Object Oriented Programming III

Raul P. Pelaez

March 17, 2026

Contents

1 Introduction

Today, we will focus on the OOP principles we left behind in the previous lessons. These are **Inheritance** and **Polymorphism**. Lets get some terminology out of the way first.

Info

Inheritance
Inheritance allows a class to, well, inherit attributes and methods from another class. For instance, a class called `Dog` might *inherit* from a class called `Animal`. So a `Dog` does everything an `Animal` does (e.g. `eat` and `sleep`), but it also has its own unique attributes and methods (e.g. `bark`).
Inheritance facilitates code reuse, but also specialization. For example, all `Animals` can `eat` and `move`, but a `Dog` "walks" while a `Mockingbird` "flies". When defining a `Dog`, we can inherit the default "eating" functionality, but provide a custom moving strategy.

Info

Polymorphism
Polymorphism is the ability to use a single interface to represent different data types. In Python, this means that a method can do different things based on the object it is acting upon. For example, a `speak` method might produce different sounds depending on whether it is called on a `Dog` or a `Cat`.
In another example, a car shop might have a `repair` method that can fix any `Car`, regardless of it being a `Toyota` or a `Ford`. As long as the object has all the methods that a `Car` has, the `repair` method will work.

Info

Duck Typing

If it quacks like a duck, it's a duck.

When we write a function:

```
def quack(obj):
    obj.quack()
```

We are not placing any restrictions on the type of the object. Only when the function is called will we know if the object has a quack method. This is called duck typing. The argument to this function does not need to be a duck, it just needs to have a `quack` method.

Even if we place type hints into the mix, Python will not enforce them.

2 Inheritance

Inheritance enables a new class (child) to acquire the properties and behaviors of an existing class (parent).

```
class Bird:
    number_of_wings = 2
    def fly(self):
        return "<generic flying noise>"

class Eagle(Bird):
    def hunt(self):
        return "Hunting for prey"

e = Eagle()
print(e.fly())
print(e.hunt())
print(e.number_of_wings)
```

Output

```
<generic flying noise>
Hunting for prey
2
```

We did not have to define the `fly` method in the `Eagle` class, an `Eagle` *is a* `Bird` and therefore it can do everything a `Bird` can do. On top of it, an `Eagle` can also `hunt`.

Advice

Remember the "is-a" relationship: an `Eagle` is a `Bird`.

2.1 Specialization: Overriding Methods

When a method is defined in a child class with the same name as a method in the parent class, the child class method will override the parent class method.

```
class Bird:
    def fly(self):
        return "<generic flying noise>"

class Eagle(Bird):
    def fly(self):
        return "<soaring high>"

e = Eagle()
print(e.fly())
```

Output

```
<soaring high>
```

Advanced

Initialization in inherited classes

When a class is instantiated, the `__init__` method is called. If the child class has an `__init__` method, it will override the parent class's `__init__` method. If you want to call the parent class's `__init__` method, you can do so using the `super()` function.

```
class Bird:
  def __init__(self):
    print("A bird is born")
class Eagle(Bird):
  def __init__(self):
    super().__init__()
    print("An eagle is born")
e = Eagle()
```

Output

```
A bird is born
An eagle is born
```

Try removing the call to `super()`, only the second message will be printed.

Advanced

Multiple Inheritance

Python supports multiple inheritance, which means a class can inherit from more than one parent class.

```
class Flier:
    def fly(self):
        return "Flying"
class Eater:
    def eat(self):
        return "Eating"
class Bird(Flier, Eater):
    def chirp(self):
        return "Chirping"
b = Bird()
print(b.fly())
print(b.eat())
print(b.chirp())
```

Output

```
Flying
Eating
Chirping
```

Warning

OOP gets real messy real quick when one abuses these advanced features. It is usually better to keep things simple and use inheritance sparingly. As a matter of fact, I do not remember the last time I had to use multiple inheritance in a project.

Advice

Inheritance in the wild

In the obiquitous ML library `PyTorch`, all neural network layers inherit from a base class called `nn.Module`. This means that all layers have the same interface, and we can use them interchangeably in our code.

