Object-Oriented Programming for Python with Turtle and Pygame

Updated February 11, 2026

Contents

Prologue

This guide offers a concise overview of object-oriented programming (OOP) in Python. It also introduces Turtle graphics and Pygame for creating visual programs and games. Before diving into these topics, it's highly recommended to complete the Intro to Python guide.

1 Introduction to Object-Oriented Programming (OOP)

1.1 Why OOP?

Imagine you're building an animal simulator game. You need dogs, cats, and birds — each with its own name, age, favorite food, and behaviors like eating or playing. Without OOP, you might write separate variables and functions for every single animal, and the code would get messy fast.

Object-oriented programming (OOP) is a way of organizing code by grouping related data and actions together into reusable building blocks. It's one of the most widely used approaches in professional software development, and it's how Python's own built-in types (strings, lists, dictionaries) are built under the hood.

1.2 The Cookie Cutter Analogy

The core idea of OOP comes down to two things: **classes** and **objects**.

Think of a **class** as a cookie cutter. It defines the shape — what every cookie will look like. But the cookie cutter itself isn't a cookie. An **object** is an actual cookie made from that cutter. You

can make as many cookies (objects) as you want from the same cutter (class), and each cookie can be decorated differently (have different data).

In Python terms: a class defines *what* something is (its data and behaviors), and an object is a *specific instance* of that class with actual values.

1.3 A First Taste

Here's a tiny example to show what this looks like in code. Don't worry about understanding every detail yet — we'll break it all down in Chapter 2.

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

    def greet(self):
        print(f"Hi! I'm {self.name} the {self.animal_type}.")

# Create two pet objects from the same class
buddy = Pet("Buddy", "dog")
whiskers = Pet("Whiskers", "cat")

buddy.greet()
whiskers.greet()
```

```
Hi! I'm Buddy the dog.
Hi! I'm Whiskers the cat.
```

Notice how we wrote the `Pet` class once, then created two different pets from it — each with its own name and type. That's the power of OOP: **define once, use many times**.

1.4 What's Ahead

OOP has a few new ideas to learn, so we'll take them one step at a time:

- **Chapter 2**: Classes and objects — the building blocks
- **Chapter 3**: OOP principles — inheritance, polymorphism, encapsulation, and abstraction
- **Chapter 4**: Importing code across files
- **Chapter 5**: Turtle graphics — drawing with OOP
- **Chapter 6**: Pygame — building games with OOP

Let's get started!

2 Classes and Objects

A **class** is a blueprint for creating objects. An **object** is a specific thing created from that blueprint. To use OOP, you first *define* a class (the blueprint), then *instantiate* objects from it (the things). This is similar to how you define a function first, then call it — but at a larger scale.

A class starts with a header:

```
class ClassName:
```

Unlike variable or function names, class names capitalize each word and do **not** use underscores. For example: `Pet`, `SpaceShip`, `GameCharacter`.

A class can contain **variables** (data) and **methods** (functions that belong to the class). Let's build one up piece by piece.

2.1 The `__init__()` Method

The `__init__()` method (also called the **constructor**) is a special function that runs automatically when you create a new object from a class. It sets up the object's starting data. A class should have exactly one `__init__()` method.

```
def __init__(self, name, age):
```

The first parameter, `self`, refers to the specific object being created. Python fills it in automatically — you never pass it yourself. The remaining parameters (`name`, `age`) are values you provide when creating the object.

2.2 Instance Variables

Instance variables are data that belong to a specific object. Each object gets its own copy. You create them inside `__init__()` using the `self.` prefix:

```
class Pet:
    def __init__(self, name, age, animal_type, favorite_food):
        self.name = name
        self.age = age
        self.animal_type = animal_type
        self.favorite_food = favorite_food
        print(f"{self.name} is a {self.age} year old {self.animal_type} who loves
            {self.favorite_food}!")
```

Here, `self.name`, `self.age`, `self.animal_type`, and `self.favorite_food` are all instance variables. When you create different Pet objects, each one will have its own values for these variables.

2.3 Class Variables

Class variables are shared by *all* objects created from the class. They're defined outside of any method, directly inside the class body.

