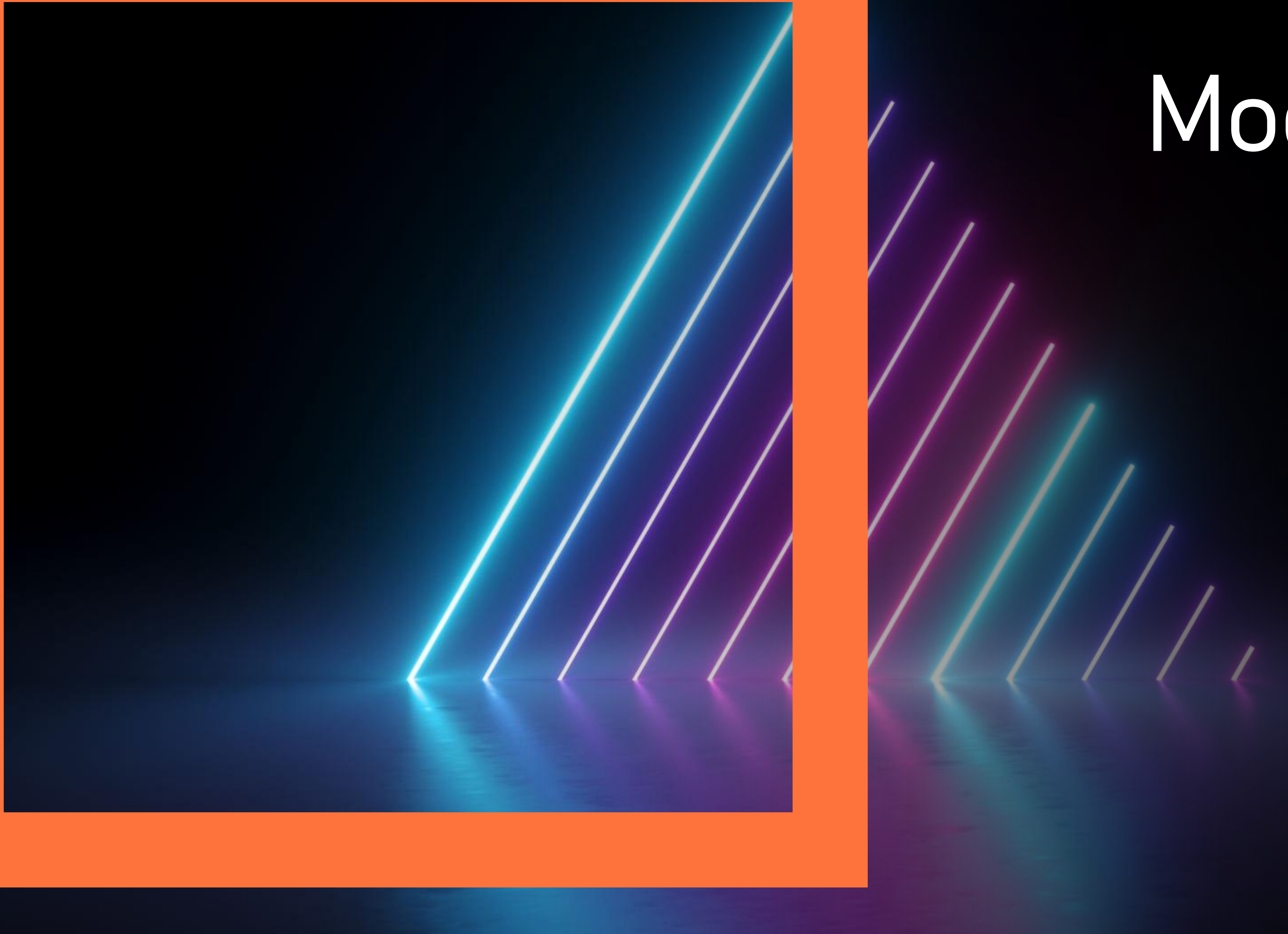
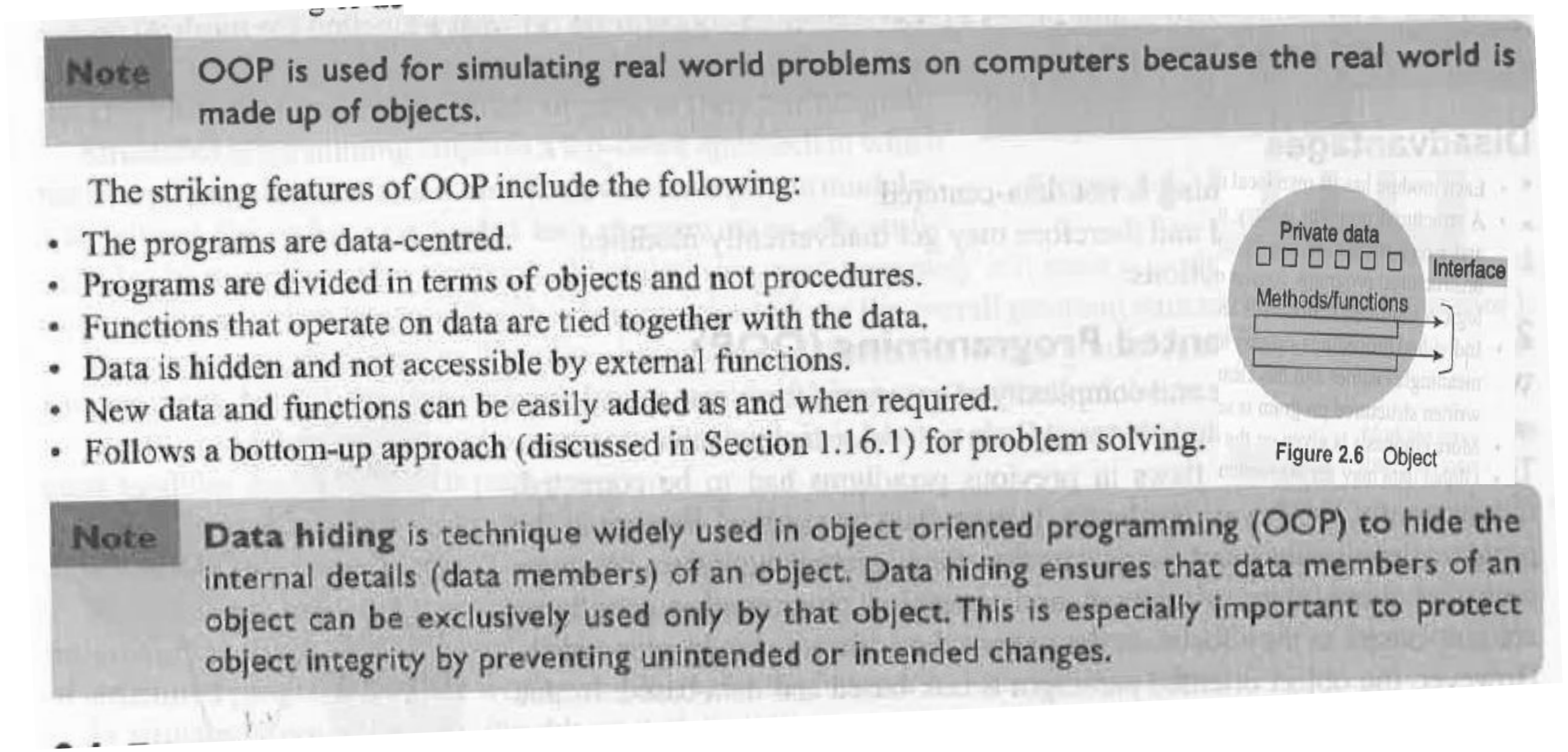
# Module-IV





**Note** OOP is used for simulating real world problems on computers because the real world is made up of objects.

The striking features of OOP include the following:

- The programs are data-centred.
- Programs are divided in terms of objects and not procedures.
- Functions that operate on data are tied together with the data.
- Data is hidden and not accessible by external functions.
- New data and functions can be easily added as and when required.
- Follows a bottom-up approach (discussed in Section 1.16.1) for problem solving.

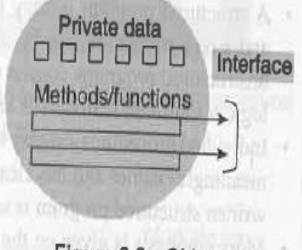
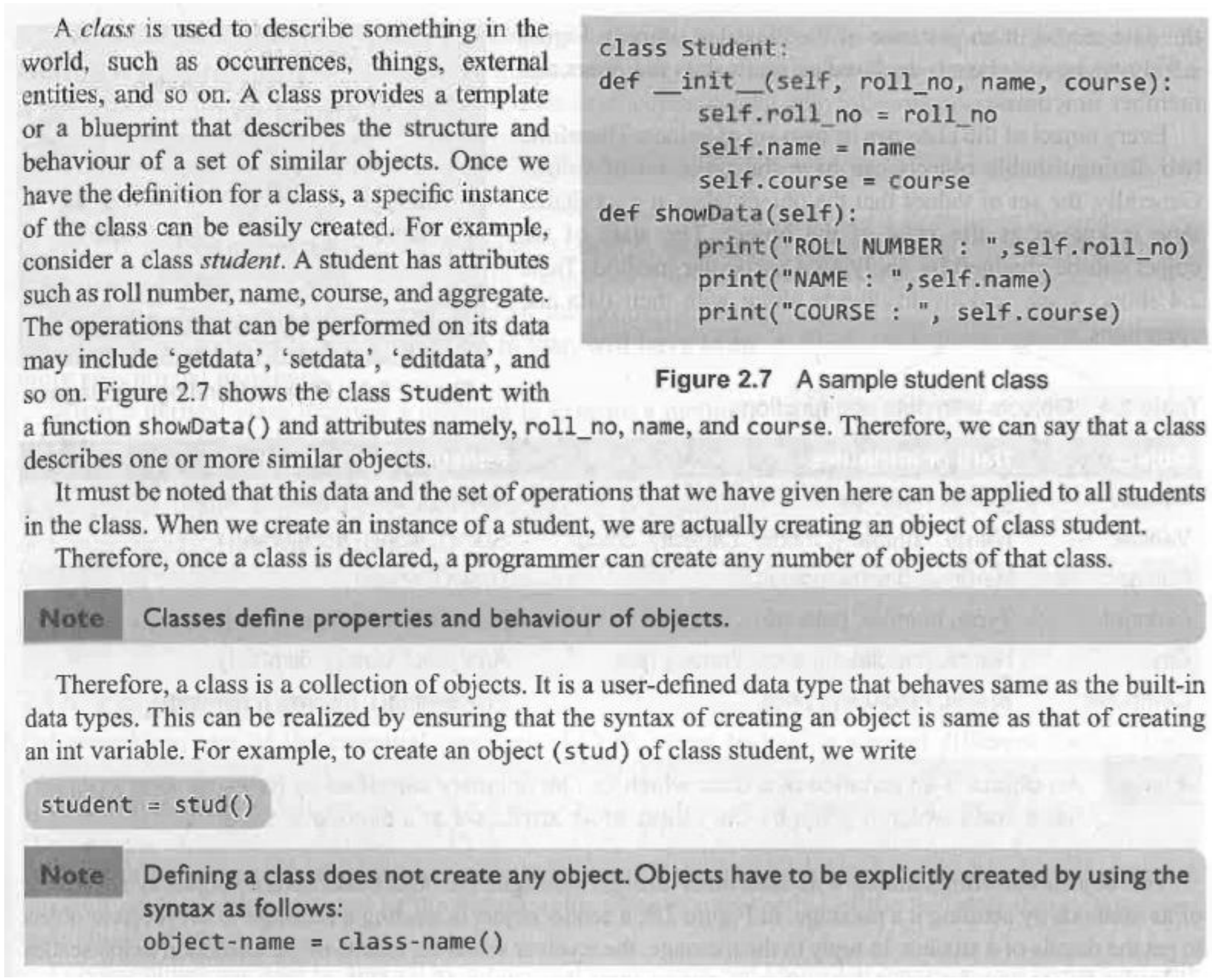


Figure 2.6 Object

**Note** **Data hiding** is technique widely used in object oriented programming (OOP) to hide the internal details (data members) of an object. Data hiding ensures that data members of an object can be exclusively used only by that object. This is especially important to protect object integrity by preventing unintended or intended changes.



A *class* is used to describe something in the world, such as occurrences, things, external entities, and so on. A class provides a template or a blueprint that describes the structure and behaviour of a set of similar objects. Once we have the definition for a class, a specific instance of the class can be easily created. For example, consider a class *student*. A student has attributes such as roll number, name, course, and aggregate. The operations that can be performed on its data may include 'getdata', 'setdata', 'editdata', and so on. Figure 2.7 shows the class `Student` with a function `showData()` and attributes namely, `roll_no`, `name`, and `course`. Therefore, we can say that a class describes one or more similar objects.

```
class Student:
def __init__(self, roll_no, name, course):
        self.roll_no = roll_no
        self.name = name
        self.course = course
def showData(self):
        print("ROLL NUMBER : ",self.roll_no)
        print("NAME : ",self.name)
        print("COURSE : ", self.course)
```

**Figure 2.7** A sample student class

It must be noted that this data and the set of operations that we have given here can be applied to all students in the class. When we create an instance of a student, we are actually creating an object of class student.

