

## Contents

1. Concepts of object-oriented programming .....	2
2. Classes and Objects .....	3

## 1. Concepts of object-oriented programming

Object-oriented programming revolves around defining and using new types.

In python a class is how python represents a type. A function `isinstance()` reports whether an object is an instance of a class. i.e whether an object has a particular type.

➔ `isinstance('abc', str)`

➔ `True`

➔ `isinstance(55.2, str)`

➔ `False`

➔ `isinstance('abc', object)`

➔ `True`

➔ `isinstance(max, object)`

➔ `True`

Python has a class called `object`. Every other class is based on it. Even classes and functions are instances of `object`.

OOP Concepts:

**Class:** A class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods). A class is a user-defined data type that serves as a template for creating objects. It defines a set of attributes and methods that the created objects can use.

**Object:** An object is an instance of a class. It is a self-contained component that contains attributes and methods needed to make a particular type of data useful. When a class is defined, no memory is allocated until an object of that class is created. Objects are individual instances of a class that can have different values for the attributes defined in the class.

**Attributes:** Attributes are the variables that belong to an object. They are used to store the state of an object.

**Methods:** Methods are functions that belong to a class and define the behaviors of the objects created from the class.

**Constructor:** A constructor is a special method that is automatically called when an object of a class is created. It initializes the attributes of the object.

**Inheritance:** Inheritance is a mechanism where a new class (child class) inherits the attributes and methods of an existing class (parent class). This allows for code reuse and the creation of a hierarchical relationship between classes.

**Encapsulation:** Encapsulation is the bundling of data (attributes) and methods that operate on the data into a single unit, or class. It restricts direct access to some of an object's components, which can help prevent the accidental modification of data.

**Abstraction:** Abstraction is the concept of hiding the complex implementation details and showing only the essential features of the object. It helps in reducing programming complexity and effort.

**Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common super class. It is typically used to call methods that behave differently depending on the object that invokes the method.

**Overriding:** Overriding occurs when a child class provides a specific implementation of a method that is already defined in its parent class.

**Overloading:** Overloading is a feature that allows a class to have more than one method having the same name, if their parameter lists are different.

## 2. Classes and Objects

Syntax of class:

```
class ClassName:  
    # body of class
```

```
class_and_objects.py > ...
1 class Student:
2
3     # Attributes of class
4     name = ""
5     age = 0
6
7 # Creating an instance of class i.e object
8 stu1 = Student()
9 stu1.name = "Ram"
10 stu1.age = 20
11
12 stu2 = Student()
13 stu2.name = "Sita"
14 stu2.age = 30
15
16 print(f"{stu1.name} is {stu1.age} years old")
17 print(f"{stu2.name} is {stu2.age} years old")
18
19 print(type(Student), type(stu1))
```

The given code defines a Student class with class-level attributes name and age, and then creates two instances of this class (stu1 and stu2) where the attributes are individually set for each instance. It prints out the name and age for each instance, demonstrating that these attributes can hold different values for different instances. The code also prints the types of the Student class and the stu1 instance, showing <class 'type'> for the class itself and <class '\_\_main\_\_.Student'> for the instance, indicating that Student is a class (created using the type metaclass) and stu1 is an instance of Student.

Try the following:

➔ `print(isinstance(stu1, Student))`

➔ `print(isinstance(stu1, object))`

```
1 class Book:
2
3     '''Information about book'''
4
5
6 python_book = Book()
7 python_book.title = 'Introduction to Python Programming'
8 python_book.authors = ['Ram Thapa', 'Hari Thapa', 'Krishna Shrestha']
9 print(f"{python_book.title} by {' '.join(python_book.authors)}")
```

The first assignment statement creates a Book object and then assigns that object to variable `python_book`. The second assignment statement creates a title variable inside the Book object; that variable refers to the string 'Introduction to Python Programming'. The third assignment statement creates variable `authors`, also inside the Book object, which refers to the list of strings ['Ram Thapa','Hari Thapa','Krishna Shrestha'].

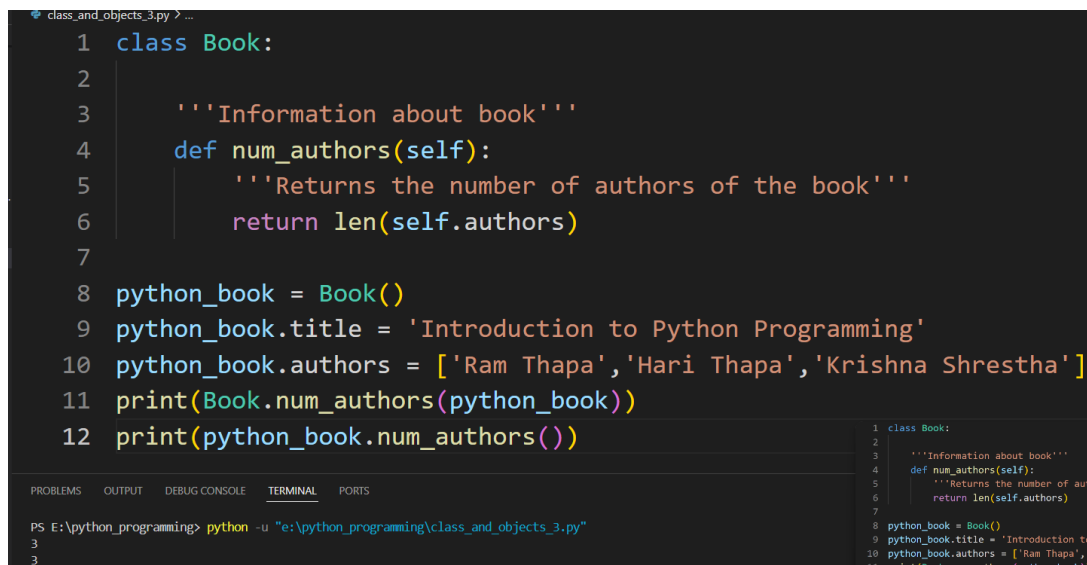
Variables `title` and `authors` are called instance variables because they are variables inside an instance of a class. We can access these instance variables through variable `python_book`.