```
class Pet:
    pet_count = 0  # Class variable --- shared by all Pet objects

    def __init__(self, name, age, animal_type, favorite_food):
        self.name = name                # Instance variable --- unique to each Pet
        self.age = age
        self.animal_type = animal_type
        self.favorite_food = favorite_food

        print(f"{self.name} is a {self.age} year old {self.animal_type} who loves
            {self.favorite_food}!")
        Pet.pet_count += 1              # Update the shared count
```

Notice the difference: instance variables use **`self.`** (unique to each object), while class variables use the **class name** as a prefix (e.g., `Pet.pet_count`).

2.4 Creating Objects (Instantiation)

To create an object, call the class name like a function, passing the values that `__init__()` expects (minus `self`):

```
buddy = Pet("Buddy", 3, "dog", "chicken")
whiskers = Pet("Whiskers", 5, "cat", "tuna")

print(f"Total pets: {Pet.pet_count}")
print(f"{buddy.name} is {buddy.age} years old")
print(f"{whiskers.name} is {whiskers.age} years old")
```

```
Buddy is a 3 year old dog who loves chicken!
Whiskers is a 5 year old cat who loves tuna!
Total pets: 2
Buddy is 3 years old
Whiskers is 5 years old
```

Creating an object is formally called **instantiation** — you're creating an *instance* of the class. Each instance has its own copy of the instance variables but shares the class variable `pet_count`.

Outside the class, you access instance variables with the **object name** (e.g., `buddy.name`) and class variables with the **class name** (e.g., `Pet.pet_count`).

2.5 Instance Methods

Instance methods are functions defined inside a class that operate on a specific object. They always take `self` as their first parameter, which gives them access to that object's instance variables.

While instance variables represent the **characteristics** of an object, instance methods represent its **behaviors**.

```
class Pet:
    pet_count = 0

    def __init__(self, name, age, animal_type, favorite_food):
        self.name = name
        self.age = age
        self.animal_type = animal_type
        self.favorite_food = favorite_food
        Pet.pet_count += 1

    def play(self):
        if self.age < 5:
            print(f"{self.name} is playing happily!")
        else:
            print(f"{self.name} is taking a nap.")

    def eat(self, food):
        if food == self.favorite_food:
            print(f"{self.name} loves eating {food}!")
        else:
            print(f"{self.name} doesn't really like {food}.")

buddy = Pet("Buddy", 3, "dog", "chicken")
buddy.play()
buddy.eat("chicken")
buddy.eat("broccoli")
```

```
Buddy is playing happily!
Buddy loves eating chicken!
Buddy doesn't really like broccoli.
```

The `play` method takes no extra parameters (just `self`), while `eat` takes a `food` parameter. Both use `self.` to access the object's own data. When you call `buddy.play()`, Python automatically passes `buddy` as `self`.

2.6 The `__str__()` Method

When you `print()` an object, Python doesn't know how to display it nicely by default. The special `__str__()` method lets you define what string should appear:

```
class Pet:
    def __init__(self, name, age, animal_type):
        self.name = name
        self.age = age
        self.animal_type = animal_type

    def __str__(self):
        return f"{self.name} the {self.animal_type} (age {self.age})"

buddy = Pet("Buddy", 3, "dog")
print(buddy)
```

```
Buddy the dog (age 3)
```

Without `__str__()`, printing an object would show something unhelpful like `<__main__.Pet object at 0x7f...>`. Defining `__str__()` makes your objects much easier to work with and debug.

2.7 Objects as Variables

Since classes act like custom data types, objects can be used anywhere variables can — including in lists, as function arguments, or as return values:

```python
class Toy:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def play_with(self):
        print(f"Playing with the {self.color} {self.name}!")

    def __str__(self):
        return f"{self.color} {self.name}"

# Create toy objects and put them in a list
teddy = Toy("teddy bear", "brown")
ball = Toy("ball", "red")
robot = Toy("robot", "blue")

toy_box = [teddy, ball, robot]