Therefore, once a class is declared, a programmer can create any number of objects of that class.

**Note** Classes define properties and behaviour of objects.

Therefore, a class is a collection of objects. It is a user-defined data type that behaves same as the built-in data types. This can be realized by ensuring that the syntax of creating an object is same as that of creating an int variable. For example, to create an object `(stud)` of class student, we write

```
student = stud()
```

**Note** Defining a class does not create any object. Objects have to be explicitly created by using the syntax as follows:

```
object-name = class-name()
```

## 9.2 CLASSES AND OBJECTS

Classes and objects are the two main aspects of object oriented programming. In fact, a class is the basic building block in Python. A **class** creates a new type and object is an instance (or variable) of the class. Classes provides a blueprint or a template using which objects are created. In fact, *in Python, everything is an object or an instance of some class.* For example, all integer variables that we define in our program are actually instances of class int. Similarly, all string variables are objects of class string. Recall that we had used string methods using the variable name followed by the dot operator and the method name. We have already studied that we can find out the type of any object using the `type()` function.

**Note** The Python Standard Library is based on the concept of classes and objects.

### 9.2.1 Defining Classes

Python has a very simple syntax of defining a class. This syntax can be given as,

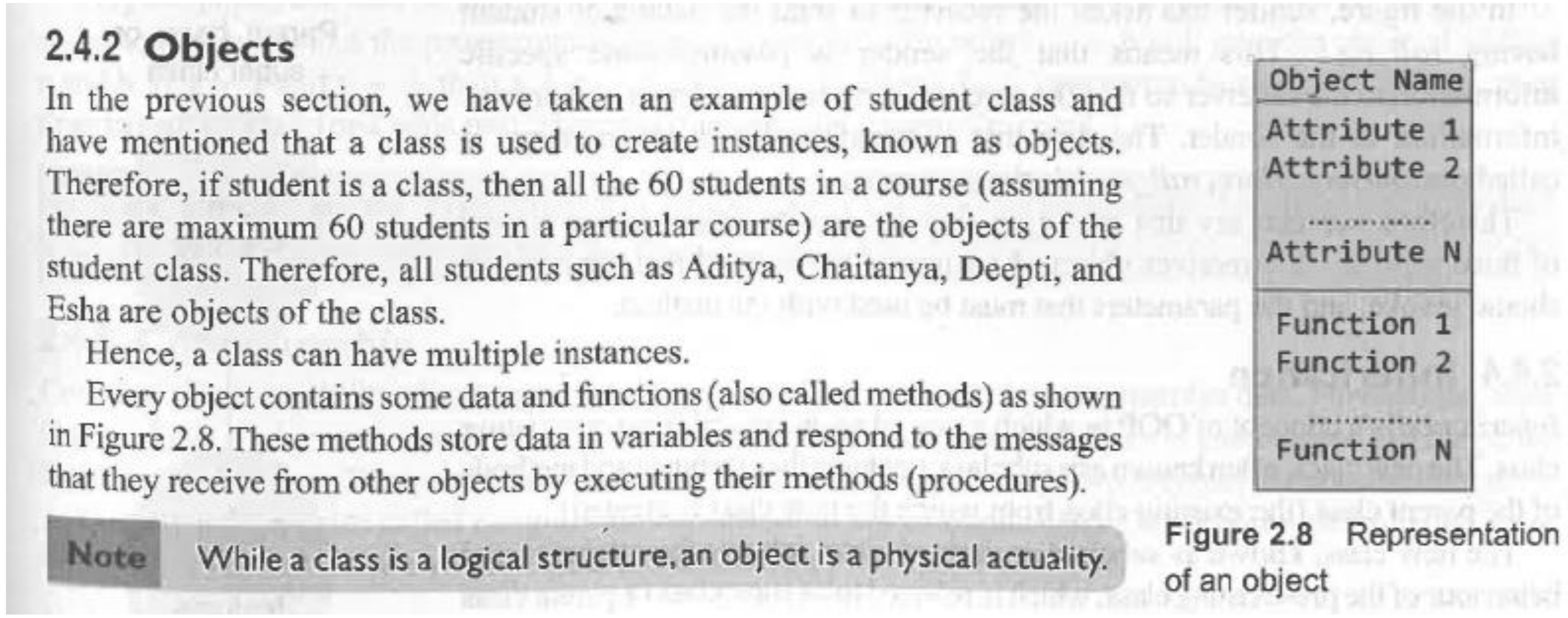
```
class class_name:
    <statement-1>
    <statement-2>
    .
    .
    .
    <statement-N>
```

**Programming Tip:** A class can be defined in a function or with an `if` statement.

From the syntax, we see that class definition is quite similar to function definition. It starts with a keyword class followed by the `class_name` and a colon (`:`). The statement in the definition can be any of these—sequential instructions, decision control statements, loop statements, and can even include function definitions. Variables defined in a class are called *class variables* and functions defined inside a class are called *class methods*. Class variables and class methods are together known as *class members*. The class members can be accessed through class objects. Class methods have access to all the data contained in the instance of the object.

Class definitions can appear anywhere in a program, but they are usually written near the beginning of the program, after the import statements. Note that when a class definition is entered, a new namespace is created, and used as the local scope. Therefore, all assignments to local variables go into this new namespace.

**Note** A class creates a new local namespace where all its attributes (data and functions) are defined.



## 2.4.2 Objects

In the previous section, we have taken an example of student class and have mentioned that a class is used to create instances, known as objects. Therefore, if student is a class, then all the 60 students in a course (assuming there are maximum 60 students in a particular course) are the objects of the student class. Therefore, all students such as Aditya, Chaitanya, Deepti, and Esha are objects of the class.

Hence, a class can have multiple instances.

Every object contains some data and functions (also called methods) as shown in Figure 2.8. These methods store data in variables and respond to the messages that they receive from other objects by executing their methods (procedures).

**Note** While a class is a logical structure, an object is a physical actuality.

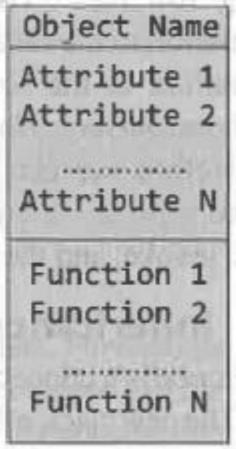
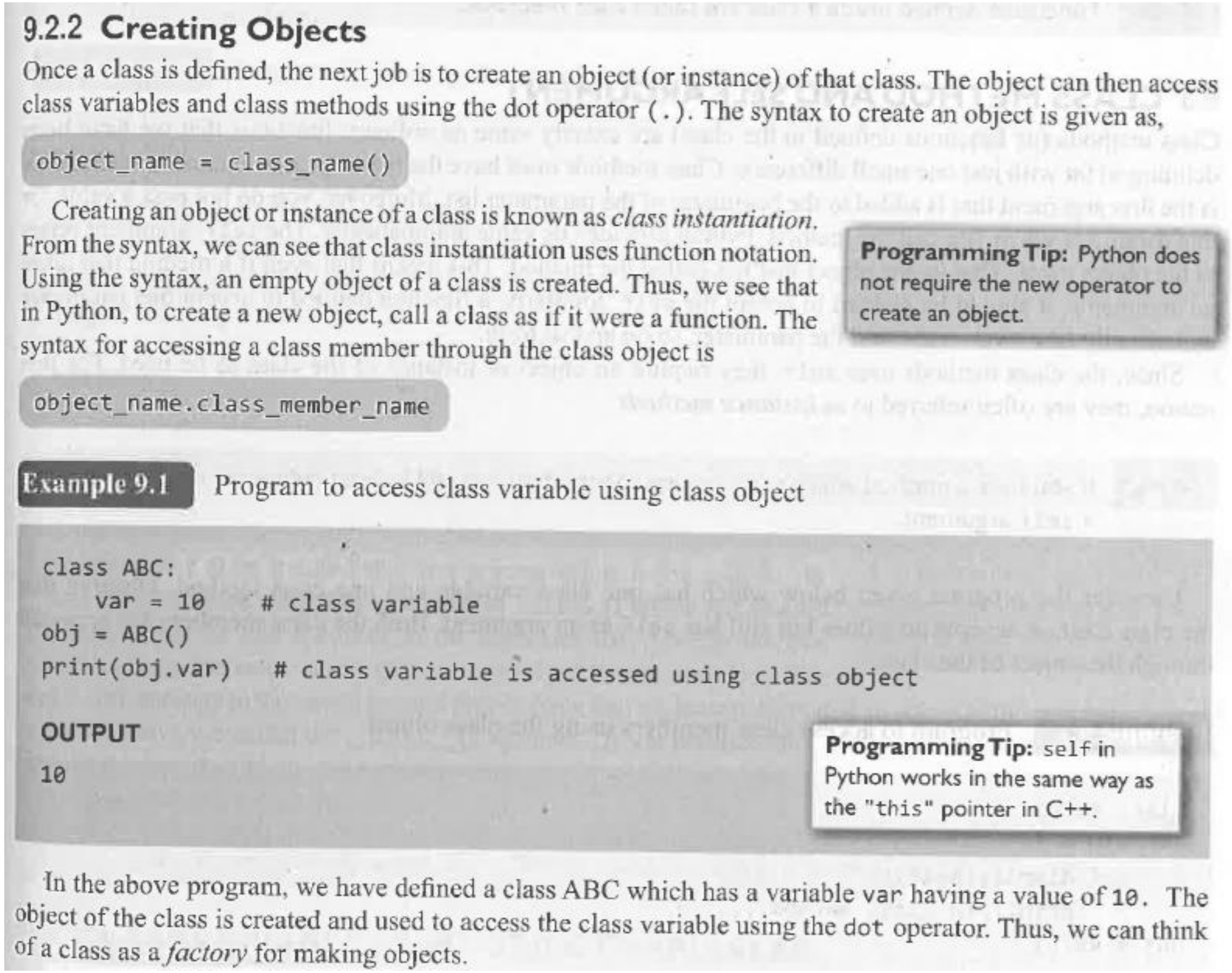


Figure 2.8 Representation of an object



## 9.2.2 Creating Objects

Once a class is defined, the next job is to create an object (or instance) of that class. The object can then access class variables and class methods using the dot operator (`.`). The syntax to create an object is given as,

```
object_name = class_name()
```

Creating an object or instance of a class is known as *class instantiation*. From the syntax, we can see that class instantiation uses function notation. Using the syntax, an empty object of a class is created. Thus, we see that in Python, to create a new object, call a class as if it were a function. The syntax for accessing a class member through the class object is

**Programming Tip:** Python does not require the new operator to create an object.

```
object_name.class_member_name
```