For instance, we can implement a linear layer (already available in `PyTorch` as `nn.Linear`) as follows:

```python
import torch
import torch.nn as nn

class Linear(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(out_features, in_features))
        self.bias = nn.Parameter(torch.randn(out_features))

    def forward(self, x):
        return x @ self.weight.t() + self.bias
```

All layers in PyTorch, as well as those defined by the user, can be used in the same way, as long as they inherit from `nn.Module` and implement the `forward` method. PyTorch uses this fact to create code that can work with any neural network. For instance, the backpropagation mechanism (called `autograd`) works on any model that is a `nn.Module`, regardless of its specific architecture. The training loop also does not care about the specific architecture of the network, it just calls the `forward` method of the model. This enables the same PyTorch code to work with a simple MLP classifier, a ResNet, or the latest version of the GPT LLM architecture.

3 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. This is useful when you have a collection of objects and you want to perform the same operation on all of them.

For instance, we can make a functionality generic for all `Birds` if we just rely on the `fly` method when writing it. Since all classes that inherit from `Birds` have a `fly` method, our function will work with all of them. Let me give you an example:

```python
class Logger:
      def log(self, message):
          print(f"Log: {message}")
class CheerfulLogger(Logger):
      def log(self, message):
          print(f"Good morning!, there is a message: {message}")
class TimestampLogger(Logger):
      def log(self, message):
          from datetime import datetime
          print(f"{datetime.now()}: {message}")

def process_list(list, logger):
      for item in list:
          logger.log(item)
list = ["An event occurred"]
process_list(list, Logger())
process_list(list, CheerfulLogger())
process_list(list, TimestampLogger())
```

Output

```
Log: An event occurred
Good morning!, there is a message: An event occurred
2026-03-17 11:38:44.784148: An event occurred
```

We were able to customize the behavior of the `process_list` method by passing different objects to the `process_list` function, without having to modify the function itself. This is polymorphism in action. In the example above, in the `process_list` we do not care about the specific details of how the logger is implemented.

Advice

Polymorphism and Duck Typing
In Python, the concept of polymorphism is closely tied to duck typing. If an object has the necessary methods, it can be used in a polymorphic way. This powerful feature is part of the reason why Type hints are not enforced in Python.
For example, this is an instance of Duck typing and not polymorphism:

```
def print_all(l):
   for item in l:
      print(item, end=" ")
   print()
import numpy as np
import pandas as pd
print_all([1,2,3])
print_all(np.array([1,2,3]))
print_all(pd.Series([1,2,3]))
```

Output

```
1 2 3
1 2 3
1 2 3
```

This function will work for anything that can be iterated over, but it does not rely on all inputs inheriting from a common superclass.

Advice

Many ways of achieving polymorphism
It can be dizzying to think about all the ways one can achieve the same functionality in a language like Python. For instance, the `process_list` function in the example above did not actually required the loggers to inherit from `Logger` due to duck typing. But we could have also just skipped writing any classes and used a function that takes a function as an argument to use as a logger. We could had also written the function so that it takes a string representing a logging strategy, and `if-then-else` our way to the right logger. I can think of at least another 4 ways of doing it using more advanced Python features.
Knowing when to use each approach/tool is a matter of experience and intuition. We should strive to reduce complexity in our code and make it as easy to understand as possible. The best way to gather this intuition is to mess it up a few times and learn from it.
For instance, you might decide to use a class hierarchy in your design to solve a particular problem, only to realize deep down the line that the codebase is now utterly unreadable. You learn about why and how this happened, and the next time you are faced with a similar problem, you will have an intuition of what *not* to do.
The true power of a senior programmer is being an insanely efficient troubleshooter, just due to the sheer amount of mishaps they have had to deal with in the past. In Spanish, we say "Más sabe el diablo por viejo que por diablo".