# Loop through the list and call methods on each object
for toy in toy_box:
    toy.play_with()

print(f"I have {len(toy_box)} toys: {', '.join(str(t) for t in toy_box)}")
```

```
Playing with the brown teddy bear!
Playing with the red ball!
Playing with the blue robot!
I have 3 toys: brown teddy bear, red ball, blue robot
```

Understanding that objects are just like variables is key to using OOP effectively. You'll see this idea everywhere in the Turtle and Pygame chapters.

2.8 Access Reference

Here's a quick reference for how to access class members from different places:

Try it yourself! Create a `Student` class with a name, grade, and a method `introduce()` that prints a greeting. Then create three students and store them in a list.

Where you are	Class variable	Instance variable
Inside an instance method	`ClassName.var`	`self.var`
Outside the class	`ClassName.var`	`object_name.var`

Table 1: Access reference for class and instance variables. Assume `ClassName` is the name of the class, and `object_name` is the name of the object created from `ClassName`.

3 OOP Principles

There are four core principles of OOP. These principles help you write code that's organized, reusable, and easier to manage as your programs grow. They build on the class and object concepts from Chapter 2.

3.1 Inheritance

Inheritance lets a class borrow properties and methods from another class, so you don't have to write the same code twice. The class that inherits is called the **child class**, and the class being inherited from is called the **parent class**.

In Python, you put the parent class name in parentheses after the child class name. The **`super()`** function calls methods from the parent class — most commonly, the parent's `__init__()`:

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

    def speak(self):
        print(f"{self.name} says {self.sound}!")

    def __str__(self):
        return self.name

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, "Woof")  # Call parent's __init__
        self.breed = breed

    def fetch(self):
        print(f"{self.name} the {self.breed} fetches the ball!")

class Cat(Animal):
    def __init__(self, name, is_indoor):
        super().__init__(name, "Meow")
        self.is_indoor = is_indoor

    def purr(self):
        print(f"{self.name} is purring...")

buddy = Dog("Buddy", "Golden Retriever")
whiskers = Cat("Whiskers", True)
```

```
30
31 buddy.speak()       # Inherited from Animal
32 buddy.fetch()       # Dog's own method
33
34 whiskers.speak()    # Inherited from Animal
35 whiskers.purr()     # Cat's own method
```

```
Buddy says Woof!
Buddy the Golden Retriever fetches the ball!
Whiskers says Meow!
Whiskers is purring...
```

Both `Dog` and `Cat` inherit the `speak()` method from `Animal`, so we only had to write it once. Each child class then adds its own unique methods (`fetch`, `purr`).

3.1.1 Method Overriding

A child class can **override** (replace) a method it inherited from the parent by defining a method with the same name. This lets child classes customize inherited behavior:

```
1  class Animal:
2      def __init__(self, name):
3          self.name = name
4
5      def speak(self):
6          print(f"{self.name} makes a sound.")
7
8  class Dog(Animal):
9      def speak(self):  # Overrides Animal's speak()
10         print(f"{self.name} says Woof!")
11
12 class Cat(Animal):
13     def speak(self):  # Overrides Animal's speak()
14         print(f"{self.name} says Meow!")
15
16 generic = Animal("Some Animal")
17 buddy = Dog("Buddy")
18 whiskers = Cat("Whiskers")
19
20 generic.speak()
21 buddy.speak()
22 whiskers.speak()
```

```
Some Animal makes a sound.
Buddy says Woof!
Whiskers says Meow!
```

Even though `Dog` and `Cat` both inherit from `Animal`, each provides its own version of `speak()`. This is method overriding — and it leads directly into our next principle.

3.2 Polymorphism

Polymorphism means "many forms." It's the idea that objects of different classes can be treated the same way if they share a common interface (like a method name). Combined with inheritance, this is very powerful:

```
# Using the Animal, Dog, Cat classes from above

animals = [Dog("Buddy"), Cat("Whiskers"), Dog("Rex"), Cat("Luna")]

for animal in animals:
    animal.speak()  # Same method call, different behavior!
```

```
Buddy says Woof!
Whiskers says Meow!
Rex says Woof!
Luna says Meow!
```

The loop calls `speak()` on every animal without knowing or caring whether it's a Dog or a Cat. Each object responds in its own way. This is polymorphism — the same method name produces different behavior depending on the object's class.

This pattern is especially useful in games: you might have a list of `Enemy` objects where some are `Zombie`, some are `Ghost`, and some are `Dragon`, but your game loop just calls `enemy.update()` on each one and lets polymorphism handle the rest.

3.3 Encapsulation

Encapsulation bundles data and methods into a single unit (a class) and controls access to some parts. The idea is to hide internal details so that outside code only interacts with the object through its public methods — like how you use a microwave by pressing buttons without needing to know how the circuits inside work.