**Example 9.1** Program to access class variable using class object

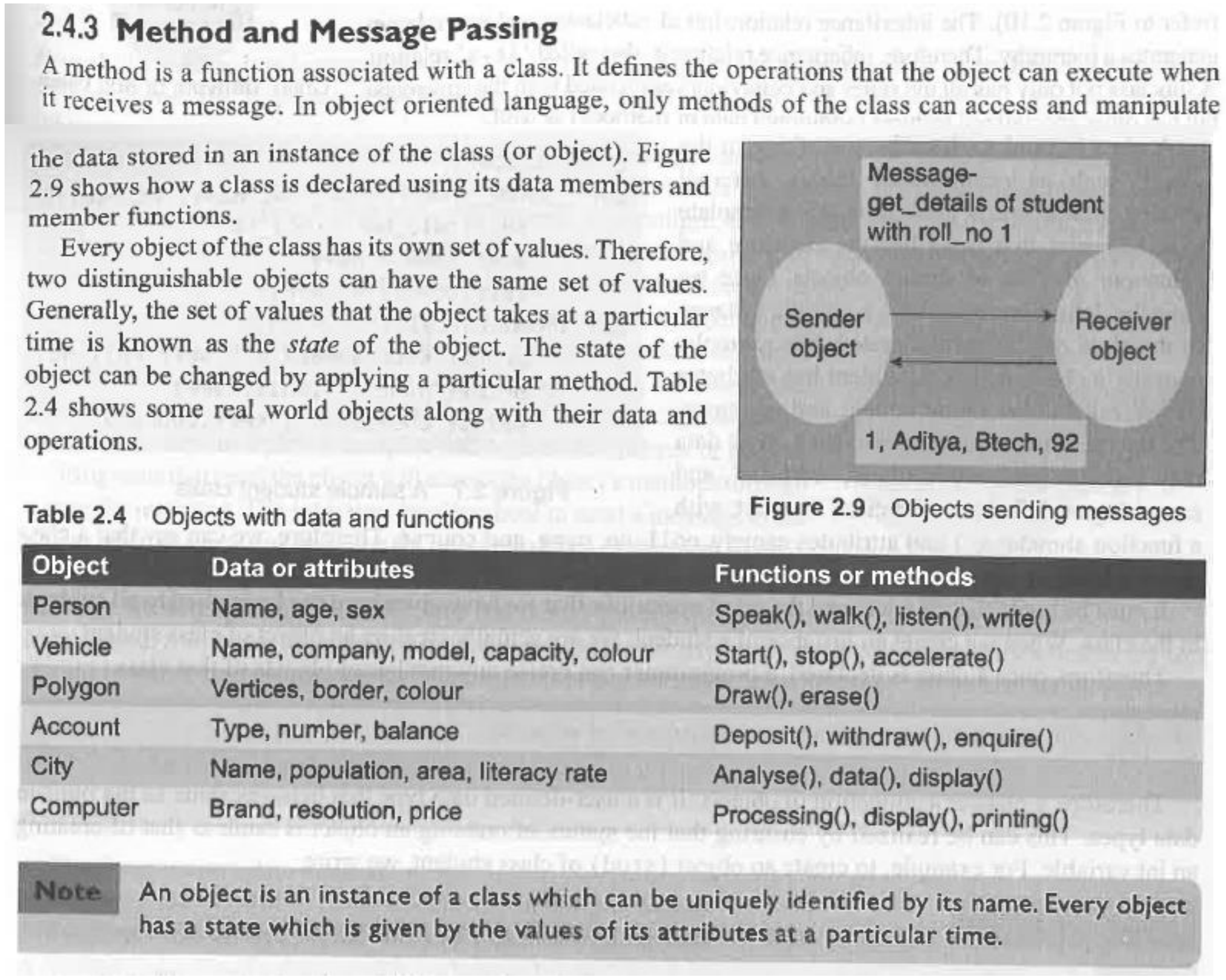
```
class ABC:
    var = 10     # class variable
obj = ABC()
print(obj.var)    # class variable is accessed using class object
```

**OUTPUT**

```
10
```

**Programming Tip:** `self` in Python works in the same way as the "`this`" pointer in C++.

In the above program, we have defined a class ABC which has a variable `var` having a value of `10`. The object of the class is created and used to access the class variable using the dot operator. Thus, we can think of a class as a *factory* for making objects.



## 2.4.3 Method and Message Passing

A method is a function associated with a class. It defines the operations that the object can execute when it receives a message. In object oriented language, only methods of the class can access and manipulate the data stored in an instance of the class (or object). Figure 2.9 shows how a class is declared using its data members and member functions.

Every object of the class has its own set of values. Therefore, two distinguishable objects can have the same set of values. Generally, the set of values that the object takes at a particular time is known as the *state* of the object. The state of the object can be changed by applying a particular method. Table 2.4 shows some real world objects along with their data and operations.

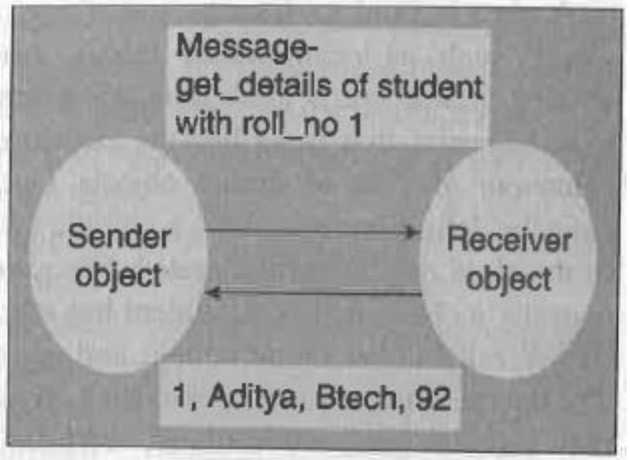
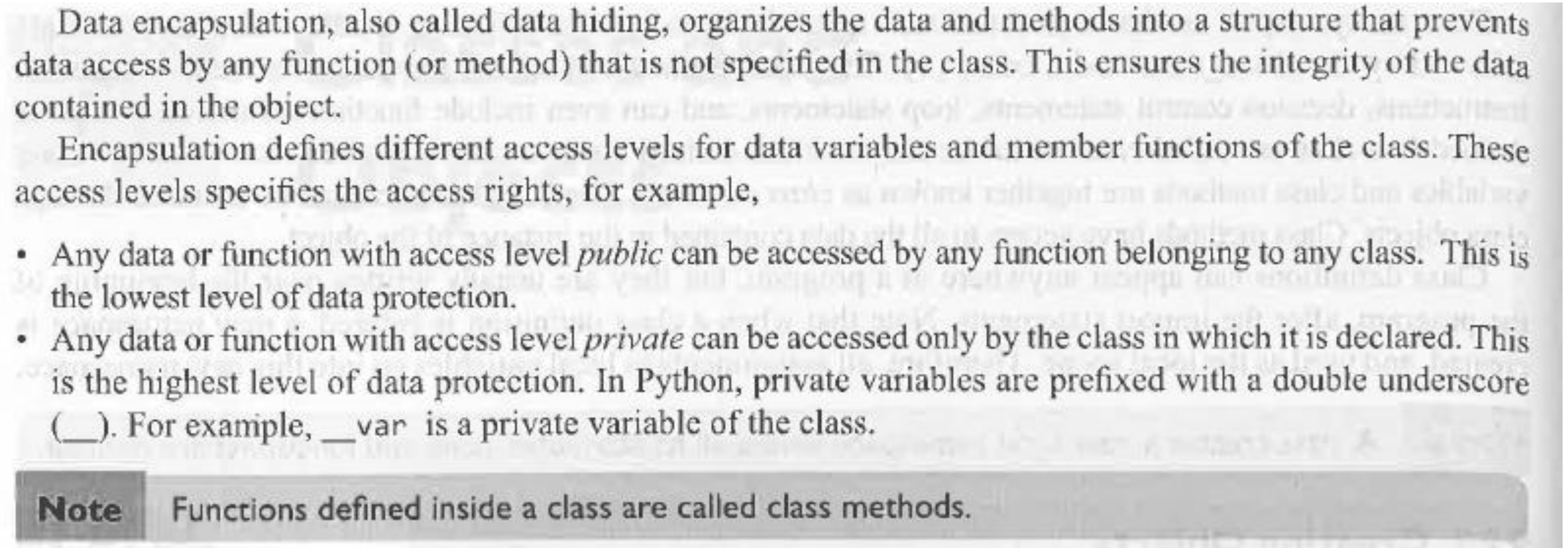


**Figure 2.9** Objects sending messages

**Table 2.4** Objects with data and functions

| Object | Data or attributes | Functions or methods |
|---|---|---|
| Person | Name, age, sex | Speak(), walk(), listen(), write() |
| Vehicle | Name, company, model, capacity, colour | Start(), stop(), accelerate() |
| Polygon | Vertices, border, colour | Draw(), erase() |
| Account | Type, number, balance | Deposit(), withdraw(), enquire() |
| City | Name, population, area, literacy rate | Analyse(), data(), display() |
| Computer | Brand, resolution, price | Processing(), display(), printing() |

**Note** An object is an instance of a class which can be uniquely identified by its name. Every object has a state which is given by the values of its attributes at a particular time.

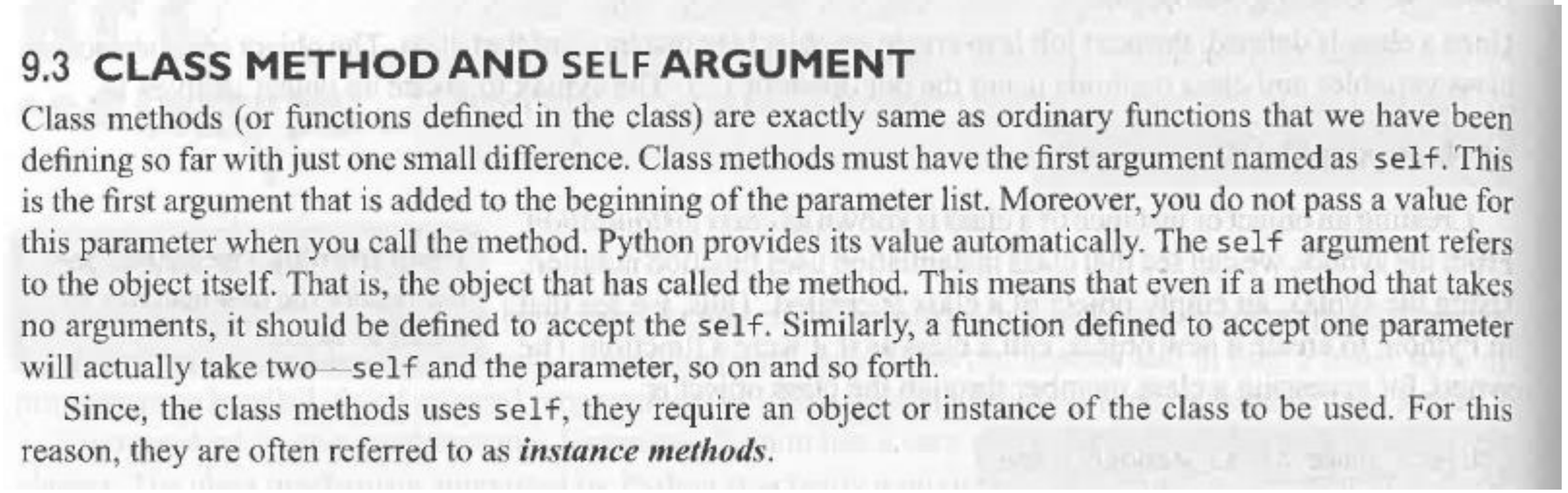


Data encapsulation, also called data hiding, organizes the data and methods into a structure that prevents data access by any function (or method) that is not specified in the class. This ensures the integrity of the data contained in the object.

Encapsulation defines different access levels for data variables and member functions of the class. These access levels specifies the access rights, for example,

- Any data or function with access level *public* can be accessed by any function belonging to any class. This is the lowest level of data protection.
- Any data or function with access level *private* can be accessed only by the class in which it is declared. This is the highest level of data protection. In Python, private variables are prefixed with a double underscore (__). For example, `__var` is a private variable of the class.

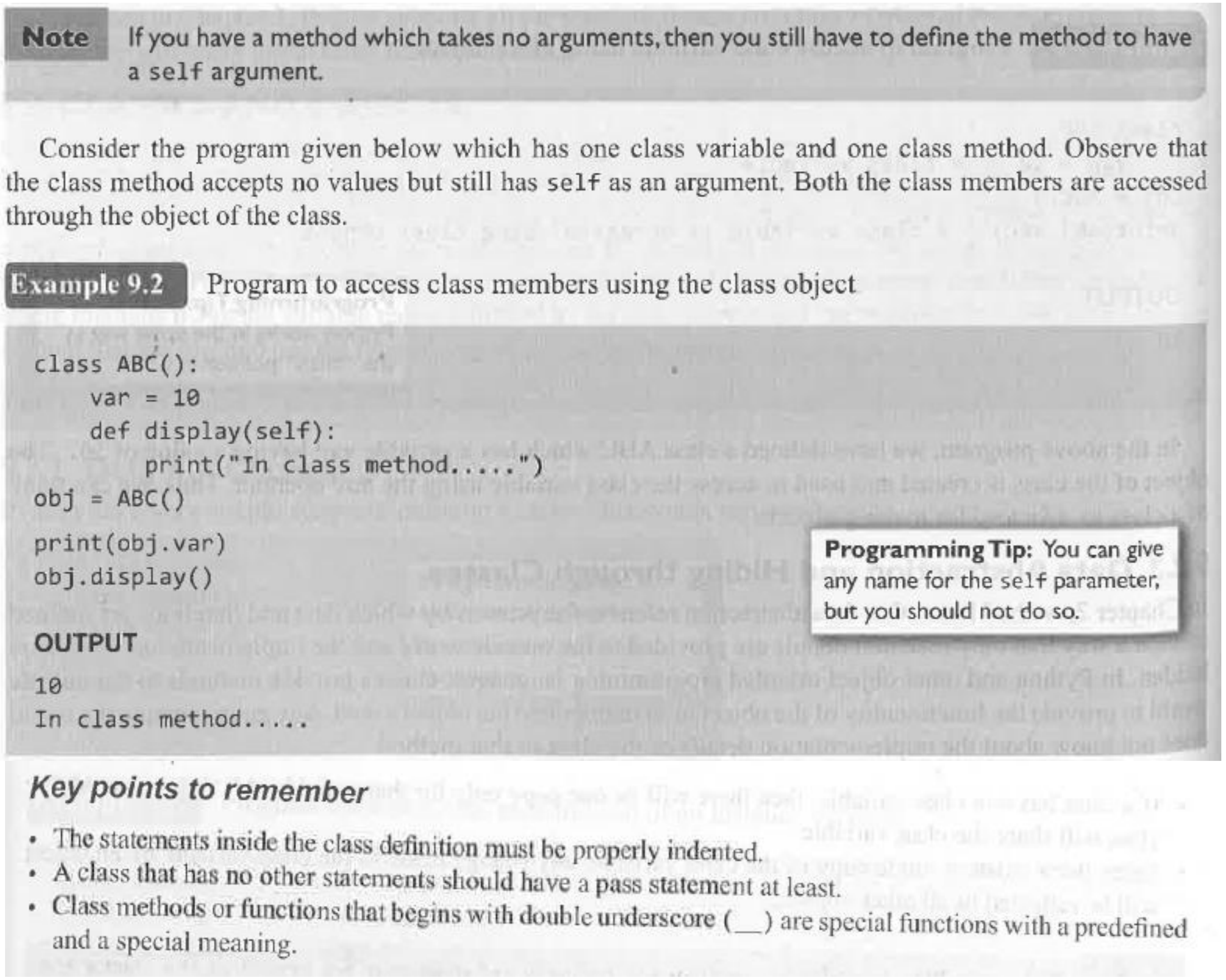
**Note** Functions defined inside a class are called class methods.