3.1 Calling the base class methods: `super()`

We can use the `self` member from within methods in a class to access other properties of the current instance. But what if we want to access a method from the parent class? Let me show you a situation where this is useful:

```
class Kettle:
    def use(self):
        return "Boiling"
class ElectricKettle(Kettle):
    def use(self):
        action = "Boiling"
        return action + " with electricity"
```

In the example above, we are repeating ourselves, and we are assuming that the action of a `Kettle` is always `Boiling`. It would be great if we could call the `use` method from the parent class and then add the `with electricity` part. We can do this with the `super()` function.

```
class Kettle:
    def use(self):
        return "Boiling"
class ElectricKettle(Kettle):
    def use(self):
        action = super().use()
        return action + " with electricity"
```

Now if we change the `use` method in the `Kettle` class, the `ElectricKettle` class will automatically reflect the change.

4 Abstract Classes

Abstract classes serve as blueprints for other classes. They define methods that *must* be implemented in any subclass. Python's `abc` module provides the tools to create abstract classes. Sometimes, we want base classes to define an interface, or API, without providing a concrete (default) implementation. For instance, all `Shapes` should have a method called `area()`, but computing the area of a `Shape` itself does not really make sense.

Example:

```
from abc import ABC, abstractmethod

class Shape(ABC):

    @abstractmethod
    def area(self):
        """Return the area of the shape."""
        pass # All abstract methods must have a body that just passes


class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

rect = Rectangle(4, 5)
print("Rectangle area:", rect.area())
# shape = Shape() # This will raise an error
```

Info

@ decorators
The `@abstractmethod` line above the method definition is called a "decorator". We will see about them soon. In essence, a decorator is a function that takes another function and extends its behavior without modifying it. In this case, the `abstractmethod` decorator marks the method as abstract, meaning it must be implemented in any subclass.

Advice

Abstract classes cannot be instantiated directly. They are meant to be subclassed, ensuring that all abstract methods are implemented. Try to copy the code above, but delete the implementation of `area` from the `Rectangle`.
As usually in Python, these are just tools for the developer. Duck typing means we do not strictly need an abstract class called `Shape`. But if it exists, the developer that writes the `Triangle` class will know that they need to implement an `area` method.

5 Exercises

5.1 Orbital Mechanics

Info

Theory: Newton's Laws of Motion and Gravitational Force
Newton's second law of motion can be expressed as:

$$\vec{F} = m\vec{a} = m\ddot{\vec{q}} = m\frac{d^2\vec{q}}{dt^2}$$

where F is the force acting on an object located at position $\vec{q}$, m is the mass of the object, and $\vec{a}$ is the acceleration of the object. If the particle is subject to a gravitational force (say from a star), the force can be expressed as:

$$\vec{F} = -\frac{GMm}{r^3}\vec{q}$$

where G is the gravitational constant, M is the mass of the star, and r is the distance from the star to the planet. We are assuming that the star is at the origin (0, 0) and the planet is at position $\vec{q} = (x, y)$.
This is a second-order differential equation, but we can rewrite it as a system of first-order differential equations:

$$\frac{d\vec{q}}{dt} = \vec{v} \tag{1}$$
$$\frac{d\vec{v}}{dt} = -\frac{GM}{r^3}\vec{q} \tag{2}$$

where $\vec{q} = (x, y)$ is the position of the planet, and $\vec{v} = (v_x, v_y)$ is the velocity of the planet.

Advice

- Interestingly, the final equations are independent of the mass of the planet.
- The first equation, in vectorial form, represents two equations (one for each direction). In 2D, you must read it as: $\frac{dx}{dt} = v_x$ and $\frac{dy}{dt} = v_y$.

Info

Theory: Euler's Method for Numerical Integration
Euler's method is a simple numerical method to approximate solutions to differential equations. Given a differential equation of the form:

$$\frac{d\vec{y}}{dt} = \vec{f}(t, \vec{y})$$

where $\vec{y}$ is a vector of variables, and $\vec{f}$ is a function that returns the derivative of $\vec{y}$ at time t, Euler's method approximates the solution by:

$$\vec{y}_{n+1} = \vec{y}_n + \vec{f}(t_n, \vec{y}_n) \cdot dt$$

where $\vec{y}_n$ is the state of the system at time t_n, and dt is the time step. The smaller the time step, the more accurate the approximation.
In our case, we have two equations of this form, one for the position and one for the velocity of the planet.