In Python, encapsulation is achieved by convention using **underscores** in names:

- A single underscore `_var` signals: *"this is meant to be internal — please don't touch it directly."* Python won't stop you, but other programmers will know it's not part of the public interface.
- A double underscore `__var` triggers **name mangling**: Python renames the variable to `_ClassName__var`, making it harder to access accidentally from outside. This helps avoid name clashes in child classes.

Neither is truly "private" the way some other languages enforce it — in Python, these are **conventions** rather than strict rules.

```
class PiggyBank:
    def __init__(self):
        self._coins = 0              # Internal (by convention)
```

```
        self.__secret_code = 1234     # Name-mangled (harder to access)

    def add_coin(self):
        self._coins += 1
        print(f"Added a coin! Total: {self._coins}")

    def get_coins(self):
        return self._coins

    def open_bank(self, code):
        if code == self.__secret_code:
            print(f"Bank opened! You have {self._coins} coins!")
        else:
            print("Wrong code! Bank stays locked!")

bank = PiggyBank()
bank.add_coin()
bank.add_coin()
bank.add_coin()
print(f"Coins: {bank.get_coins()}")
bank.open_bank(1234)
bank.open_bank(9999)
```

```
Added a coin! Total: 1
Added a coin! Total: 2
Added a coin! Total: 3
Coins: 3
Bank opened! You have 3 coins!
Wrong code! Bank stays locked!
```

Instead of changing `_coins` directly, outside code uses `add_coin()` and `get_coins()`. This lets the class control *how* its data is accessed and modified.

3.4 Abstraction

Abstraction means hiding complex details and showing only the essential interface. While encapsulation hides *data*, abstraction hides *complexity*. You use something through a simple interface without needing to understand what happens inside.

In Python, you can create **abstract classes** using the `abc` module. An abstract class defines methods that every child class *must* implement, without providing the implementation itself:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass          # No implementation --- child classes must provide one

    @abstractmethod
    def describe(self):
        pass

class Rectangle(Shape):
```

```
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

    def describe(self):
        print(f"Rectangle: {self.width} x {self.height}, area = {self.area()}")

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def describe(self):
        print(f"Circle: radius {self.radius}, area = {self.area():.2f}")

# shape = Shape()  # This would cause an error --- you can't create an abstract class!

shapes = [Rectangle(4, 5), Circle(3), Rectangle(10, 2)]
for shape in shapes:
    shape.describe()
```

```
Rectangle: 4 x 5, area = 20
Circle: radius 3, area = 28.27
Rectangle: 10 x 2, area = 20
```

The `Shape` class says "every shape must have an `area()` and `describe()` method" but doesn't say *how*. Each child class fills in the details. This also demonstrates polymorphism — we loop through different shapes and call `describe()` on each one.

For beginners, the key takeaway is: abstraction means **providing simple interfaces that hide complexity**. You'll see this in action when using Turtle and Pygame — you call methods like `forward()` or `draw()` without needing to know how pixels are rendered on screen.

Try it yourself! Create a `Vehicle` parent class with a `move()` method, then create `Car` and `Bicycle` child classes that override `move()` with their own messages. Put them in a list and loop through it.

4 `import` and Cross-File Programming

When Python programs grow larger, keeping everything in one file becomes hard to manage. The **`import`** statement lets you use code from other Python files and libraries, which is especially useful when you have many classes.

4.1 Import Methods

There are several ways to import modules in Python:

```
# Method 1: Import the entire module
import math
print(math.sqrt(25))  # 5.0
print(math.pi)        # 3.14159...

# Method 2: Import specific items
from math import sqrt, pi
print(sqrt(25))       # 5.0
print(pi)             # 3.14159...