## 9.3 CLASS METHOD AND SELF ARGUMENT

Class methods (or functions defined in the class) are exactly same as ordinary functions that we have been defining so far with just one small difference. Class methods must have the first argument named as `self`. This is the first argument that is added to the beginning of the parameter list. Moreover, you do not pass a value for this parameter when you call the method. Python provides its value automatically. The `self` argument refers to the object itself. That is, the object that has called the method. This means that even if a method that takes no arguments, it should be defined to accept the `self`. Similarly, a function defined to accept one parameter will actually take two—`self` and the parameter, so on and so forth.

Since, the class methods uses `self`, they require an object or instance of the class to be used. For this reason, they are often referred to as ***instance methods***.



**Note** If you have a method which takes no arguments, then you still have to define the method to have a `self` argument.

Consider the program given below which has one class variable and one class method. Observe that the class method accepts no values but still has `self` as an argument. Both the class members are accessed through the object of the class.

**Example 9.2** Program to access class members using the class object

```
class ABC():
    var = 10
    def display(self):
        print("In class method.....")
obj = ABC()
print(obj.var)
obj.display()
```

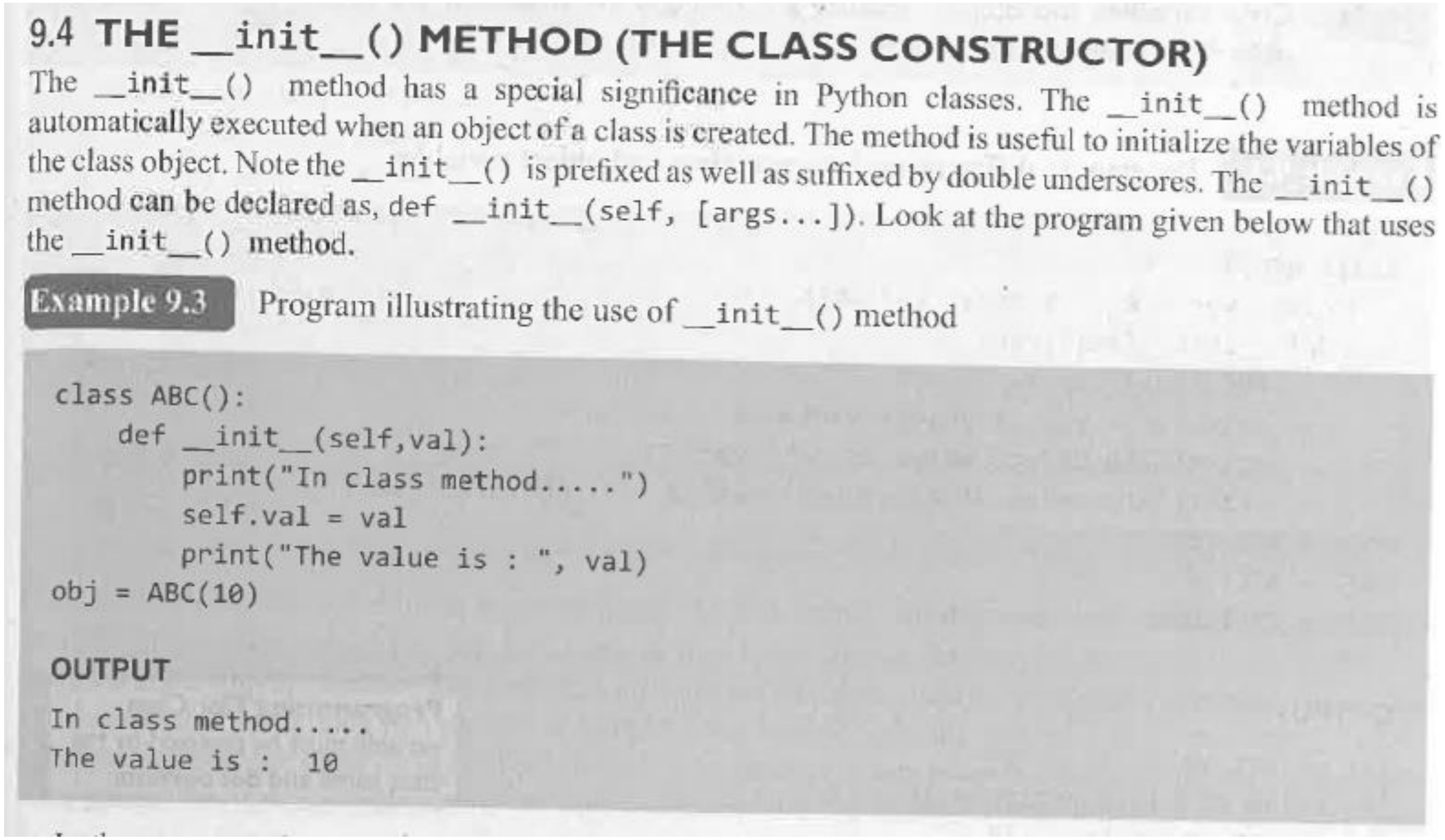
**OUTPUT**

```
10
In class method.....
```

**Programming Tip:** You can give any name for the `self` parameter, but you should not do so.

***Key points to remember***

- The statements inside the class definition must be properly indented.
- A class that has no other statements should have a pass statement at least.
- Class methods or functions that begins with double underscore (__) are special functions with a predefined and a special meaning.



## 9.4 THE __init__() METHOD (THE CLASS CONSTRUCTOR)

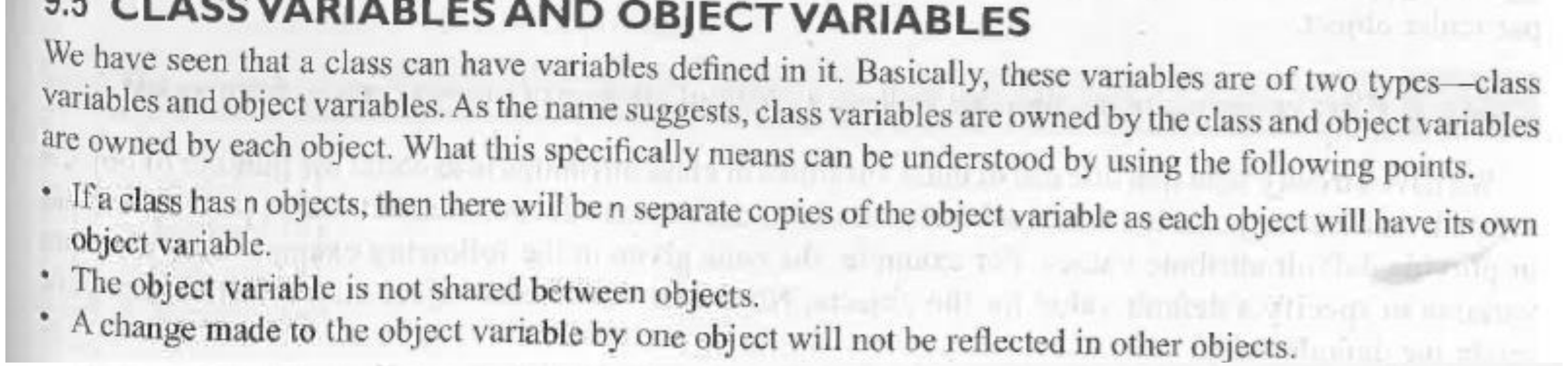
The __init__() method has a special significance in Python classes. The __init__() method is automatically executed when an object of a class is created. The method is useful to initialize the variables of the class object. Note the __init__() is prefixed as well as suffixed by double underscores. The __init__() method can be declared as, def __init__(self, [args...]). Look at the program given below that uses the __init__() method.

**Example 9.3** Program illustrating the use of __init__() method

```
class ABC():
    def __init__(self,val):
        print("In class method.....")
        self.val = val
        print("The value is : ", val)
obj = ABC(10)
```

**OUTPUT**

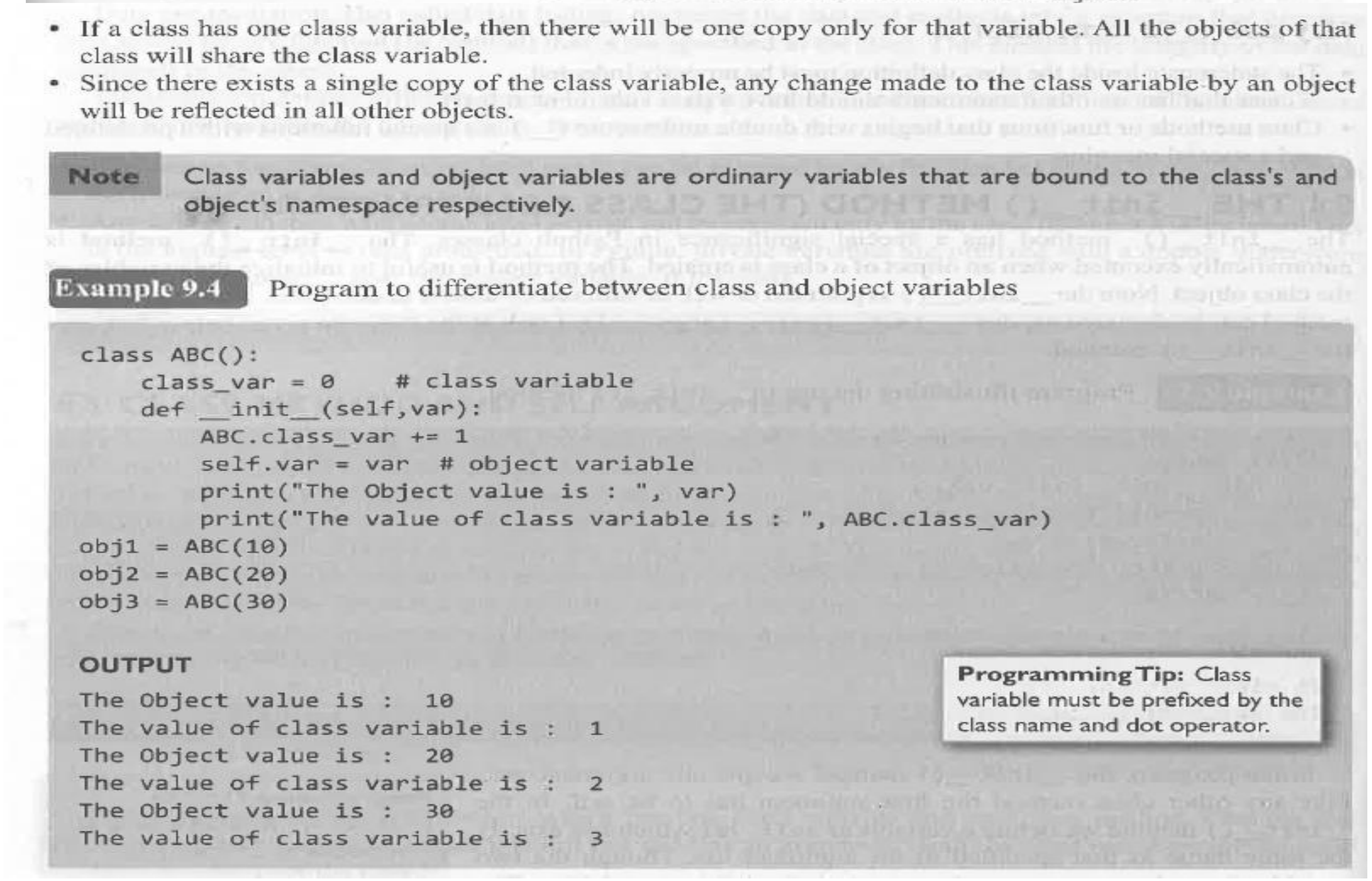
```
In class method.....
The value is :  10
```



## 9.5 CLASS VARIABLES AND OBJECT VARIABLES

We have seen that a class can have variables defined in it. Basically, these variables are of two types—class variables and object variables. As the name suggests, class variables are owned by the class and object variables are owned by each object. What this specifically means can be understood by using the following points.