### Writing a Method in Class Book:

We can define a method `num_authors` that will return the numbers of authors of the book. Book method `num_authors` looks just like a function except that it has a parameter called `self`, which refers to a Book. there are two ways to call a method. One way is to access the method through the class, and the other is to use object-oriented syntax.

➔ `Book.num_authors(python_book)`

➔ `Python_book.num_authors()`



```

1 class Book:
2
3     '''Information about book'''
4     def num_authors(self):
5         '''Returns the number of authors of the book'''
6         return len(self.authors)
7
8 python_book = Book()
9 python_book.title = 'Introduction to Python Programming'
10 python_book.authors = ['Ram Thapa','Hari Thapa','Krishna Shrestha']
11 print(Book.num_authors(python_book))
12 print(python_book.num_authors())

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\python\_programming> python -u "e:\python\_programming\class\_and\_objects\_3.py"

```

1 class Book:
2
3     '''Information about book'''
4     def num_authors(self):
5         '''Returns the number of authors of the book'''
6         return len(self.authors)
7
8 python_book = Book()
9 python_book.title = 'Introduction to Python Programming'
10 python_book.authors = ['Ram Thapa','Hari Thapa','Krishna Shrestha']
11 print(Book.num_authors(python_book))

```

Let's take a close look at the first call on method `num_authors`.

➔ `Book.num_authors(python_book)`

In class `Book` is method `num_authors`. The argument to the call, `python_book`, is passed to parameter `self`. Python treats the second call on `num_authors` exactly as it did the first; the first call is equivalent to this one

➔ `Python_book.num_authors()`

The second version is much more common because it lists the object first; we think of that version as asking the book how many authors it has. Thinking of method calls this way can really help develop an object-oriented mentality.

In the `python_book` example, we assigned the title and list of authors after the `Book` object was created. That approach isn't scalable; we don't want to have to type those extra assignment statements every time we create a `Book`. Instead, we'll write a method that does this for us as we create the `Book`. This is a special method and is called `__init__`. We'll also include the publisher, ISBN, and price as parameters of `__init__`:

```
class Book:

    """Information about a book, including title, list of authors publisher, ISBN, and price.
    """

    def __init__(self, title, authors, publisher, isbn, price):
        """Create a new book entitled title, written by the people in authors,
        published by publisher, with ISBN isbn and costing price dollars.

        >>> python_book = Book( \
            'Practical Programming', \
            ['Campbell', 'Gries', 'Montejo'], \
            'Pragmatic Bookshelf', \
            '978-1-6805026-8-8', \
            25.0)
        """
        self.title = title
        self.authors = authors[:]
        self.publisher = publisher
        self.ISBN = isbn
        self.price = price
```

```

def num_authors(self):
    """Returns the number of authors of the book"""
    return len(self.authors)

python_book = Book(
    'Practical Programming',
    ['Campbell', 'Gries', 'Montejo'],
    'Pragmatic Bookshelf',
    '978-1-6805026-8-8',
    25.0)

print(python_book.ISBN, python_book.title, python_book.authors)

```

Method `__init__` is called whenever a `Book` object is created. Its purpose is to initialize the new object; this method is sometimes called a constructor. Here are the steps that Python follows when creating an object:

1. It creates an object at a particular memory address.
2. It calls method `__init__`, passing in the new object into the parameter `self`.
3. It produces that object's memory address.

**Note: Methods belong to classes. Instance variables belong to objects. If we try to access an instance variable as we do a method, we get an error.**

### Another special method: `__str__()`

When we call `print(python_book)`, then `python_book.__str__()` is called to find out what string to print. The output Python produces when we print a `Book` isn't particularly useful:

```

36 print(python_book)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\python_programming> python -u "e:\python_programming\class_and_objects_4.py"
< main .Book object at 0x000001F96E0F3FD0>

```

This is the default behavior for converting objects to strings: it just shows us where the object is in memory. This is the behavior defined in class object's method `__str__`, which our Book class has inherited.

Let's define method `Book.__str__` to provide useful output; this method goes inside class `Book`, along with `__init__` and `num_authors`:

```
24 def __str__(self):
25     """Return a human-readable string representation of this Book.
26     """
27     return f'''Title: {self.title}
28 Author: {' , '.join(self.authors)}
29 Publisher: {self.publisher}
30 ISBN: {self.ISBN}
31 Price: Rs.{self.price}
32 '''
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\class_and_objects_4.py"
Title: Practical Programming
Author: Campbell, Gries, Montojo
Publisher: Pragmatic Bookshelf
ISBN: 978-1-6805026-8-8
Price: Rs.25.0
```

The result is displayed when `print(python_book)`

### Deleting Object Properties

To delete an attribute from an object, you can use the `del` statement followed by the object's attribute.

```
45 del python_book.price
46 print(python_book.price)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\class_and_objects_5.py"
Traceback (most recent call last):
  File "e:\python_programming\class_and_objects_5.py", line 46, in <module>
    print(python_book.price)
AttributeError: 'Book' object has no attribute 'price'
```

### Deleting Objects

```
48 del python_book
49 print(python_book)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS E:\python_programming> python -u "e:\python_programming\class_and_objects_5.py"
Traceback (most recent call last):
  File "e:\python_programming\class_and_objects_5.py", line 49, in <module>
    print(python_book)
NameError: name 'python book' is not defined
```