# Method 3: Import with a nickname (alias)
import random as r
print(r.randint(1, 10))

# Method 4: Import everything (not recommended --- makes it unclear where things
    come from)
from math import *
```

For a detailed reference on built-in libraries like `random`, `math`, and `time`, see Chapter 9 of the Intro to Python guide.

4.2 Importing Your Own Files

This is where imports become essential for OOP. When you have classes in separate files, you import them just like libraries — using the filename without the `.py` extension.

Imagine you have two files in the same folder:

`pet.py` — contains the Pet class:

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

    def speak(self):
        print(f"{self.name} the {self.animal_type} says hello!")

    def __str__(self):
        return f"{self.name} the {self.animal_type}"
```

`main.py` — your main program:

```
from pet import Pet

buddy = Pet("Buddy", "dog")
buddy.speak()
print(buddy)
```

Output:

```
Buddy the dog says hello!
Buddy the dog
```

You can also import multiple classes from the same file:

```
# If pet.py also contained a Toy class:
from pet import Pet, Toy
```

Or import the whole file as a module:

```
import pet

buddy = pet.Pet("Buddy", "dog")
```

4.2.1 Why Organize Code This Way?

As your programs grow, separating classes into their own files keeps things organized. A typical project structure might look like:

```
my_game/
    main.py          <- runs the game
    player.py        <- Player class
    enemy.py         <- Enemy class
    item.py          <- Item class
```

Each file focuses on one class (or a group of related classes), and `main.py` imports what it needs. This makes code easier to read, debug, and work on with others.

4.3 Documentation

Every good Python library comes with **documentation** that explains how to use its classes and functions. Reading documentation is an essential skill.

Key places to find Python documentation:

- Official Python docs: docs.python.org
- Library-specific websites (e.g., pygame.org)
- The `help()` function in the Python interpreter

```
import math
help(math.sqrt)    # Shows documentation for sqrt
help(print)         # Shows how to use print
```

4.4 `sys` and `os`

Two useful built-in libraries for working with the file system:

`sys` provides system-specific information:

```
import sys
print("Python version:", sys.version)
print("Platform:", sys.platform)
```

`os` lets you interact with the operating system:

```
import os
print("Current folder:", os.getcwd())
print("Files here:", os.listdir("."))
print("Does 'test.txt' exist?", os.path.exists("test.txt"))
```

These become handy when your program needs to find files, check paths, or work across different computers.

Try it yourself! Create two files: `animal.py` with an `Animal` class, and `main.py` that imports `Animal`, creates a few objects, and prints them.

5 Turtle Graphics

Turtle is a Python library that lets you draw pictures using simple commands. Think of it as a digital pen that follows your instructions to move around and draw lines, shapes, and colorful patterns. Turtle comes built-in with Python, so you don't need to install anything extra.

Turtle is a great bridge between the OOP concepts from Chapters 2-3 and practical programming — every turtle you create is an **object** with its own properties (position, color, direction) and methods (`forward()`, `right()`, `circle()`).

5.1 Basic Turtle Commands

```
import turtle

# Create a turtle object and a screen object
my_turtle = turtle.Turtle()
screen = turtle.Screen()

# Basic movement
my_turtle.forward(100)     # Move forward 100 steps
my_turtle.right(90)        # Turn right 90 degrees
my_turtle.forward(50)      # Move forward 50 steps
my_turtle.left(45)         # Turn left 45 degrees
my_turtle.backward(75)     # Move backward 75 steps

# Keep the window open until clicked
screen.exitonclick()
```

When you run this, a window opens and you see lines appear as the turtle moves!

5.2 Drawing Shapes

Using loops makes drawing shapes easy:

```
import turtle

artist = turtle.Turtle()
screen = turtle.Screen()

# Draw a square (4 sides, 90-degree turns)
for i in range(4):
    artist.forward(100)
    artist.right(90)

# Move without drawing
artist.penup()
artist.goto(150, 0)
artist.pendown()

# Draw a triangle (3 sides, 120-degree turns)
for i in range(3):
    artist.forward(80)
    artist.right(120)

screen.exitonclick()
```

The general pattern is: a regular polygon with `n` sides uses turns of `360 / n` degrees.

