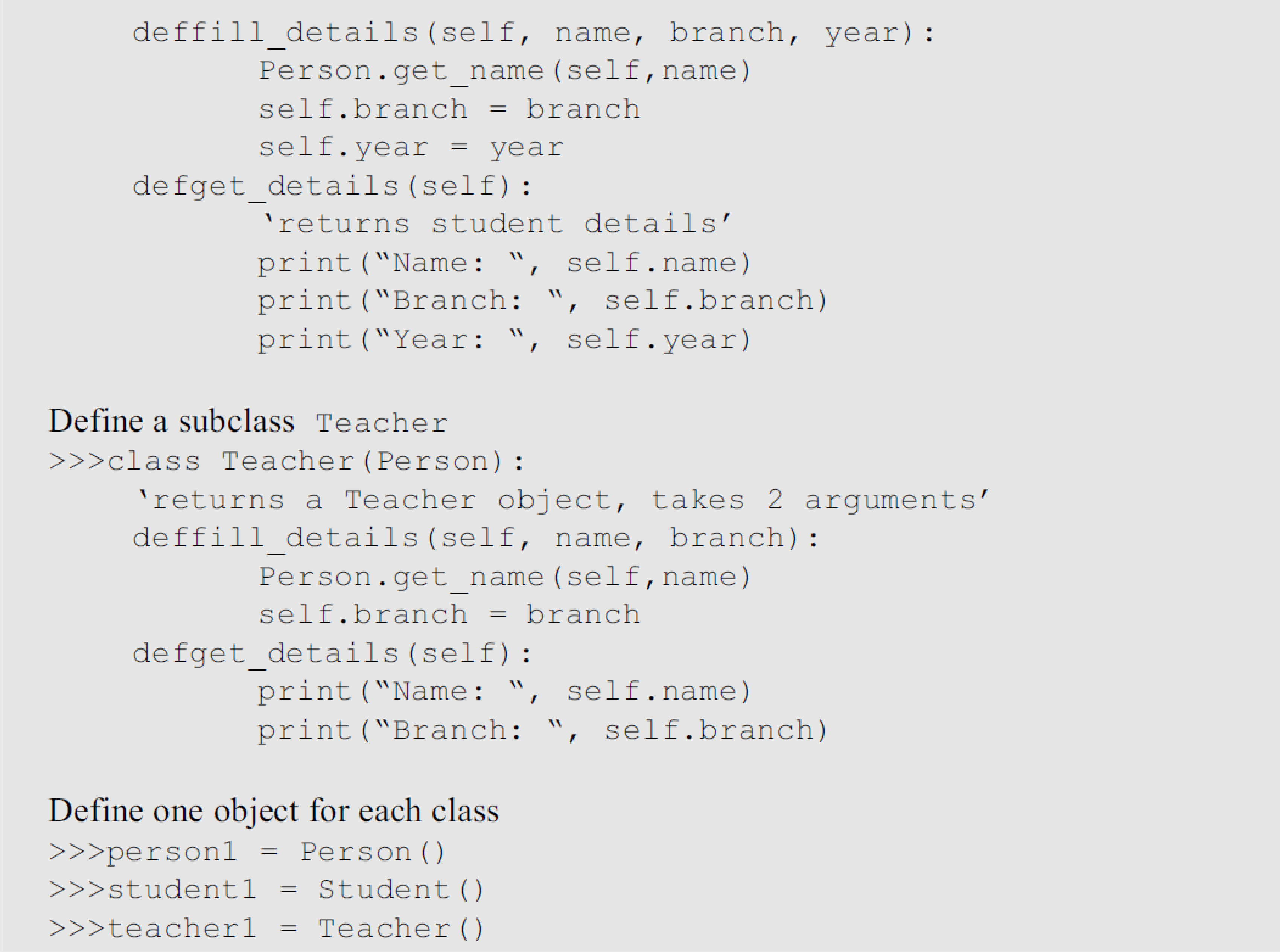
Define a parent class Person

```
>>>class Person(object):
    ‘returns a Person object with given name’
    defget_name(self,name):
          self.name = name
    defget_details(self):
          ‘returns a string containing name of person’
          return self.name
```

Define a subclass Student

```
>>>class Student(Person):
    ‘return a Student object, takes 3 arguments’
```