- If a class has n objects, then there will be *n* separate copies of the object variable as each object will have its own object variable.
- The object variable is not shared between objects.
- A change made to the object variable by one object will not be reflected in other objects.



- If a class has one class variable, then there will be one copy only for that variable. All the objects of that class will share the class variable.
- Since there exists a single copy of the class variable, any change made to the class variable by an object will be reflected in all other objects.

**Note** Class variables and object variables are ordinary variables that are bound to the class's and object's namespace respectively.

**Example 9.4** Program to differentiate between class and object variables

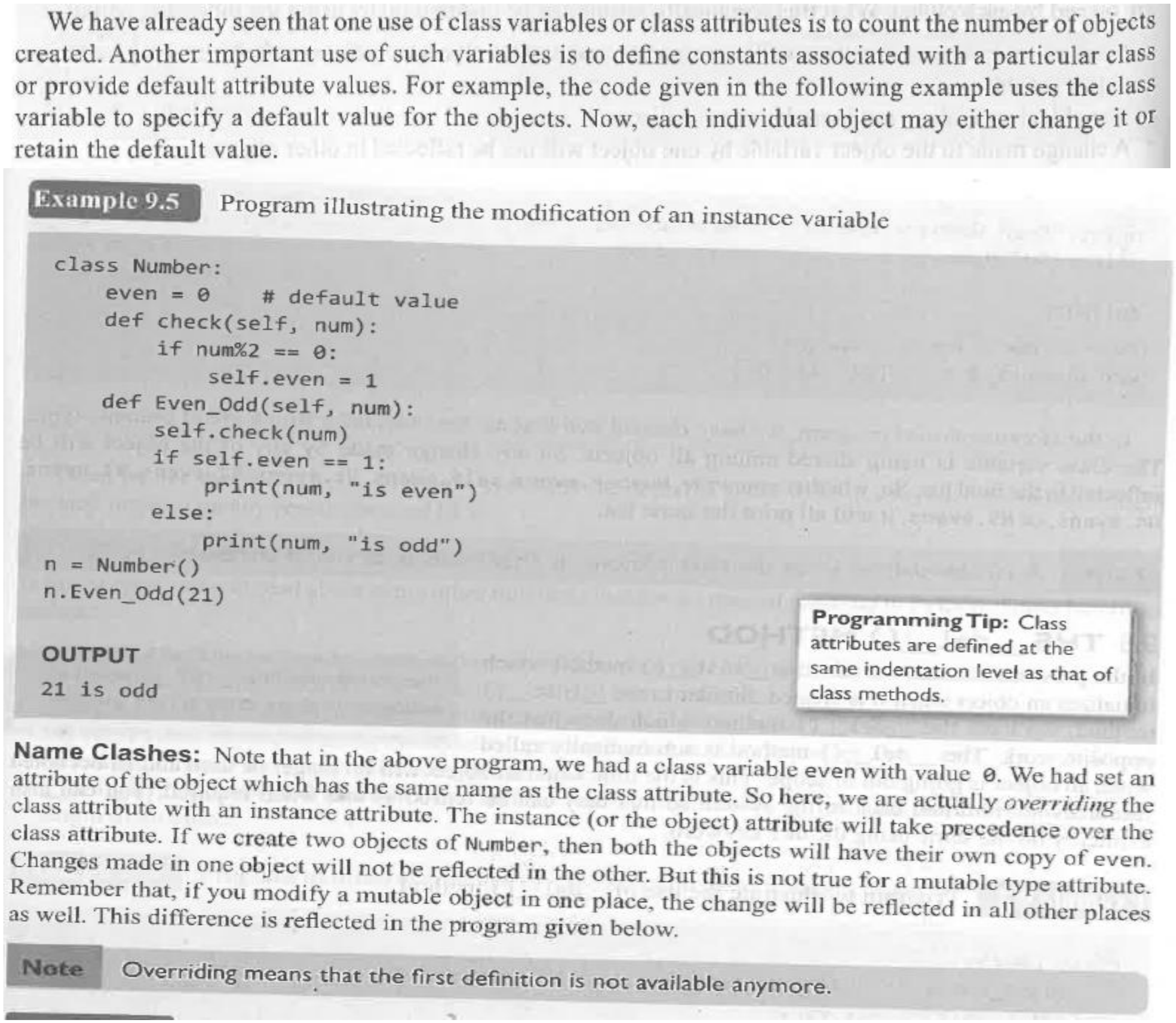
```
class ABC():
    class_var = 0    # class variable
    def __init__(self,var):
        ABC.class_var += 1
        self.var = var  # object variable
        print("The Object value is : ", var)
        print("The value of class variable is : ", ABC.class_var)
obj1 = ABC(10)
obj2 = ABC(20)
obj3 = ABC(30)
```

**OUTPUT**

```
The Object value is :  10
The value of class variable is :  1
The Object value is :  20
The value of class variable is :  2
The Object value is :  30
The value of class variable is :  3
```

**Programming Tip:** Class variable must be prefixed by the class name and dot operator.

www.iitkirba.xyz



We have already seen that one use of class variables or class attributes is to count the number of objects created. Another important use of such variables is to define constants associated with a particular class or provide default attribute values. For example, the code given in the following example uses the class variable to specify a default value for the objects. Now, each individual object may either change it or retain the default value.

**Example 9.5** Program illustrating the modification of an instance variable

```
class Number:
    even = 0     # default value
    def check(self, num):
        if num%2 == 0:
            self.even = 1
    def Even_Odd(self, num):
        self.check(num)
        if self.even == 1:
            print(num, "is even")
        else:
            print(num, "is odd")
n = Number()
n.Even_Odd(21)
```

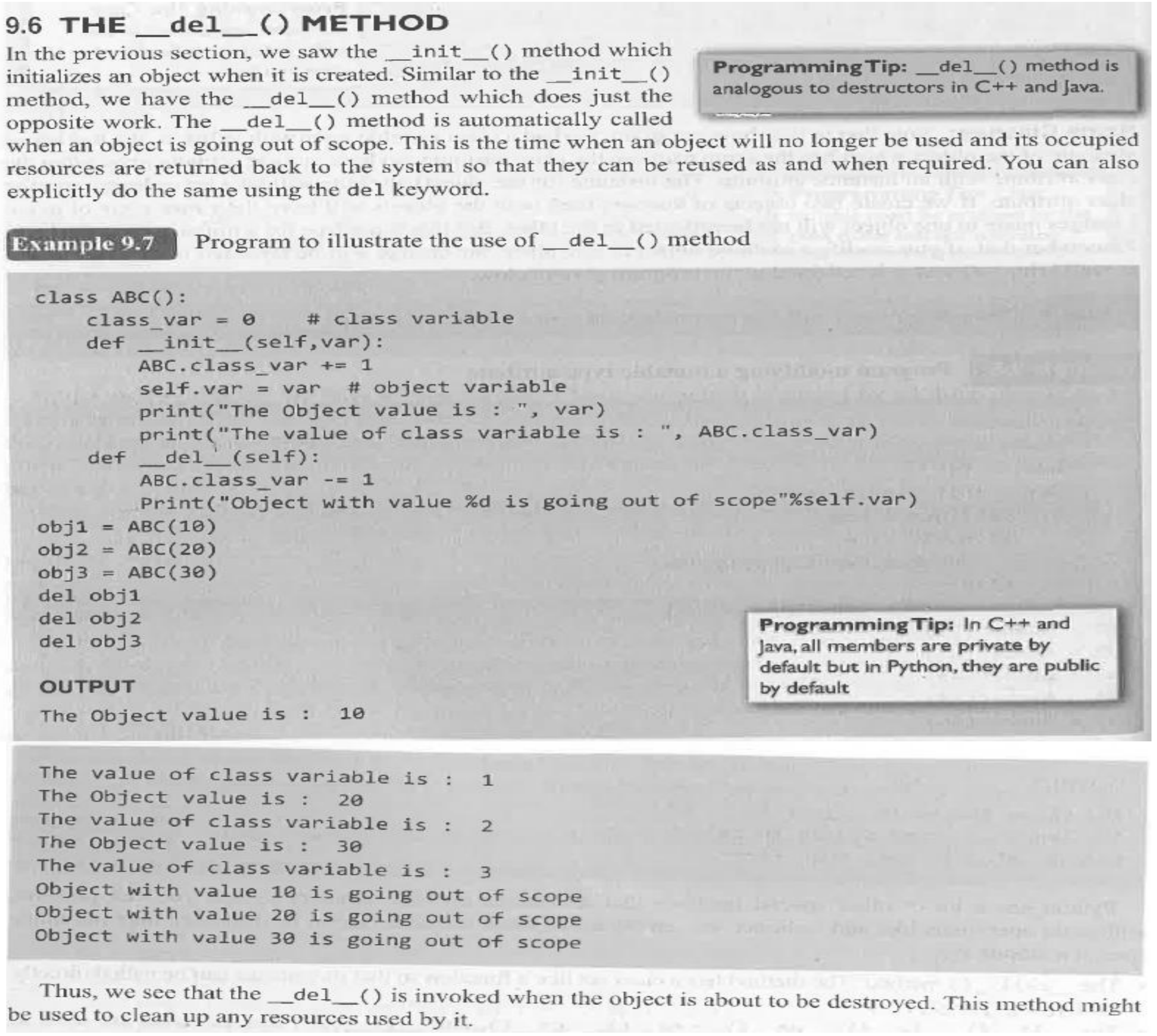
**OUTPUT**

```
21 is odd
```

**Programming Tip:** Class attributes are defined at the same indentation level as that of class methods.

**Name Clashes:** Note that in the above program, we had a class variable even with value `0`. We had set an attribute of the object which has the same name as the class attribute. So here, we are actually *overriding* the class attribute with an instance attribute. The instance (or the object) attribute will take precedence over the class attribute. If we create two objects of `Number`, then both the objects will have their own copy of even. Changes made in one object will not be reflected in the other. But this is not true for a mutable type attribute. Remember that, if you modify a mutable object in one place, the change will be reflected in all other places as well. This difference is reflected in the program given below.

**Note** Overriding means that the first definition is not available anymore.



## 9.6 THE __del__() METHOD

In the previous section, we saw the __init__() method which initializes an object when it is created. Similar to the __init__() method, we have the __del__() method which does just the opposite work. The __del__() method is automatically called when an object is going out of scope. This is the time when an object will no longer be used and its occupied resources are returned back to the system so that they can be reused as and when required. You can also explicitly do the same using the del keyword.

