Python II: OOP I

Raul P. Pelaez

March 11, 2026

Contents

1 Introduction

Object Oriented Programming (OOP) is a paradigm that organizes your code around objects; entities that combine data and behavior.

Advice

Object oriented programming lies on some main principles:

- **Encapsulation**: Bundle data and methods that operate on that data.
- **Abstraction**: Hide complex implementation details behind a simple interface.
- **Inheritance and Polymorphism**: Reuse and extend existing classes.

In this first lesson on OOP, we will focus on **Encapsulation**. We will use this concept to cover the basic Python syntax for defining classes, creating objects, and understanding the role of methods and data members.

2 Encapsulation: Object Oriented Programming Fundamentals

Info

OOP revolves around the concept of classes and objects
A **class** is a blueprint that defines attributes (data members) and behaviors (methods). An **object** is an instance of a class created using that blueprint.
Example:

```
import numpy as np

v = np.array([1, 2, 3])  # v is an object of the np.array class
print(v.mean())
```

Here, `np.array` is the name of a class, `v` is an object of that class, and `mean()` is a method that operates on the object.

A class bundles data and methods that operate on that data. For instance, a class called `Deck` might represent a deck of cards and include a method called `shuffle()`. A class called `Animal` could have a data member called `number_of_legs` and a method called `eat()`.

Let us explore the basic syntax for defining classes and creating objects in Python. After this lesson, you will be able to understand the following code:

Example:

```python
class Person:
    species = "Human"
    def __init__(self, name, age):
        self.name = name       # Data member
        self.age = age         # Data member

    def greet(self):
        return f"Hello, my name is {self.name}. I am {self.age} years old."

# Creating an object (instance of Person)
person = Person("Alice", 20)
print(person.greet())  # Output: Hello, my name is Alice.
```

Output

```
Hello, my name is Alice. I am 20 years old.
```

Advice

The Python Data Model: Everything in Python is an object
In Python, everything is (or behaves like) an object, including integers, strings, and functions. Classes and objects are fundamental to Python's design, making it an object-oriented language. However, as some other programming languages, Python is flexible enough to adequate to other paradigms, like functional programming.

3 Class: Methods and Data Members

Classes encapsulate both data and behavior:

- **Data members (attributes)** represent the state of an object.
- **Instance Methods** (or just methods) are functions defined within a class that operate on its members/attributes.

Info

The dot operator

To access data members and methods of an object, use the dot operator (`.`). For example, `person.name` accesses the `name` data member of the `person` object. Since everything in Python is like an object, the dot operator is used extensively, for instance:

```
import numpy as np
v = np.array([1, 2, 3])
```

We use the dot operator to access the `array` method of the `numpy` module (which is like-an-object too!).

Classes are defined using the `class` keyword, followed by the class name.

```
class Person:
    pass
```

We just defined a class called `Person` that holds no data and no functionality. A little bare bones, so let us add some spice to it. `pass` is a placeholder that allows us to define an empty class without causing a syntax error.

3.1 Data members

There are two types of data members:

- **Class attributes**: Shared by all instances
- **Instance attributes**: Unique to each object

Info

Class attributes

```
class Person:
    species = "Human"
    number_of_legs = 2
```

In the example above, `species` and `number_of_legs` are class attributes of the `Person` class. They are shared by all instances of the class and can be accessed using the dot operator.

Objects (instances of a class) are created using the class name followed by parentheses.

Example

```
class Person:
    species = "Human"
    number_of_legs = 2

a_person = Person()
print(a_person.species) # Output: Human
print(a_person.number_of_legs) # Output: 2
```

Output

```
Human
2
```

The other type of data member is an **instance attribute**. These are unique to each object (for instance, the name of a person). We define instance attributes in the `__init__` dunder method, which is called when an object is created.

Advice

Member visibility
In Python, anyone can access any data member or method of an object. This can make it hard to separate between the implementation and the interface parts of a class. It is standard to name those members that are not meant to be accessed from outside the class with a leading underscore, like `_hidden`. The Python programmer can still access these members, but knows the author of the class designed it to be private.

3.2 Methods

As you can see in the example above, using a class as just a group of data members does not takes us too far. Classes are more powerful when they include methods that operate on the data members.

We can define our own methods, but some of them are special and have a predefined meaning. For example, if present, the `__init__` method is called when an object is created. All methods in a class must have the `self` parameter as the first argument. This parameter refers to the object itself, and it is used to access the object's data members and other methods.

```python
class Calculator:
    def __init__(self, initial=0):
        self.value = initial  # Instance data member

    def add(self, amount):
        self.value += amount  # Instance method using self
        return self.value

calc = Calculator(10) # Calls __init__ with initial=10
print(calc.add(5))              # Output: 15
print(calc.add(5))              # Output: 20
```

Output

```
15
20
```

Methods are defined within the class using the `def` keyword. Note that we do not need to pass `self` when calling these methods; Python infers this from the object on which the method is called.

Any variable that we assign to `self` within the class becomes an instance data member. These data members are unique to each object and can be accessed using the dot operator. In the example above, `value` is an instance data member of the `Calculator` class that we are setting in the `__init__` method.

Info

The `self` keyword
`self` is a reference to the current object. It is used to access the object's data members and methods from within the class. When calling a method, Python automatically passes the object as the first argument to the method. This is why we always define the first parameter of instance methods as `self`.