5.3 Colors and Pen Control

```
import turtle

painter = turtle.Turtle()
screen = turtle.Screen()
screen.bgcolor("lightblue")

painter.color("red")
painter.pensize(5)
painter.speed(3)        # 1 = slowest, 10 = fast, 0 = instant

# Draw a flower (6 overlapping circles)
for i in range(6):
    painter.circle(50)
    painter.right(60)

# Draw the stem
painter.color("green")
painter.pensize(8)
painter.right(90)
painter.forward(150)

screen.exitonclick()
```

5.3.1 Understanding Colors

There are two ways to specify colors:

Color names — easy to use:

```
painter.color("red")
painter.color("purple")
painter.color("lightblue")
```

RGB tuples — mix Red, Green, and Blue (0–255 each):

```
screen = turtle.Screen()
screen.colormode(255)         # Enable 0--255 mode

painter.color((255, 0, 0))     # Pure red
painter.color((0, 255, 0))     # Pure green
painter.color((0, 0, 255))     # Pure blue
painter.color((255, 255, 0))   # Yellow (red + green)
painter.color((0, 0, 0))       # Black (no color)
painter.color((255, 255, 255)) # White (all colors)
```

RGB colors are important because Pygame (Chapter 6) uses the same system.

5.4 Displaying Text

The `write()` method lets you put text on the screen:

```
import turtle

writer = turtle.Turtle()
writer.hideturtle()       # Hide the turtle shape

writer.goto(0, 100)
writer.write("Hello!", align="center", font=("Arial", 24, "bold"))

writer.goto(0, 50)
writer.color("blue")
writer.write("Centered text", align="center", font=("Arial", 14, "normal"))
```

The `font` takes a tuple of `("FontName", size, "style")`, and `align` can be `"left"`, `"center"`, or `"right"`.

5.5 Keyboard Input

Turtle can respond to key presses, making programs interactive:

```
import turtle

my_turtle = turtle.Turtle()
screen = turtle.Screen()
```

```
5
6  def move_up():
7      my_turtle.setheading(90)
8      my_turtle.forward(20)
9
10 def move_down():
11     my_turtle.setheading(270)
12     my_turtle.forward(20)
13
14 # Connect keys to functions
15 screen.onkey(move_up, "Up")
16 screen.onkey(move_down, "Down")
17 screen.listen()   # Start listening for key presses
18
19 screen.exitonclick()
```

Key rules:

- Call `screen.listen()` to activate keyboard input
- Use lowercase for letter keys: `"a"`, `"b"`, `"c"`
- Arrow keys: `"Up"`, `"Down"`, `"Left"`, `"Right"`
- Special keys: `"space"`, `"Return"`, `"Delete"`

5.6 Turtle and OOP

Here's where turtle connects to everything from Chapters 2-3. We can create custom classes that wrap turtle objects:

```
1  import turtle
2
3  class DrawingTurtle:
4      def __init__(self, name, color, size):
5          self.name = name
6          self.turtle = turtle.Turtle()
7          self.turtle.color(color)
8          self.turtle.pensize(size)
9          self.turtle.speed(5)
10
11     def draw_square(self, side_length):
12         for i in range(4):
13             self.turtle.forward(side_length)
14             self.turtle.right(90)
15
16     def draw_circle(self, radius):
17         self.turtle.circle(radius)
18
19     def move_to(self, x, y):
20         self.turtle.penup()
21         self.turtle.goto(x, y)
22         self.turtle.pendown()
23
```

```
# Create turtle objects --- each has its own color and pen size
artist1 = DrawingTurtle("Pablo", "blue", 3)
artist2 = DrawingTurtle("Penny", "purple", 5)

screen = turtle.Screen()

artist1.draw_square(80)
artist1.move_to(200, 100)
artist1.draw_circle(40)

artist2.move_to(-200, -100)
artist2.draw_square(60)

screen.exitonclick()
```

This demonstrates core OOP concepts in action: each `DrawingTurtle` is an **object** with its own state (name, color, size) and behaviors (`draw_square`, `draw_circle`). Different objects operate independently — **encapsulation** and **polymorphism** at work.

Try it yourself! Create a `ColorfulTurtle` class with a method `draw_star(size)` that draws a five-pointed star. Make three turtles with different colors and have each one draw a star at a different position.

6 Pygame Development

Now that you've created drawings with Turtle, you're ready for the next level: game development! **Pygame** is a Python library for creating games with moving objects, sound effects, and animations. It uses OOP extensively, making it perfect for applying the concepts from Chapters 2-3.