**Programming Tip:** __del__() method is analogous to destructors in C++ and Java.

**Example 9.7** Program to illustrate the use of __del__() method

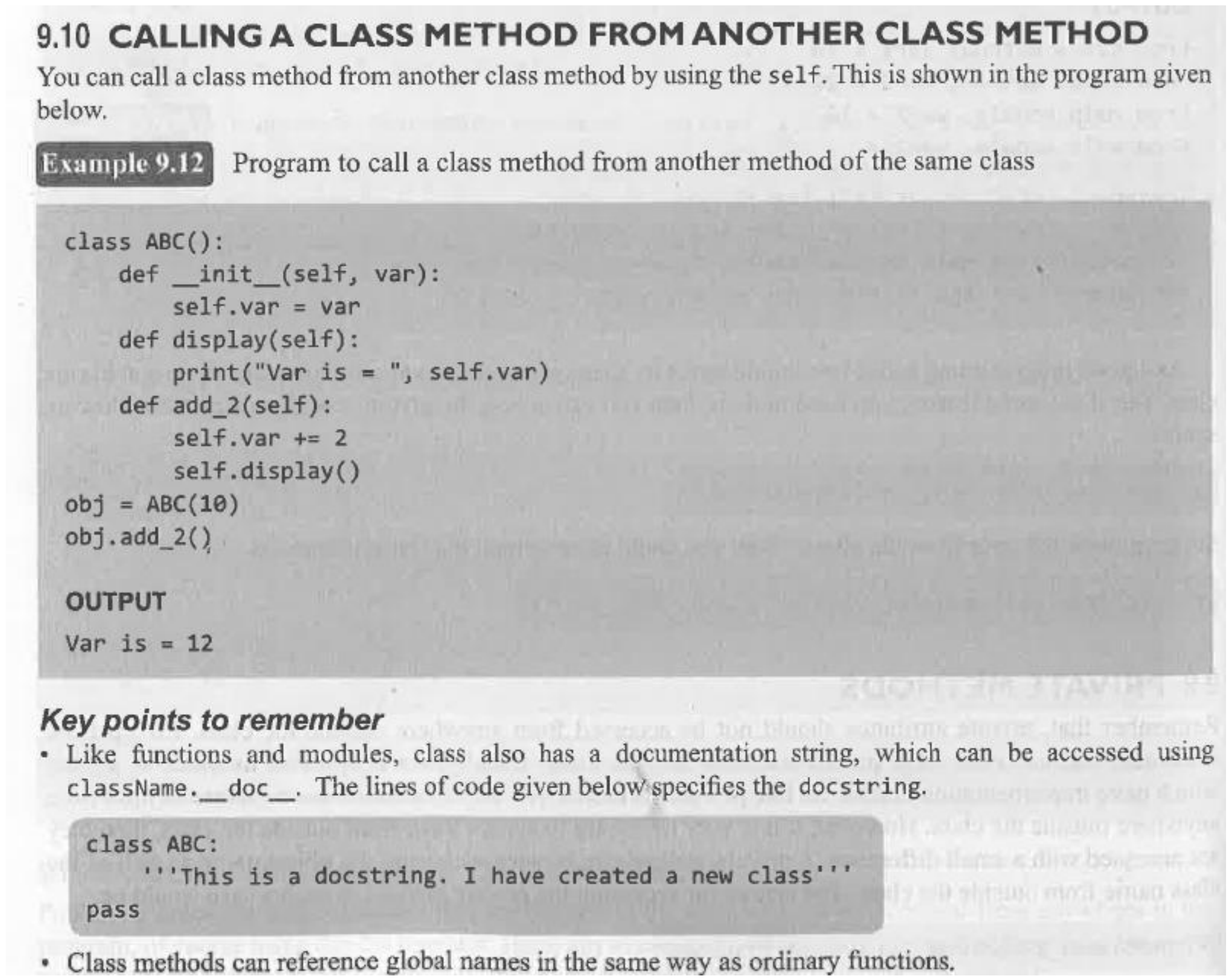
```
class ABC():
    class_var = 0     # class variable
    def __init__(self,var):
        ABC.class_var += 1
        self.var = var  # object variable
        print("The Object value is : ", var)
        print("The value of class variable is : ", ABC.class_var)
    def __del__(self):
        ABC.class_var -= 1
        Print("Object with value %d is going out of scope"%self.var)
obj1 = ABC(10)
obj2 = ABC(20)
obj3 = ABC(30)
del obj1
del obj2
del obj3
```

**Programming Tip:** In C++ and Java, all members are private by default but in Python, they are public by default

**OUTPUT**

```
The Object value is :  10
The value of class variable is :  1
The Object value is :  20
The value of class variable is :  2
The Object value is :  30
The value of class variable is :  3
Object with value 10 is going out of scope
Object with value 20 is going out of scope
Object with value 30 is going out of scope
```

Thus, we see that the __del__() is invoked when the object is about to be destroyed. This method might be used to clean up any resources used by it.



## 9.10 CALLING A CLASS METHOD FROM ANOTHER CLASS METHOD

You can call a class method from another class method by using the `self`. This is shown in the program given below.

**Example 9.12** Program to call a class method from another method of the same class

```
class ABC():
    def __init__(self, var):
        self.var = var
    def display(self):
        print("Var is = ", self.var)
    def add_2(self):
        self.var += 2
        self.display()
obj = ABC(10)
obj.add_2()
```

**OUTPUT**

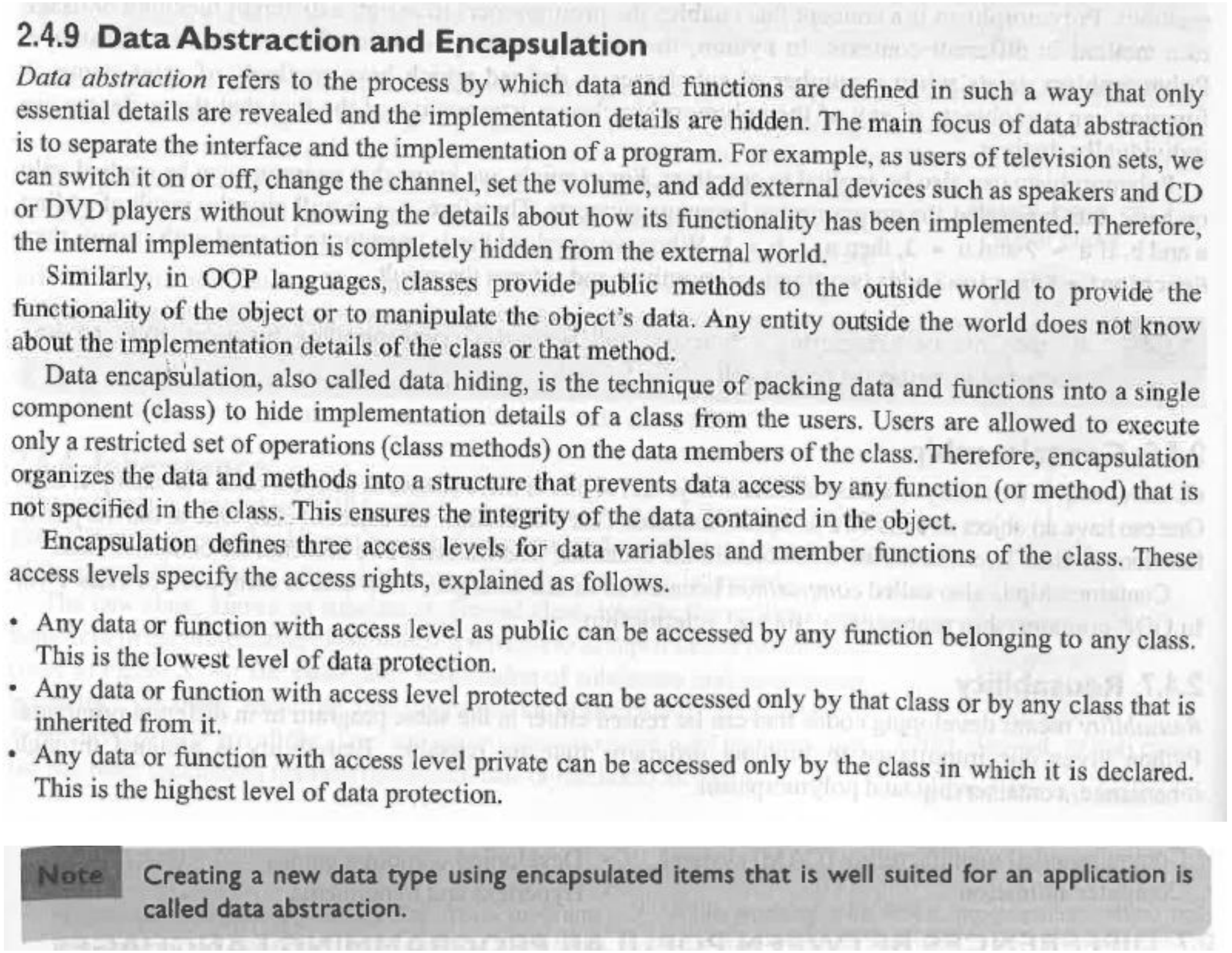
```
Var is = 12
```

***Key points to remember***

- Like functions and modules, class also has a documentation string, which can be accessed using `className.__doc__`. The lines of code given below specifies the `docstring`.

```
class ABC:
    '''This is a docstring. I have created a new class'''
pass
```

- Class methods can reference global names in the same way as ordinary functions.



### 2.4.9 Data Abstraction and Encapsulation

*Data abstraction* refers to the process by which data and functions are defined in such a way that only essential details are revealed and the implementation details are hidden. The main focus of data abstraction is to separate the interface and the implementation of a program. For example, as users of television sets, we can switch it on or off, change the channel, set the volume, and add external devices such as speakers and CD or DVD players without knowing the details about how its functionality has been implemented. Therefore, the internal implementation is completely hidden from the external world.

Similarly, in OOP languages, classes provide public methods to the outside world to provide the functionality of the object or to manipulate the object's data. Any entity outside the world does not know about the implementation details of the class or that method.

Data encapsulation, also called data hiding, is the technique of packing data and functions into a single component (class) to hide implementation details of a class from the users. Users are allowed to execute only a restricted set of operations (class methods) on the data members of the class. Therefore, encapsulation organizes the data and methods into a structure that prevents data access by any function (or method) that is not specified in the class. This ensures the integrity of the data contained in the object.

Encapsulation defines three access levels for data variables and member functions of the class. These access levels specify the access rights, explained as follows.

- Any data or function with access level as public can be accessed by any function belonging to any class. This is the lowest level of data protection.
- Any data or function with access level protected can be accessed only by that class or by any class that is inherited from it.
- Any data or function with access level private can be accessed only by the class in which it is declared. This is the highest level of data protection.

**Note** Creating a new data type using encapsulated items that is well suited for an application is called data abstraction.

## 12.1.3 Exceptions

Even if a statement is syntactically correct, it may still cause an error when executed. Such errors that occur at run-time (or during execution) are known as *exceptions*. An exception is an event, which occurs during the execution of a program and disrupts the normal flow of the program's instructions. When a program encounters a situation which it cannot deal with, it raises an exception. Therefore, we can say that an exception is a Python object that represents an error.

When a program raises an exception, it must handle the exception or the program will be immediately terminated. You can handle exceptions in your programs to end it gracefully, otherwise, if exceptions are not handled by programs, then error messages are generated. Let us see some examples in which exceptions occurs.

```
  • >>> 5/0
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    5/0
ZeroDivisionError: integer division or modulo by zero
  • >>> var + 10
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    var + 10
NameError: name 'var' is not defined
  • >>> 'Roll No' + 123
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    'Roll No' + 123
TypeError: cannot concatenate 'str' and 'int' objects
```

**Programming Tip:** Standard exception names are built-in identifiers and not reserved keywords.

In all the three cases discussed above, we have seen three types of exceptions had occurred. Since they were not handled in the code, an appropriate error message was displayed to indicate what had happened.

The string printed as the exception type (like `TypeError`) is the name of the built-in exception that occurred. However, this is not true for user-defined exceptions.

## 12.2 HANDLING EXCEPTIONS

We can handle exceptions in our program by using `try` block and `except` block. A critical operation which can raise exception is placed inside the `try` block and the code that handles exception is written in `except` block. The syntax for `try-except` block can be given as,

```
try:
    statements
except ExceptionName:
    statements
```

**Programming Tip:** Handlers do not handle exceptions that occur in statements outside the corresponding `try` block.

The `try` statement works as follows.

Step 1: First, the *try block* (statement(s) between the `try` and `except` keywords) is executed.

Step 2a: If no exception occurs, the *except block* is skipped.

Step 2b: If an exception occurs, during execution of any statement in the try block, then,

i. Rest of the statements in the `try` block are skipped.

ii. If the exception type matches the exception named after the `except` keyword, the except block is executed and then execution continues after the `try` statement.

iii. If an exception occurs which does not match the exception named in the `except` block, then it is passed on to outer `try` block (in case of nested try blocks). If no exception handler is found in the program, then it is an *unhandled exception* and the program is terminated with an error message (Refer Figure 12.2).