EXAMPLE-INHERITANCE



```
    deffill_details(self, name, branch, year):
           Person.get_name(self,name)
           self.branch = branch
           self.year = year
    defget_details(self):
           ‘returns student details’
           print(“Name: “, self.name)
           print(“Branch: “, self.branch)
           print(“Year: “, self.year)
```

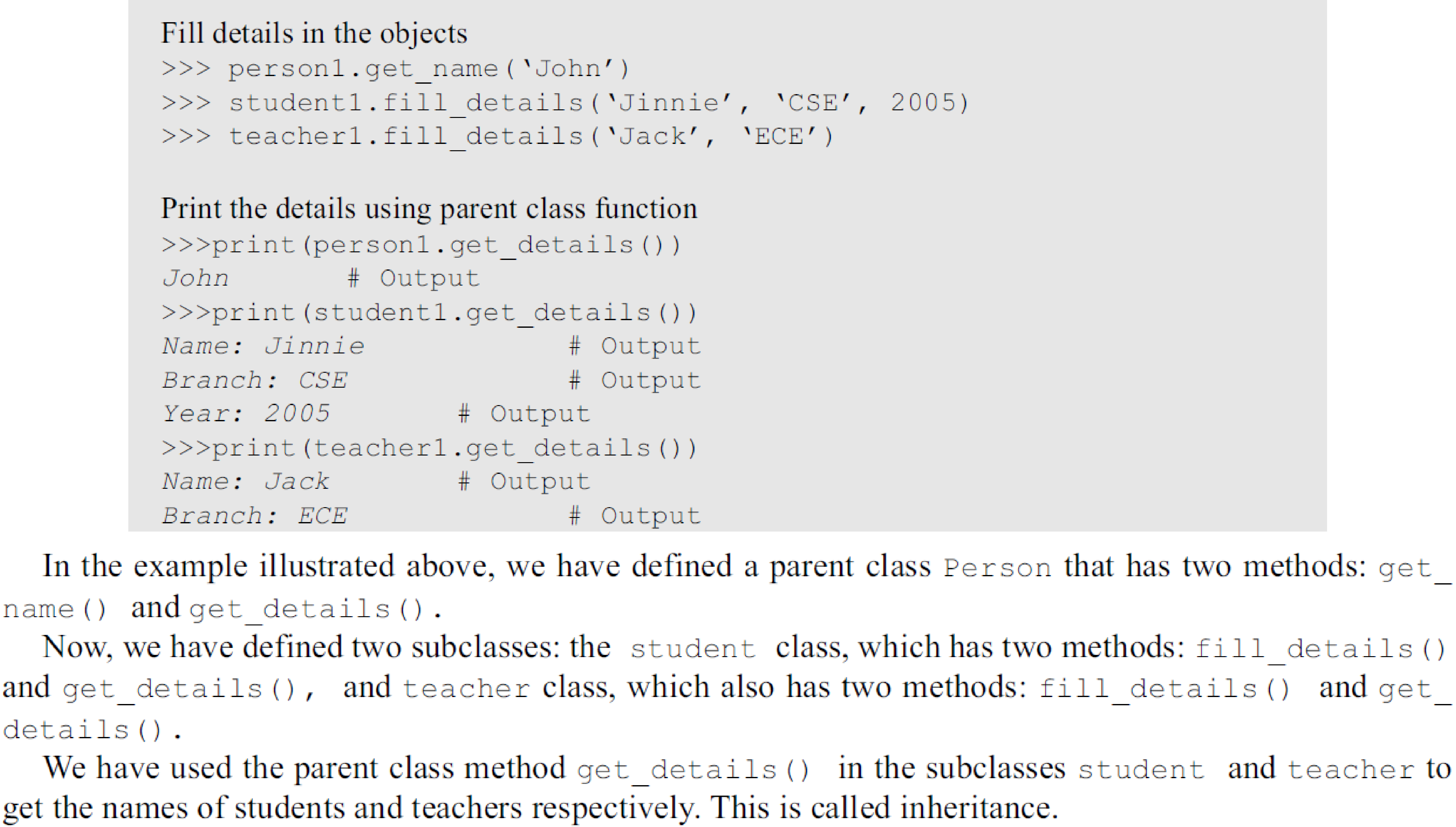
Define a subclass `Teacher`

```
>>>class Teacher(Person):
    ‘returns a Teacher object, takes 2 arguments’
    deffill_details(self, name, branch):
           Person.get_name(self,name)
           self.branch = branch
    defget_details(self):
           print(“Name: “, self.name)
           print(“Branch: “, self.branch)
```