Advice

Although it is simple, Euler's method is not very accurate to integrate orbital mechanics. We will use it here for simplicity, but in practice, you would use a more sophisticated method like Runge-Kutta 4 or velocity-Verlet.

Goal

Let's create a simple that models the motion of a planet around a star and plot its trajectory using OOP concepts.
We will define some classes for it (we will go step by step):

- A base class `DifferentialEquation` with a method `f(t, state)` that represents the system of equations governing motion.
- Implement a subclass `OrbitalMotion` that represents Newton's equations of motion under gravity.
- Create a base class `Integrator` with a method `step(state, dt)`.
- Implement a subclass `EulerIntegrator` that approximates solutions in the next step using the Euler method.

After completing the exercises, you should be able to run:

```
eq = OrbitalMotion(G=1.0, M=1.0)
dt = 0.001
integrator = EulerIntegrator(eq, dt)
t, state = 0, [1.0, 0.0, 0.0, 1.0]  # (x, y, vx, vy)
for _ in range(100):
    t, state = integrator.step(t, state)
    print(f"t={t:.2f}, x={state[0]:.3f}, y={state[1]:.3f}")
```

Advice

We are perhaps overdoing it with the OOP here. For instance, we do not strictly need base or abstract classes for this, and the `DifferentialEquation` class could be a lambda function. But this is a good exercise to practice OOP concepts.
From your final code, you will also be able to judge how well the design we chose fits the problem. I also intend to get you exposed to the software design process.

Milestone

Lets start by starting a new project. Create a new directory structure for your project, you can copy paste the `environment.yml`, etc files from a previous lesson and modify as needed. You should have a directory structure like this:

```
project/
| - environment.yml
| - setup.py
| - src/
       |- orbitals/
                  |- __init__.py
                  \- differential_equation.py
```

For the moment, both python scripts might be empty. Open all files in VSCode and save them (even if they are empty). Make sure the files are appear in the correct places (move around and use `ls`).

Advice

- In Windows PowerShell, you can use the `tree /F` command to see the directory structure similarly to the above output. You can also try `Get-ChildItem ..`
- In OSX, you can use `ls -R`, `find .`, or `tree`.

Milestone

Create and activate a virtual environment for this project with the dependencies.

hint

- Create a new environment with `conda env create -f environment.yml -n orbitals`
- Activate the environment with `conda activate orbitals`

Milestone

Make the project a `git` repo.

hint

- Run `git init` inside the project directory.
- Add the files to the repo with `git add [list of files]`
- Commit the files with `git commit -m "Initial commit"`

Milestone

Define the `DifferentialEquation` class
Import the `diferential_equation` module in the `src/orbitals/__init__.py` file:

```
from .differential_equation import *
```

Create a base class that represents a system of equations. In the file `src/orbitals/differential_equation.py`:

```
from abc import ABC, abstractmethod
class DifferentialEquation(ABC):

    @abstractmethod
    def __call__(self, t, state):
       pass
```

Our code does not have anything "runnable" yet, so lets wait a bit before installing it.

hint

- The dunder method `__call__` enables the parenthesis operator, so we can call the object as if it were a function.

Milestone

Implement `OrbitalMotion`
Inside `src/orbitals/differential_equation.py`, create a subclass `OrbitalMotion` that implements `f(t, state)` to represent the equations of motion for a planet around a star (as described in the theory section). There are four equations to implement, two for the position and two for the velocity. The two for position are trivial (just the velocity).
Let me give you a hint.

```
import numpy as np

class OrbitalMotion(DifferentialEquation):
    def __init__(self, G, M):
        # Your code here

    def __call__(self, t, state):
        """Return the derivatives of the state vector.
        The state represents the position and velocity of the planet in
        2D space, so state = [x, y, vx, vy]

        Parameters:
              t (float): Time
              state (list): State vector [x, y, vx, vy]

        Returns:
              list: Derivatives [vx, vy, ax, ay]
        """
        x, y, vx, vy = state
        # Your code here
        return [vx, vy, ax, ay]
```