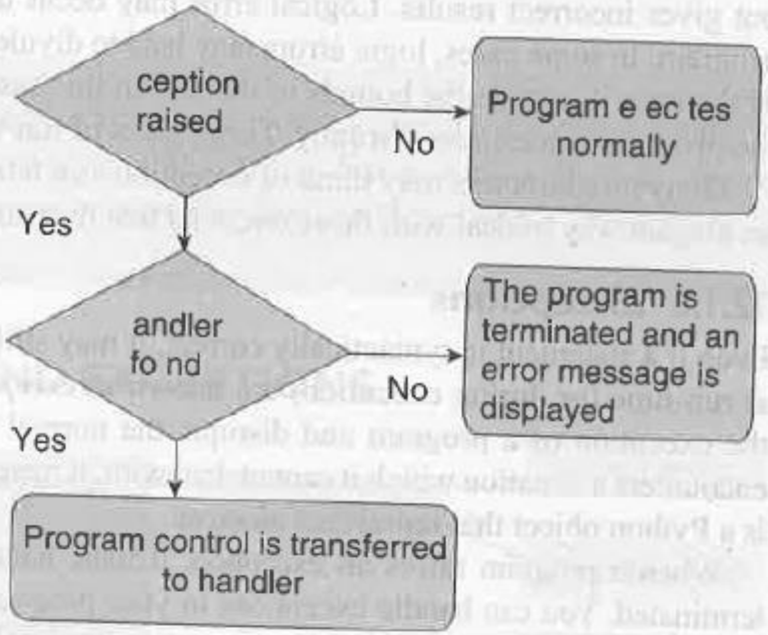
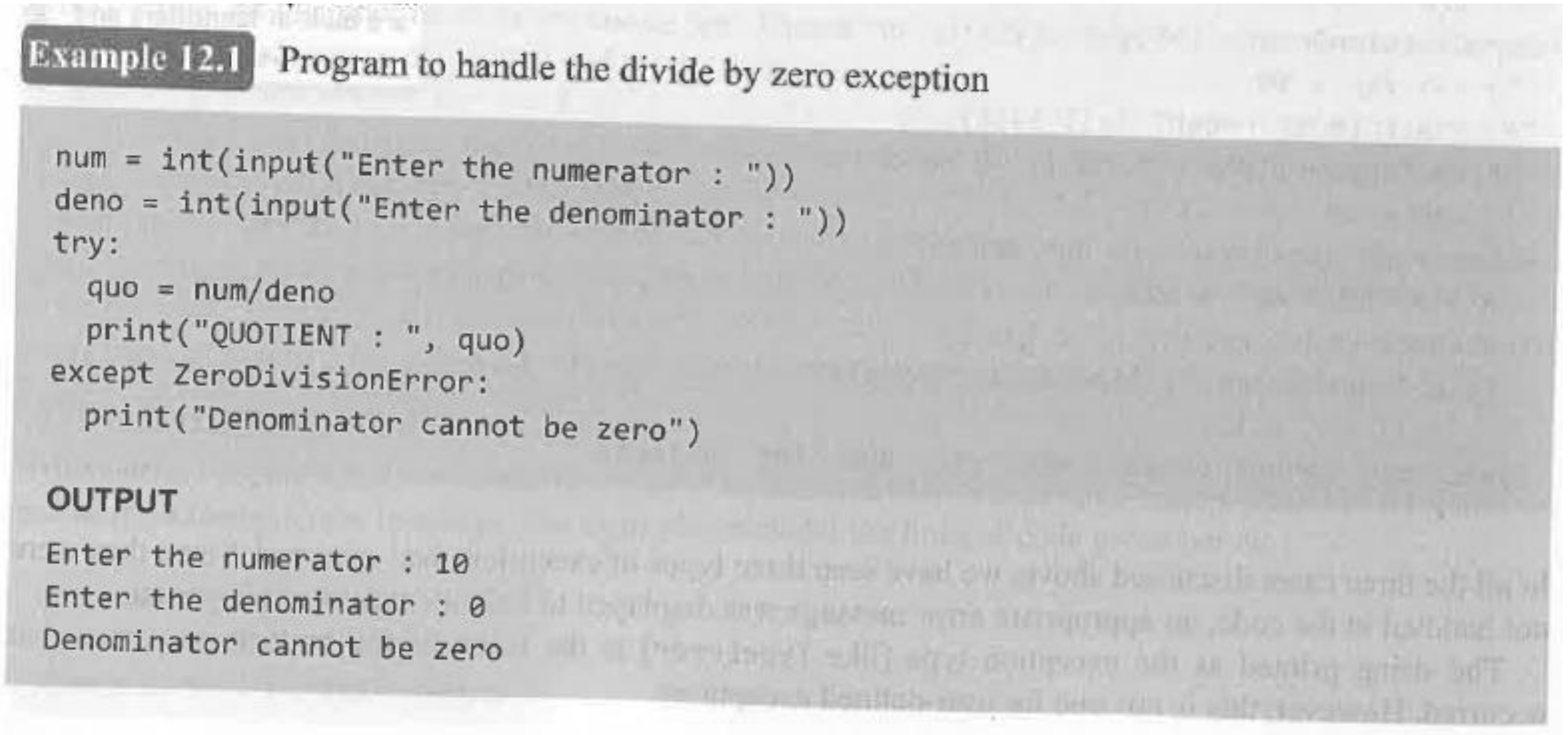

ception raised
No
Program e ec tes normally
Yes
andler fo nd
No
The program is terminated and an error message is displayed
Yes
Program control is transferred to handler

**Figure 12.2** Flowchart for Case iii under Step 2b for `try` statements

In the aforementioned program, note that a number was divided by zero, an exception occurred so the control passed to the `except` block.

www.iitkirba.xyz



**Example 12.1** Program to handle the divide by zero exception

```
num = int(input("Enter the numerator : "))
deno = int(input("Enter the denominator : "))
try:
  quo = num/deno
  print("QUOTIENT : ", quo)
except ZeroDivisionError:
  print("Denominator cannot be zero")
```

**OUTPUT**

```
Enter the numerator : 10
Enter the denominator : 0
Denominator cannot be zero
```

## 12.3 MULTIPLE EXCEPT BLOCKS

Python allows you to have multiple except blocks for a single try block. The block which matches with the exception generated will get executed. A try block can be associated with more than one except block to specify handlers for different exceptions. However, only one handler will be executed. Exception handlers only handle exceptions that occur in the corresponding try block. We can write our programs that handle selected exceptions. The syntax for specifying multiple except blocks for a single try block can be given as,

```
try:
    operations are done in this block
    ......................
except Exception1:
    If there is Exception1, then execute this block.
except Exception2:
    If there is Exception2, then execute this block.
    ......................
else:
    If there is no exception then execute this block.
    ......................
```

**Programming Tip**
try-except block is same as try-catch block. Exceptions are generated using raise keyword rather than throw.

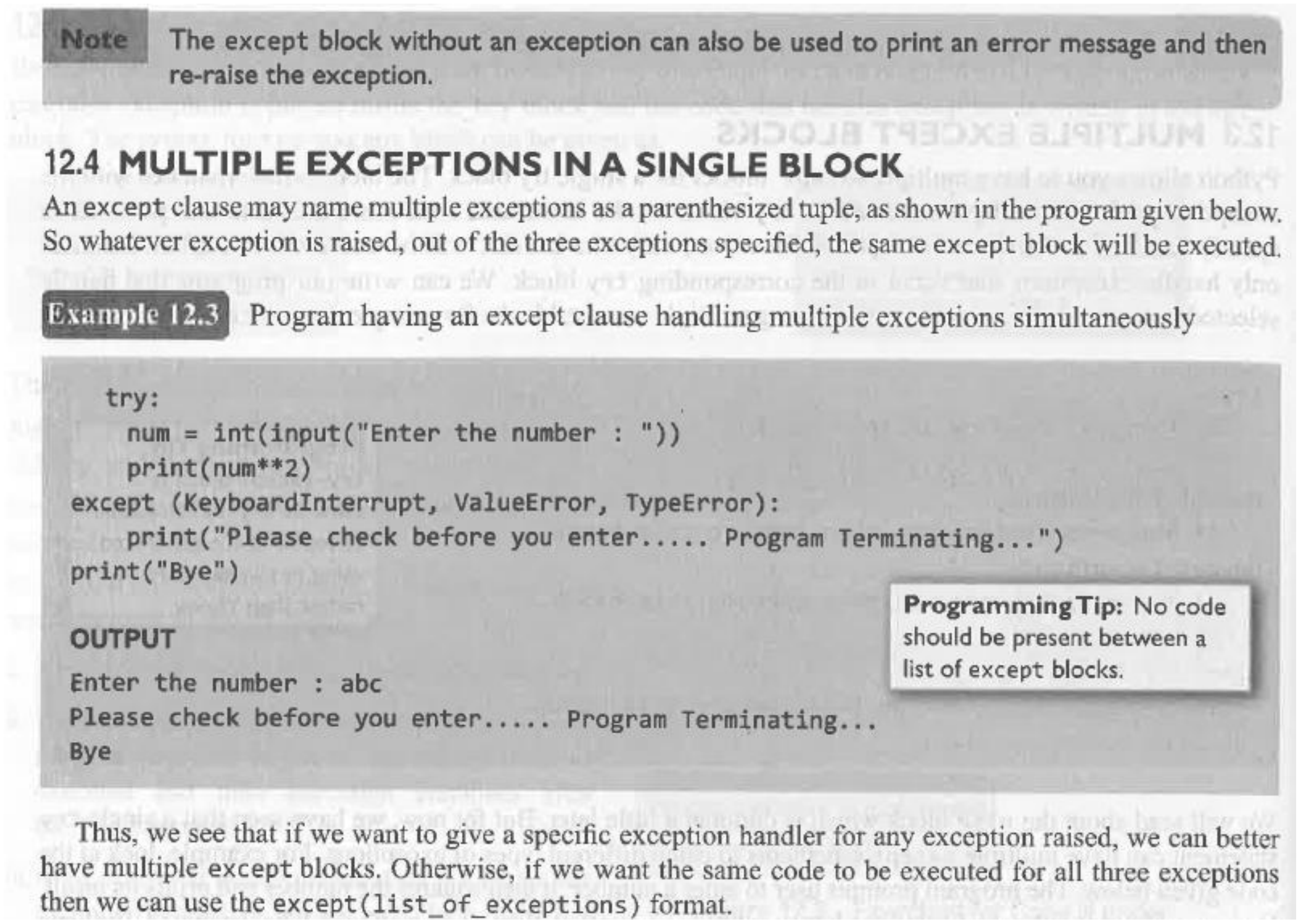
We will read about the else block which is optional a little later. But for now, we have seen that a single try statement can have multiple except statements to catch different types of exceptions. For example, look at the code given below. The program prompts user to enter a number. It then squares the number and prints its result. However, if we do not specify any number or enter a non-number, then an exception will be generated. We have two except blocks. The one matching the case will finally execute. This is very much evident from the output.

**Example 12.2** Program with multiple except blocks

```
try:
    num = int(input("Enter the number : "))
    print(num**2)
except (KeyboardInterrupt):
    print("You should have enterd a number..... Program Terminating...")
except (ValueError):
    print("Please check before you enter..... Program Terminating...")
print("Bye")
```

**OUTPUT**

```
Enter the number : abc
Please check before you enter..... Program Terminating...
Bye
```



**Note** The except block without an exception can also be used to print an error message and then re-raise the exception.

## 12.4 MULTIPLE EXCEPTIONS IN A SINGLE BLOCK

An `except` clause may name multiple exceptions as a parenthesized tuple, as shown in the program given below. So whatever exception is raised, out of the three exceptions specified, the same `except` block will be executed.

**Example 12.3** Program having an except clause handling multiple exceptions simultaneously

```
  try:
    num = int(input("Enter the number : "))
    print(num**2)
except (KeyboardInterrupt, ValueError, TypeError):
    print("Please check before you enter..... Program Terminating...")
print("Bye")
```

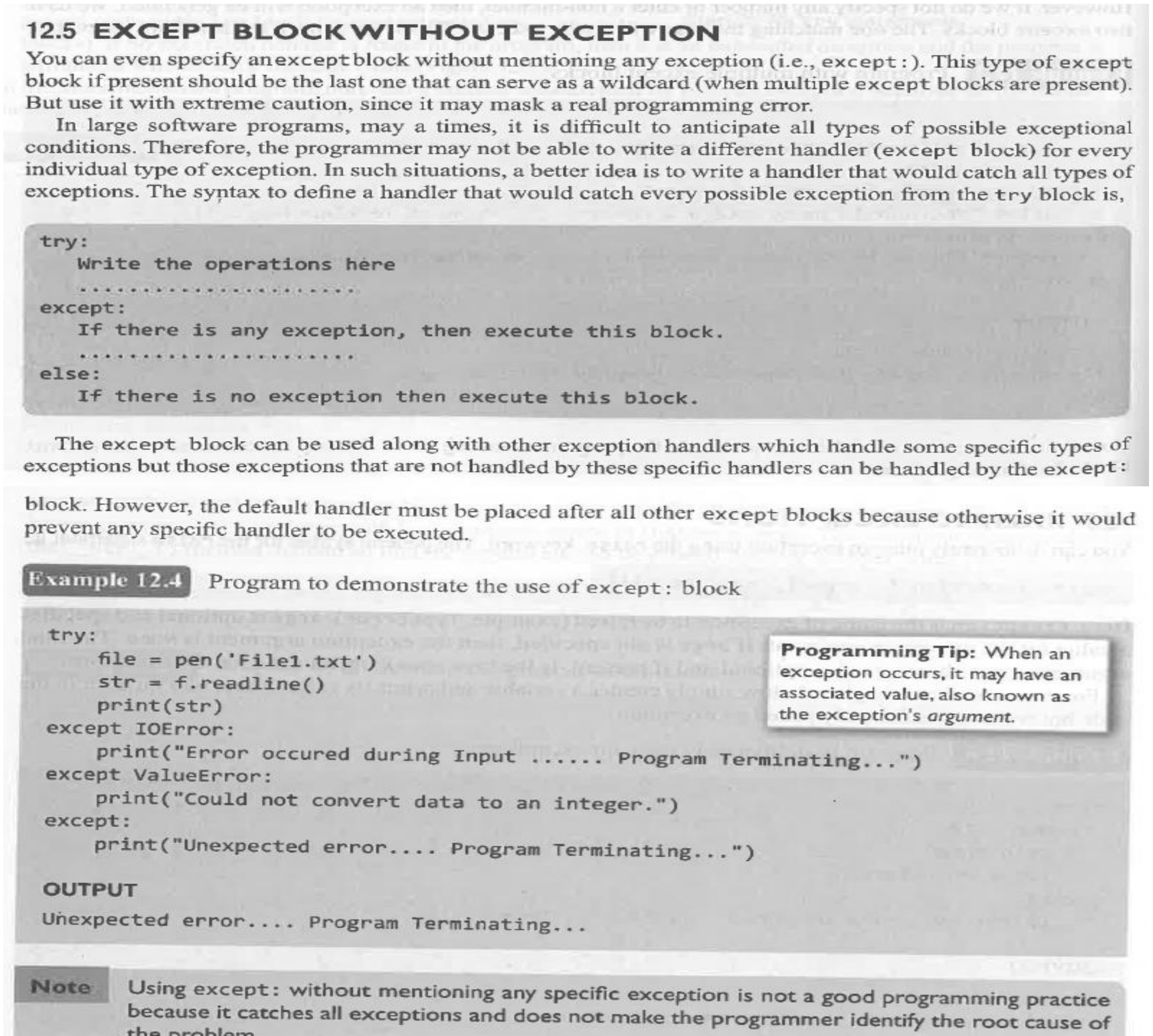
**OUTPUT**

```
Enter the number : abc
Please check before you enter..... Program Terminating...
Bye
```

**Programming Tip:** No code should be present between a list of except blocks.

Thus, we see that if we want to give a specific exception handler for any exception raised, we can better have multiple `except` blocks. Otherwise, if we want the same code to be executed for all three exceptions then we can use the `except(list_of_exceptions)` format.

www.iitkirba.xyz



## 12.5 EXCEPT BLOCK WITHOUT EXCEPTION

You can even specify an `except` block without mentioning any exception (i.e., `except:`). This type of except block if present should be the last one that can serve as a wildcard (when multiple `except` blocks are present). But use it with extreme caution, since it may mask a real programming error.