Define one object for each class

```
>>>person1 = Person()
>>>student1 = Student()
>>>teacher1 = Teacher()
```

EXAMPLE-INHERITANCE



Fill details in the objects

```
>>> person1.get_name(‘John’)
>>> student1.fill_details(‘Jinnie’, ‘CSE’, 2005)
>>> teacher1.fill_details(‘Jack’, ‘ECE’)
```

Print the details using parent class function

```
>>>print(person1.get_details())
John       # Output
>>>print(student1.get_details())
Name: Jinnie            # Output
Branch: CSE             # Output
Year: 2005        # Output
>>>print(teacher1.get_details())
Name: Jack        # Output
Branch: ECE             # Output
```

In the example illustrated above, we have defined a parent class `Person` that has two methods: `get_name()` and `get_details()`.

Now, we have defined two subclasses: the `student` class, which has two methods: `fill_details()` and `get_details()`, and `teacher` class, which also has two methods: `fill_details()` and `get_details()`.

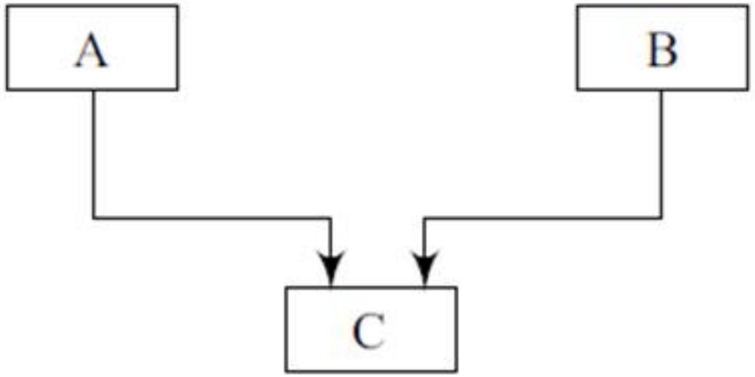
We have used the parent class method `get_details()` in the subclasses `student` and `teacher` to get the names of students and teachers respectively. This is called inheritance.

MULTIPLE INHERITANCE

In multiple inheritance, a subclass is derived from more than one base classes. The subclass inherits the properties of all the base classes. In Figure , subclass C inherits the properties of two base classes A and B.

Let us look at an example:

There are three classes, Water animal (fish, octopus, etc.), Land animal (Tigers, lions, etc.) and Amphibian (frog, crocodiles, etc.).