Milestone

Adapt the `setup.py` file if needed. For instance:

```
from setuptools import setup, find_packages

setup(
    name="orbitals",
    version="0.1",
    packages=find_packages("src"),
    package_dir={"": "src"},
)
```

Our package does not have any CLI tools, just some classes that we can import. Install it to your current env (make sure you are in the project directory and with the environment activated):

```
pip install -e .
```

hint

The `-e` flag installs the package in "editable" mode, meaning that you can modify the code and the changes will be reflected in the installed package, without having to reinstall it.

Milestone

Make a new commit with the changes thus far. Start by running `git status` to see what files have changed, then add them with `git add [list of files]` and commit with `git commit -m "Some message"`.

Milestone

Lets put an example script at the root of the project to test the `OrbitalMotion` class. Create a file called `example.py` with the following content:

```
from orbitals import OrbitalMotion

eq = OrbitalMotion(G=1.0, M=1.0)
t, state = 0, [1.0, 0.0, 0.0, 1.0]  # (x, y, vx, vy)
# The __call__ method is called when we use the object as a function
derivatives = eq(t, state)
print(derivatives)
```

Try to run it. If everything went well thus far, you should get some numbers printed.

Milestone

Define the `Integrator` base class

Lets repeat this process for the `Integrator` side. Create a new file `src/orbitals/integrator.py` and define the `Integrator` class in it. Create an abstract class `Integrator` with a method `step(t, state, dt)` that must be implemented by subclasses.

```python
from abc import ABC, abstractmethod
class Integrator(ABC):
    def __init__(self, equation, dt):
        self.equation = equation
        self.dt = dt
    @abstractmethod
    def step(self, t, state, dt):
        pass
```

Import the `Integrator` class in the `src/orbitals/__init__.py` file:

```python
from .integrator import *
```

Milestone

Implement Euler's Method

Create a subclass `EulerIntegrator` that inherits from `Integrator` and implements the step method using Euler's rule: $\text{state}_{n+1} = \text{state}_n + f(t_n, \text{state}_n) \cdot dt$ where state_n is the state of the system at time t_n, and dt is the time step. Note that the state can have an arbitrary number of variables.

```python
class EulerIntegrator(Integrator):
    def step(self, t, state):
        derivatives = self.equation(t, state)
        new_state = [s + ds * self.dt for s, ds in zip(state, derivatives)]
        return t + self.dt, new_state
```

Milestone

Test the Euler integrator

Write a test for the integrator. We can use a simple eq. for which we know the solution, e.g. $\frac{df(x,t)}{dt} = 5x$ with $f(0) = 1$.
After one integration step, the solution should be $f(dt) = f(0) + 5 \cdot f(0) \cdot dt = 1 + 5 \cdot dt$.

```
from orbitals import DifferentialEquation, EulerIntegrator
def test_euler():
    factor = 5.0
    initial = 1.0
    dt = 0.01

    class Eq(DifferentialEquation):
        def __call__(self, t, state):
            return [factor * state[0]]
    eq = Eq()
    # Alternative:
    # eq = lambda t, state: [factor * state[0]]
    i = EulerIntegrator(eq, dt=dt)
    t, state = 0, [initial]
    assert i.step(t, state) == (dt, [initial + factor * initial * dt])
```

Place this test in a file called `test/test_integrator.py` in the root of the project. Make sure it passes by running `pytest`.

Advice

- I am trying to convey here how you can test your code in projects where the output of your modules is not easily verifiable. This happens a lot in Physics and Math. In these instances, you must look for simplified cases, self-consistency checks, or known solutions to test your code.
- Perhaps you will need to `pip install` your project again.

Milestone

Plot the trajectory of a planet

Lets create an animation of the planet motion.