In large software programs, may a times, it is difficult to anticipate all types of possible exceptional conditions. Therefore, the programmer may not be able to write a different handler (`except` block) for every individual type of exception. In such situations, a better idea is to write a handler that would catch all types of exceptions. The syntax to define a handler that would catch every possible exception from the `try` block is,

```
try:
    Write the operations here
    ......................
except:
    If there is any exception, then execute this block.
    ......................
else:
    If there is no exception then execute this block.
```

The `except` block can be used along with other exception handlers which handle some specific types of exceptions but those exceptions that are not handled by these specific handlers can be handled by the `except:` block. However, the default handler must be placed after all other `except` blocks because otherwise it would prevent any specific handler to be executed.

**Example 12.4** Program to demonstrate the use of `except:` block

```
try:
    file = pen('File1.txt')
    str = f.readline()
    print(str)
except IOError:
    print("Error occured during Input ...... Program Terminating...")
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error.... Program Terminating...")
```

**Programming Tip:** When an exception occurs, it may have an associated value, also known as the exception's *argument*.

**OUTPUT**

```
Unexpected error.... Program Terminating...
```

**Note** Using `except:` without mentioning any specific exception is not a good programming practice because it catches all exceptions and does not make the programmer identify the root cause of the problem.

## 12.11 THE finally BLOCK

The `try` block has another optional block called `finally` which is used to define clean-up actions that must be executed under all circumstances. The `finally` block is always executed before leaving the `try` block. This means that the statements written in `finally` block are executed irrespective of whether an exception has occurred or not. The syntax of finally block can be given as,

```
try:
    Write your operations here
    ......................
    Due to any exception, operations written here will be skipped
finally:
    This would always be executed.
    ......................
```

Let us see with the help of a program how `finally` block will behave when an exception is raised in the `try` block and is not handled by `except` block.

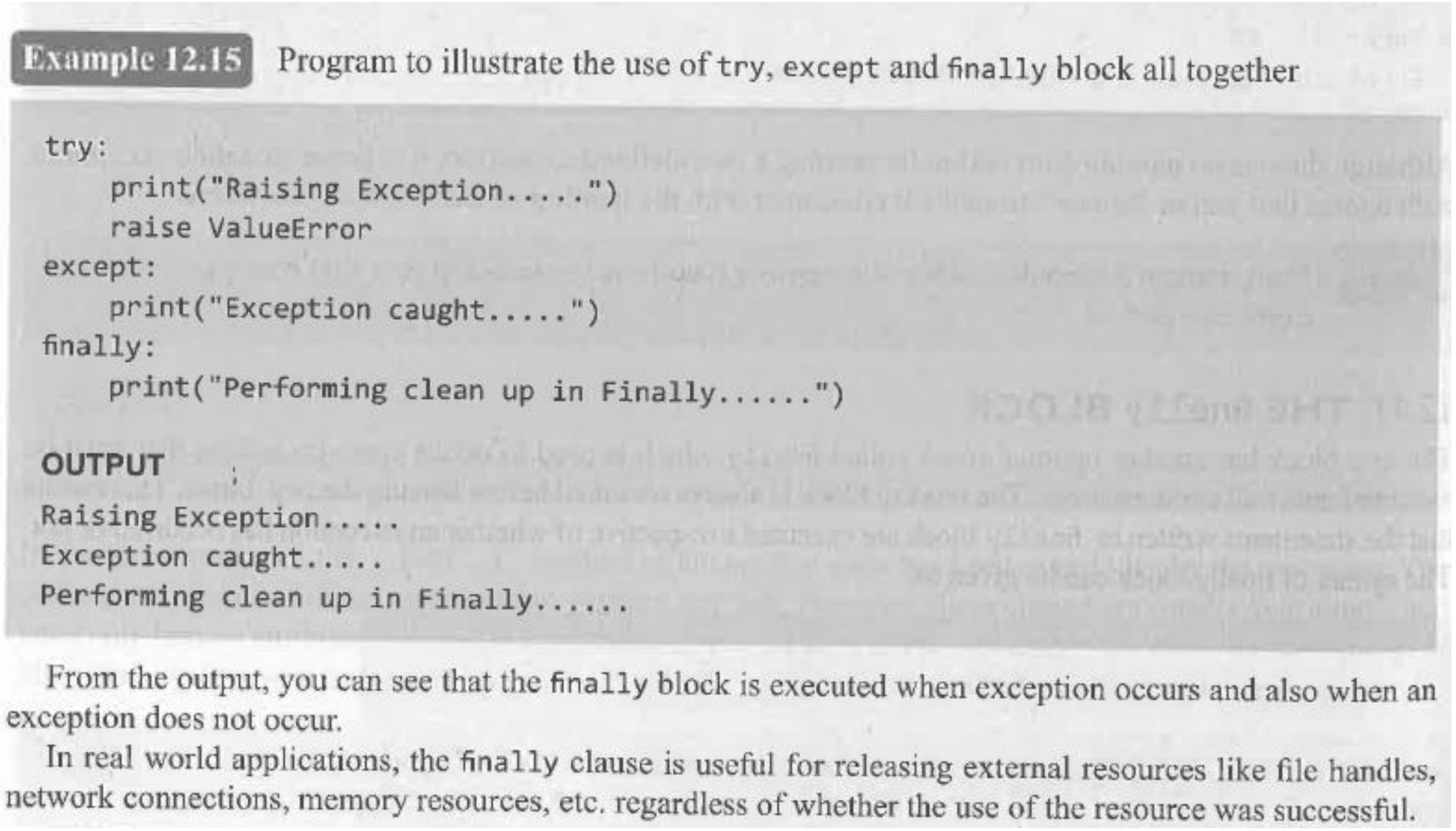
**Example 12.14** Program with `finally` block that leaves the exception unhandled

```
try:
    print("Raising Exception.....")
    raise ValueError
finally:
    print("Performing clean up in Finally......")
```

**OUTPUT**

```
Raising Exception.....
Performing clean up in Finally......
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 4, in <module>
    raise ValueError
ValueError
```

From the above code, we can conclude that when an exception occurs in the `try` block and is not handled by an `except` block or if the exception occurs in the `except` or `else` block, then it is re-raised after executing the `finally` block. The `finally` block is also executed when any block of the `try` block is exited via a `break`, `continue` or `return` statement.



**Example 12.15** Program to illustrate the use of `try`, `except` and `finally` block all together

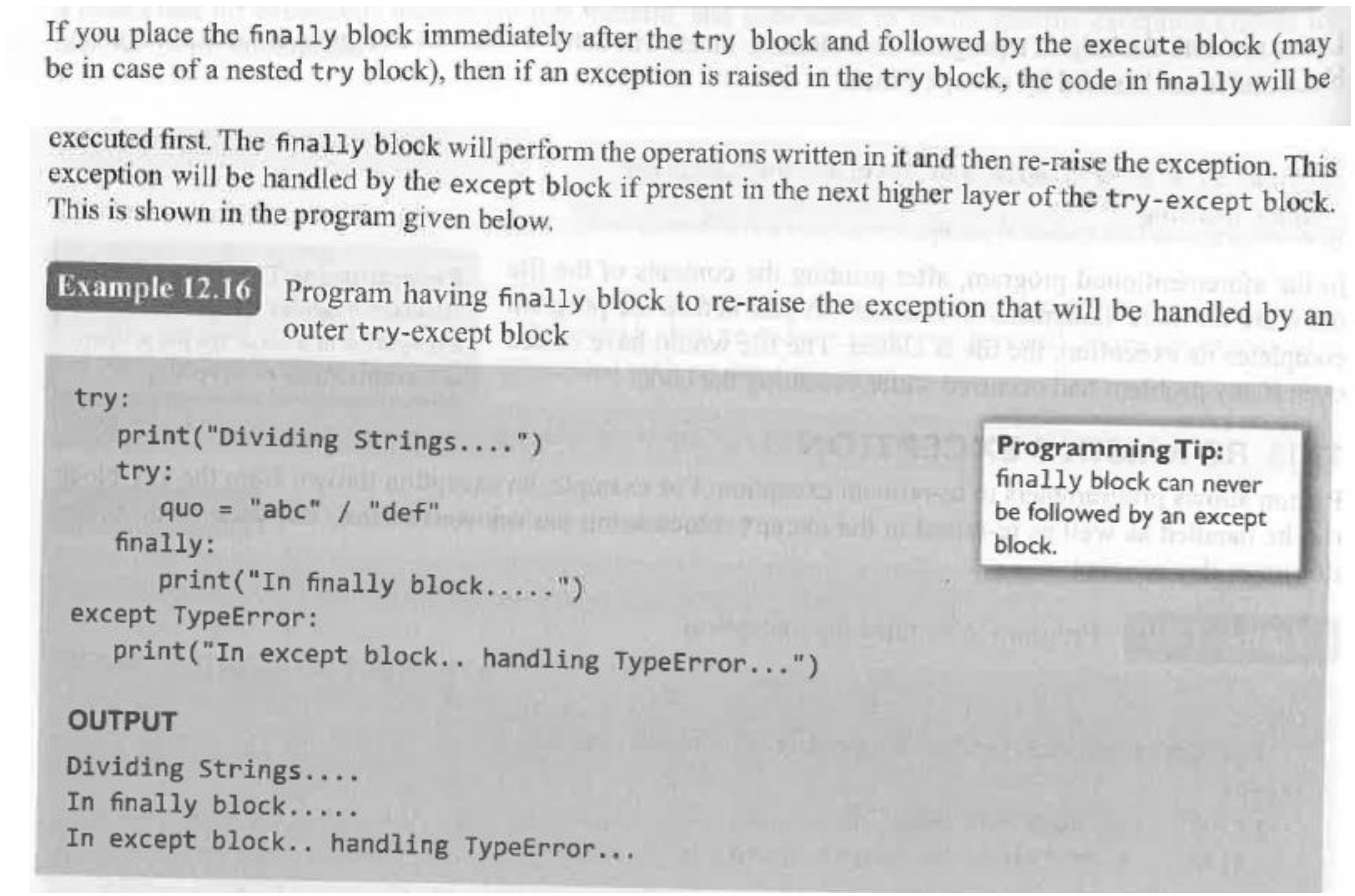
```
try:
    print("Raising Exception.....")
    raise ValueError
except:
    print("Exception caught.....")
finally:
    print("Performing clean up in Finally......")
```

**OUTPUT**

```
Raising Exception.....
Exception caught.....
Performing clean up in Finally......
```

From the output, you can see that the `finally` block is executed when exception occurs and also when an exception does not occur.

In real world applications, the `finally` clause is useful for releasing external resources like file handles, network connections, memory resources, etc. regardless of whether the use of the resource was successful.



If you place the finally block immediately after the try block and followed by the execute block (may be in case of a nested try block), then if an exception is raised in the try block, the code in finally will be executed first. The finally block will perform the operations written in it and then re-raise the exception. This exception will be handled by the except block if present in the next higher layer of the try-except block. This is shown in the program given below.

**Example 12.16** Program having finally block to re-raise the exception that will be handled by an outer try-except block

```
try:
   print("Dividing Strings....")
   try:
      quo = "abc" / "def"
   finally:
      print("In finally block.....")
except TypeError:
   print("In except block.. handling TypeError...")
```

**OUTPUT**

```
Dividing Strings....
In finally block.....
In except block.. handling TypeError...
```

**Programming Tip:** finally block can never be followed by an except block.