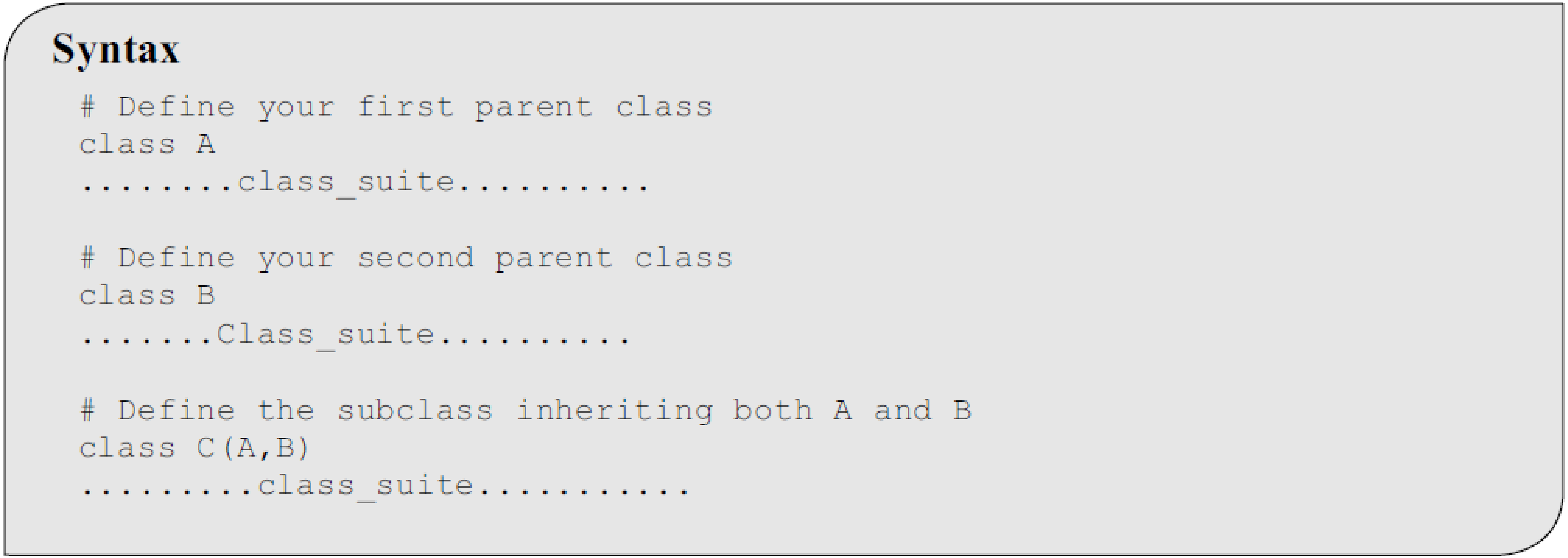
Multiple Inheritance

Figure Multiple Inheritance

Here, Amphibian is the subclass that derives the properties of the base classes, water animal and land animal. Therefore, its animals (objects) frog and crocodile live both on land and water.