Info

The `__init__` method
The `__init__` method is a special method that initializes an object when it is created. It is called automatically when a new object is instantiated. The first parameter of the `__init__` method is always `self`, which refers to the object itself.
Example

```
class Person:
   def __init__(self):
      print("A new person has been created.")
a_person = Person()  # Prints: A new person has been created.
```

Output

```
A new person has been created.
```

Info

Instance data members
These are unique to each object and are defined within the `__init__` method. They are accessed using the dot operator and are used to store the state of an object. The `self` parameter can be used to access these data members within the class.
Example

```
class Person:
    def __init__(self, name, age):
        self.name = name      # Instance attribute
        self.age = age        # Instance attribute
alice = Person("Alice", 20)
bob = Person("Bob", 25)
print(alice.name)  # Output: Alice
print(bob.age)     # Output: 25
```

Output

```
Alice
25
```

Advanced

Dunder methods

Methods that start and end with double underscores (dunder) are special methods in Python. They have a predefined meaning and are called automatically in certain situations. For example, `__init__` is called when an object is created, and `__str__` is called when the object is converted to a string (like when printing). There is a long list of dunder methods that you can define in your classes to instill certain behaviors onto them (for instance, `__iter__` to make your class iterable like a `List`).

Example

```
class Book:
  def __init__(self, title, author):
    self.title = title    # Instance attribute
    self.author = author

  def __str__(self):
    return f"'{self.title}' by {self.author}"

novel = Book("1984", "George Orwell")
print(novel)
```

Output

```
'1984' by George Orwell
```

Advanced

Other types of methods

Besides instance methods, classes can also have:

- **Class methods**: Operate on the class itself, not on instances.
- **Static methods**: Do not operate on instances or classes. They are utility functions that are logically related to the class.

Defining class and static methods involves using the `@classmethod` and `@staticmethod` decorators, respectively. We will revisit these concepts in future lessons, when we learn about decorators. Example:

```
class Calculator:
    def __init__(self, initial=0):
        self.value = initial  # Instance data member

    def add(self, amount):
        self.value += amount  # Instance method using self
        return self.value

    @classmethod
    def pi(cls):
        return 3.14159  # Class method example

    @staticmethod
    def multiply(a, b):
        return a * b  # Static method example, does not need self or cls.

calc = Calculator(10)
print(calc.add(5))                # Output: 15
print(Calculator.pi())            # Output: 3.14159
print(Calculator.multiply(3, 4)) # Output: 12
```

Advanced

Monkey Patching
In Python, one can change the properties of a class during runtime, for instance, changing the meaning of a method or adding new methods to a class. This is a powerful (and dangerous) feature of Python and is known as **monkey patching**.
Example:

```
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        return f"Hello, my name is {self.name}."

Person.greet = lambda self: f"I do not want to greet you."
person = Person("Alice")
print(person.greet())
```

Output

```
I do not want to greet you.
```

4 Exercises

4.1 Your First Class

Goal

Create a Python class named `Student` with the following features:

- **Data members**: `name`, `student_id`, and `courses` (a list of enrolled courses)
- **Method** `enroll(course)`: Adds a course to the student's courses.
- **Get courses**: Returns the list of courses the student is currently enrolled in.

Test your class by creating an object and printing its details.

hint

Remember:

- Use the `__init__` method to initialize data members.
- Always include `self` in your instance methods.

Milestone

Implement good practices in your class:

- Add some docstrings and error handling.
- Add the script you are using to a git repository.
- Add a README file with instructions on how to run your script, and environment.yml file with the dependencies.
- Add some tests to your class. For instance, test that the `enroll` method increases the length of the `courses` list by one.

4.2 Monte Carlo integration

Goal

One form of numerical integration relies on randomly sampling points in a region of interest and calculating the fraction of points that fall within the function's area. This method is known as Monte Carlo integration. Formally, in its simplest form, we can make the following approximation:

$$I = \int_a^b f(x)dx \approx \frac{b-a}{N} \sum_{i=1}^{N} f(x_i)$$

Where x_i are random points in the interval $[a, b]$. The law of large numbers ensures that this approximation converges to the true value of the integral as N increases.
Create a Python class called `MonteCarlo` that performs Monte Carlo integration. The class should have the following features:

- **Data members**: `f` (a function that takes one argument, like a lambda), and `a` and `b` (the interval of integration).
- **Method** `integrate(N)`: Approximates the integral of `f` in the interval $[a, b]$ using Monte Carlo integration with `N` random points.
- **Method** `plot(N)`: Plots the result of the integration using `1` to `N` points, showing how the approximation converges to the true value.

hint

- The function received by your functions could be a lambda function, like `lambda x: x**2`.

Milestone

Try your class by integrating some functions for which you know the exact result. For instance, $\int_0^\pi \sin(x)dx = 2$. Add some tests using this.
Implement good practices in your class:

- Add some docstrings and error handling.
- Add the script you are using to a git repository.
- Add a README file with instructions on how to run your script, and environment.yml file with the dependencies.

hint

Your class is stochastic, meaning each time you run it, you will get a different result. Take this into account when testing your class. In particular, the error of your approximation will decrease as N increases as $1/\sqrt{N}$.

Advanced milestone

Approximating pi

Expand your class so that it can handle two dimensional functions. Use it to approximate the value of π by integrating the following function:

$$H(x,y) = \begin{cases} 1 & \text{if } x^2 + y^2 \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

If we integrate this function over the unit square $\Omega = [-1,1] \times [-1,1]$, the result is π:

$$\int_{\Omega} H(x,y)dxdy = \pi$$

hint

- One can interpret this algorithm as "throwing darts" at a square and counting how many fall inside a circle inscribed in it. The ratio of darts inside the circle to the total number of darts is an approximation of $\pi/4$.