Create a new notebook at `example/example.ipynb`. In this script, we will use the `OrbitalMotion` class and the `EulerIntegrator` to simulate the motion of a planet around a star. We will then plot the trajectory of the planet.

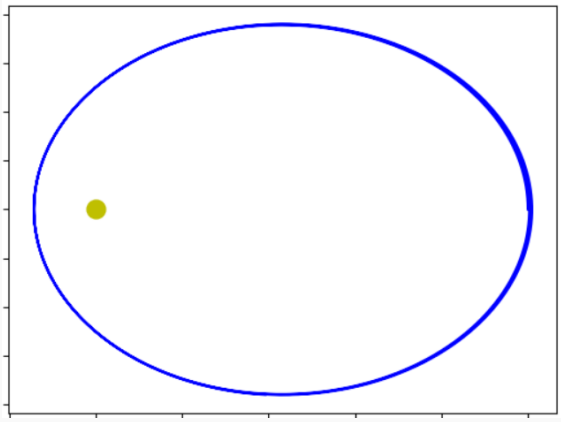
Create a new cell and add the following code:

```
import matplotlib.pyplot as plt
import IPython.display as display
import numpy as np
from orbitals import OrbitalMotion, EulerIntegrator

# Initialize orbital motion and integrator
eq = OrbitalMotion(G=1.0, M=1.0)
dt = 0.00001
integrator = EulerIntegrator(eq, dt)
t, state = 0, [1.0, 0.0, 0.0, 0.5]  # (x, y, vx, vy)

fig, ax = plt.subplots()
line, = ax.plot([], [], 'b-')
ax.plot(0, 0, 'yo', markersize=12, label="Star")  # 'yo' = yellow circle
nsteps = 1000000
trajectory = np.zeros((nsteps, 2))
for i in range(nsteps):
    t, state = integrator.step(t, state)
    trajectory[i] = state[:2]
    if i % 10000 == 0:
        line.set_xdata(trajectory[:i, 0])
        line.set_ydata(trajectory[:i, 1])
        ax.relim()
        ax.autoscale_view()
        display.clear_output(wait=True)  # Clear previous frame
        display.display(fig)  # Show updated figure
```

When running it (perhaps you need to install some dependencies), you should see a plot of the planet orbiting the star.



Advanced milestone

Implement another integrator
If you feel comfortable, try implementing another method like Runge-Kutta 4 (RK4). This method is more accurate and follows:

```
class RK4Integrator(Integrator):
    def step(self, t, state):
        f = self.equation
        dt = self.dt
        k1 = f(t, state)
        k2 = f(t + dt/2, [s + k1_i*dt/2 for s, k1_i in zip(state, k1)])
        k3 = f(t + dt/2, [s + k2_i*dt/2 for s, k2_i in zip(state, k2)])
        k4 = f(t + dt, [s + k3_i*dt for s, k3_i in zip(state, k3)])
        new_state = [s + (dt/6)*(k1_i + 2*k2_i + 2*k3_i + k4_i)
                     for s, k1_i, k2_i, k3_i, k4_i in zip(state, k1, k2, k3, k4)]
        return t + dt, new_state
```

hint

- You don't need to fully understand RK4, focus on how to override methods in a subclass.

Advanced milestone

Use polymorphism in a function
Write a function that can run any `Integrator` on any `DifferentialEquation`.

```
def solve(eq: DifferentialEquation,
          integrator: Integrator,
          t0: float,
          state0: List,
          dt: float,
          steps: int):
    # Your code here
    return results
```

Test it with both Euler and RK4 integrators!

Advanced milestone

Add a second sun
The differential equations in `OrbitalMotion` assume that one sun is located at position $(0, 0)$. Add a second sun at position $(2, 0)$. For that, we have to modify the force acting on the planet:

$$\vec{F} = -\frac{GMm}{r^3}\vec{q} - \frac{GMm}{r'^3}\vec{q'}$$

where $\vec{q'} := \vec{q} - (2, 0)$ is the vector from the planet to the second sun, with magnitude r'.

Advanced

- Can you find a stable orbit for the planet?