We can also define multiple inheritance in Python. When a class inherits the features of more than one class this is known as multiple inheritance. It is defined in the same way as inheritance.

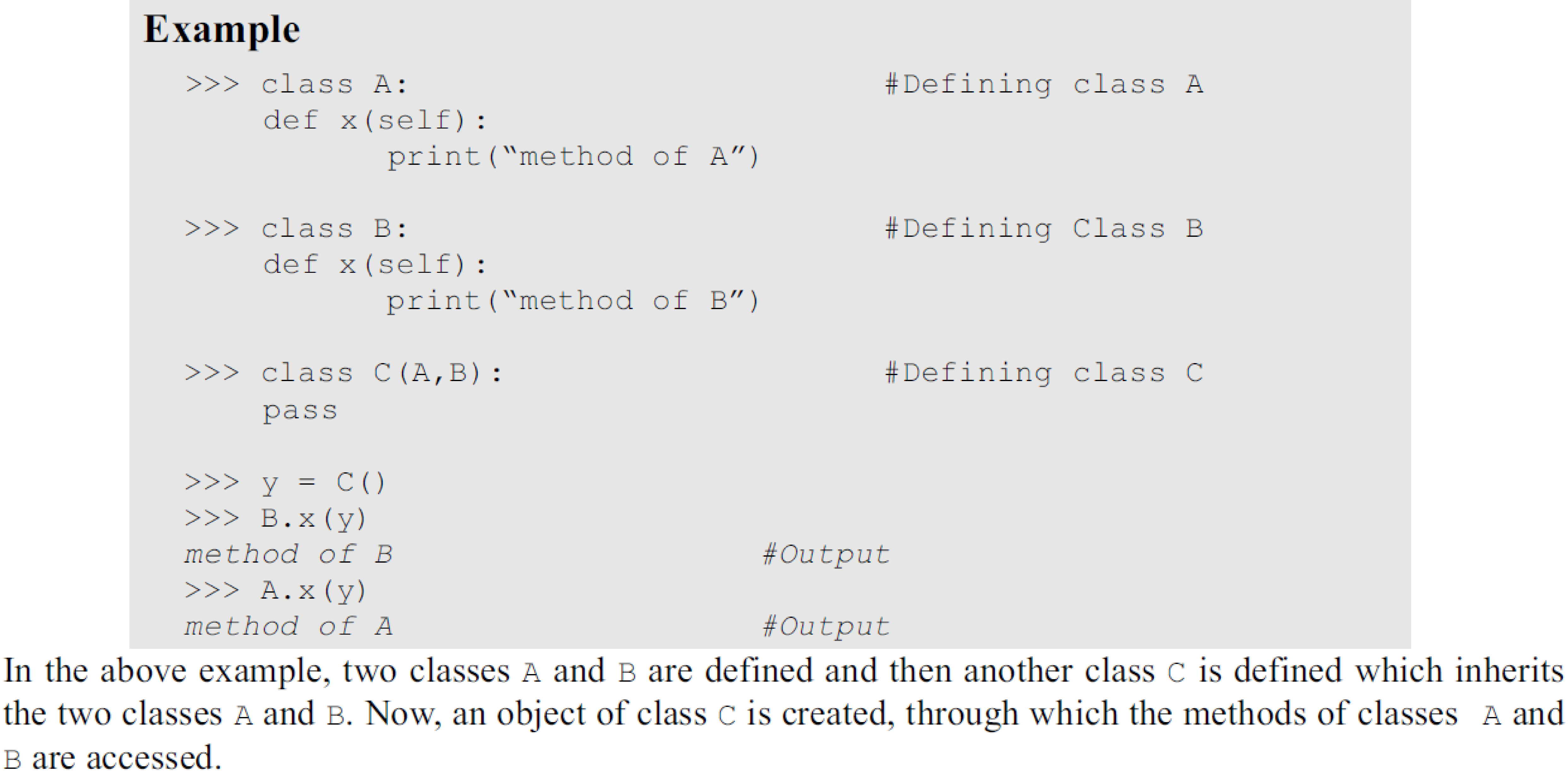
MULTIPLE INHERITANCE (CONTD..)



Syntax

```
# Define your first parent class
class A
........class_suite..........

# Define your second parent class
class B
.......Class_suite..........

# Define the subclass inheriting both A and B
class C(A,B)
.........class_suite...........
```

MULTIPLE INHERITANCE (CONTD..)



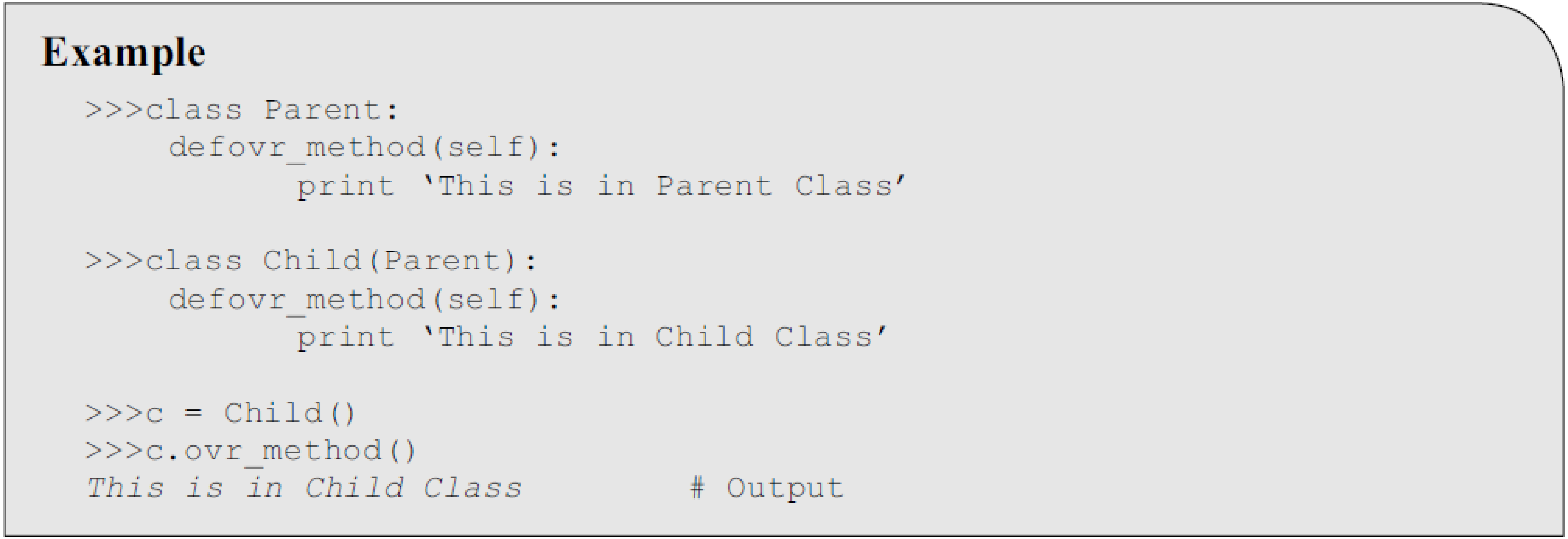
Example

```
>>> class A:                        #Defining class A
    def x(self):
         print(“method of A”)

>>> class B:                        #Defining Class B
    def x(self):
         print(“method of B”)

>>> class C(A,B):                   #Defining class C
    pass

>>> y = C()
>>> B.x(y)
method of B                    #Output
>>> A.x(y)
method of A                    #Output
```

In the above example, two classes `A` and `B` are defined and then another class `C` is defined which inherits the two classes `A` and `B`. Now, an object of class `C` is created, through which the methods of classes `A` and `B` are accessed.

METHOD OVERRIDING

Method overriding is allowed in Python. Method overriding means that the method of parent class can be used in the subclass with different or special functionality.



Example

```
>>>class Parent:
     defovr_method(self):
          print ‘This is in Parent Class’

>>>class Child(Parent):
     defovr_method(self):
          print ‘This is in Child Class’

>>>c = Child()
>>>c.ovr_method()
This is in Child Class             # Output
```

DATA ENCAPSULATION

In Python Programming Language, encapsulation is a process to restrict the access of data members. This means that the internal details of an object may not be visible from outside of the object definition. But Python provides some methods which assist in accessing these sorts of data.

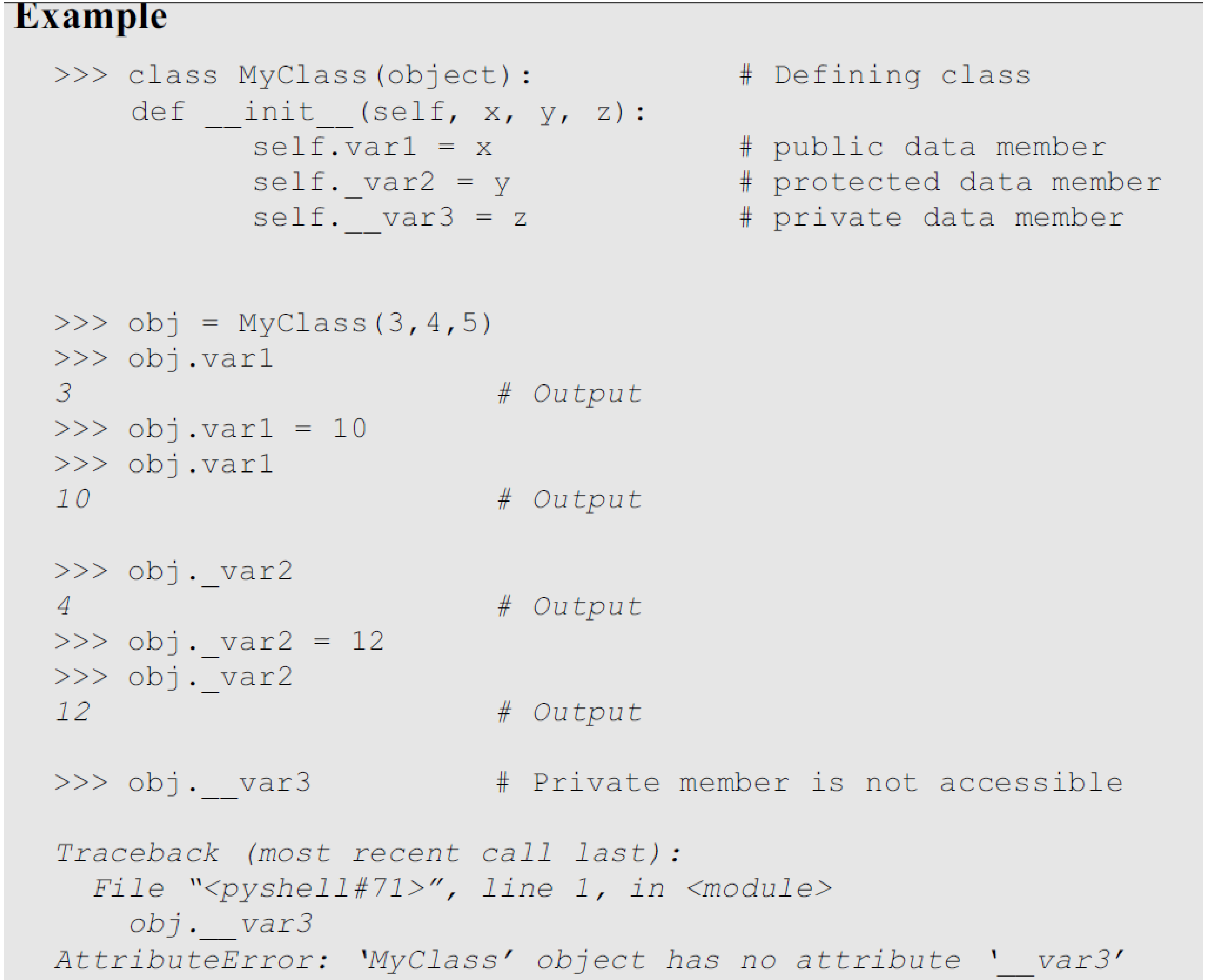
The members in a class can be assigned in three ways i.e., public, protected and private. If the name of a member is preceded by single underscore, it is assigned as a protected member, whereas if the name of a member is preceded by double underscore, it is assigned as a private member and if the name is not preceded by anything then it is a public member.

Let us summarise this concept in the given table below:

Name	Notation	Behaviour
varname	Public	Can be accessed from anywhere
_varname	Protected	They are like the public members but they cannot be directly accessed from outside
__varname	Private	They cannot be seen and accessed from outside the class

Let us understand this concept with the help of an example:

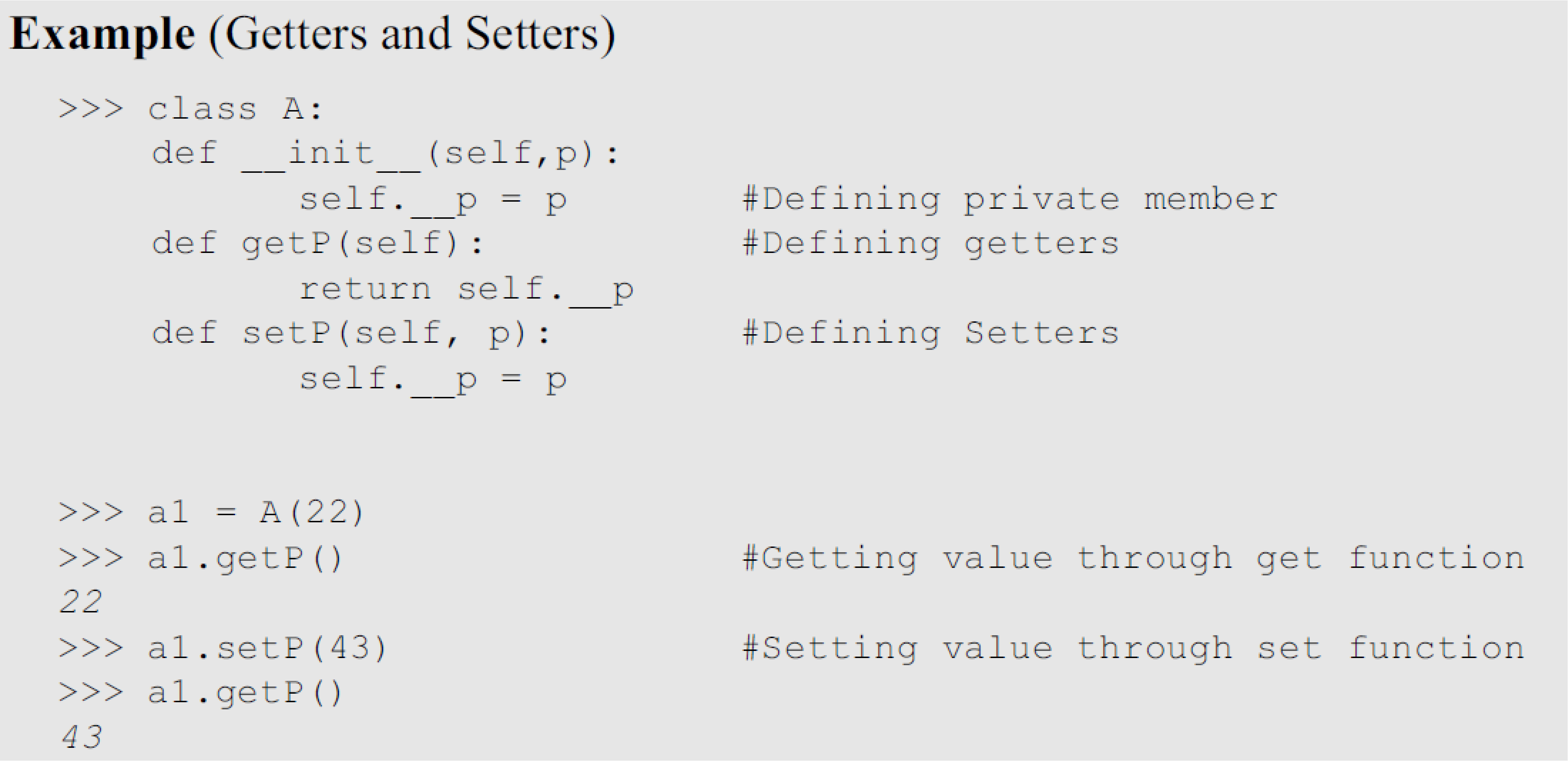
DATA ENCAPSULATION



Example

```
>>> class MyClass(object):           # Defining class
    def __init__(self, x, y, z):
           self.var1 = x             # public data member
           self._var2 = y            # protected data member
           self.__var3 = z           # private data member


>>> obj = MyClass(3,4,5)
>>> obj.var1
3                       # Output
>>> obj.var1 = 10
>>> obj.var1
10                      # Output

>>> obj._var2
4                       # Output
>>> obj._var2 = 12
>>> obj._var2
12                      # Output

>>> obj.__var3          # Private member is not accessible

Traceback (most recent call last):
  File “<pyshell#71>”, line 1, in <module>
    obj.__var3
AttributeError: ‘MyClass’ object has no attribute ‘__var3’
```

DATA ENCAPSULATION



Example (Getters and Setters)

```
>>> class A:
    def __init__(self,p):
           self.__p = p             #Defining private member
    def getP(self):                 #Defining getters
           return self.__p
    def setP(self, p):              #Defining Setters
           self.__p = p



>>> a1 = A(22)
>>> a1.getP()                       #Getting value through get function
22
>>> a1.setP(43)                     #Setting value through set function
>>> a1.getP()
43
```

DATA HIDING

In Python programming, there might be some cases when you intend to hide the attributes of objects outside the class definition. To accomplish this, use double score (__) before the name of the attributes and these attributes will not be visible directly outside the class definition. Let us understand the Python data hiding by a simple example given below:

Example

```
>>> class MyClass:                   # defining class
    __a = 0;
    def sum(self, increment):
           self.__a += increment
           print self.__a


>>> b = MyClass()                    # creating instance of class
>>> b.sum(2)
2
>>> b.sum(5)
7
>>> print b.__a

Traceback (most recent call last):
  File “<pyshell#24>”, line 1, in <module>
    print b.__a
AttributeError: MyClass instance has no attribute ‘__a’
```

DATA HIDING

As seen in the above example that the variable __a is not accessible as we tried to access it; the Python interpreter generates an error immediately. In such a case, the Python secures the members by internally changing the names to incorporate the name of the class. If you intend to access these attributes then the syntax for accessing the variable is:

```
objectName._className__attributeName
```

In the above code, if we use the aforementioned syntax to access the attributes, then the following changes are seen in the output:

```
>>> class MyClass:                          # Defining class
    __a = 0;
    def sum(self, increment):
           self.__a += increment
           print self.__a


>>> b = MyClass()                           # creating instance of class
>>> b.sum(2)
2
>>> b.sum(5)
7
>>> print b._MyClass__a                     # Accessing the hidden variable
7
```

Thank You