Before using Pygame, install it with: `pip install pygame`

6.1 The Game Loop

Every Pygame game follows the same core structure — a **game loop** that runs continuously, handling input, updating game state, and drawing to the screen:

```
import pygame

# Initialize Pygame
pygame.init()

# Set up the display
screen = pygame.display.set_mode((800, 600))
pygame.display.set_caption("My First Game")
clock = pygame.time.Clock()

# Game loop
running = True
while running:
    # 1. Handle events (input, window close)
```

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # 2. Update game state
    # (move objects, check collisions, etc.)

    # 3. Draw everything
    screen.fill((0, 0, 50))        # Dark blue background
    pygame.display.flip()           # Show the new frame
    clock.tick(60)                  # 60 frames per second

pygame.quit()
```

This loop runs roughly 60 times per second, creating the illusion of smooth movement. Pygame uses the same **RGB color system** as Turtle (Chapter 5) — for example, `(255, 0, 0)` is red and `(0, 0, 0)` is black.

6.2 Sprites

A **sprite** is a game object (a character, enemy, bullet, etc.) represented by an image or shape that can move around the screen. In Pygame, sprites are classes that inherit from `pygame.sprite.Sprite`, demonstrating **inheritance** from Chapter 3:

```
import pygame

class Spaceship(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        self.image = pygame.Surface((40, 30))
        self.image.fill((0, 255, 0))        # Green rectangle
        self.rect = self.image.get_rect()
        self.rect.center = (400, 500)        # Start position
        self.speed = 5

    def update(self):
        keys = pygame.key.get_pressed()
        if keys[pygame.K_LEFT]:
            self.rect.x -= self.speed
        if keys[pygame.K_RIGHT]:
            self.rect.x += self.speed
        if keys[pygame.K_UP]:
            self.rect.y -= self.speed
        if keys[pygame.K_DOWN]:
            self.rect.y += self.speed

        # Keep on screen
        self.rect.clamp_ip(screen.get_rect())
```

Every sprite has two essential attributes: **`self.image`** (what it looks like) and **`self.rect`** (its position and size as a rectangle).

6.2.1 The `rect` Object

The `rect` attribute is a `pygame.Rect` that stores the sprite's position and size. It has many useful properties:

```
spaceship.rect.x              # Left edge
spaceship.rect.y              # Top edge
spaceship.rect.center         # (x, y) of center
spaceship.rect.width          # Width in pixels
spaceship.rect.bottom         # Bottom edge
spaceship.rect.right          # Right edge
```

6.2.2 Sprite Groups

Sprite groups hold multiple sprites and let you update and draw them all at once:

```
all_sprites = pygame.sprite.Group()
enemies = pygame.sprite.Group()

spaceship = Spaceship()
all_sprites.add(spaceship)

# In the game loop:
all_sprites.update()          # Calls update() on every sprite
all_sprites.draw(screen)      # Draws every sprite to the screen
```

6.3 Drawing and Blitting

Drawing creates shapes directly on the screen:

```
pygame.draw.circle(screen, (255, 255, 0), (100, 100), 30)          # Yellow circle
pygame.draw.rect(screen, (255, 0, 0), (200, 200, 40, 60))          # Red rectangle
pygame.draw.line(screen, (255, 255, 255), (0, 0), (800, 600), 2)   # White line
```

Blitting copies an image onto the screen:

```
spaceship_img = pygame.image.load("spaceship.png")
screen.blit(spaceship_img, (400, 500))
```

6.4 Collision Detection

Collision detection tells you when two objects touch — essential for any game. Pygame makes this easy with sprite groups:

```
import pygame
import random

class Asteroid(pygame.sprite.Sprite):
```

```
    def __init__(self):
        super().__init__()
        self.image = pygame.Surface((20, 20))
        self.image.fill((150, 75, 0))      # Brown
        self.rect = self.image.get_rect()
        self.rect.x = random.randint(0, 780)
        self.rect.y = random.randint(-100, -10)
        self.speed = random.randint(2, 6)

    def update(self):
        self.rect.y += self.speed
        if self.rect.top > 600:            # Fell off the bottom
            self.rect.y = random.randint(-100, -10)
            self.rect.x = random.randint(0, 780)
```

To check if the spaceship hits any asteroid:

```
# Check collisions between one sprite and a group
hits = pygame.sprite.spritecollide(spaceship, asteroids, True)
# True means: remove the asteroid from the group when hit

if hits:
    print(f"Hit {len(hits)} asteroid(s)!")
```

You can also check collisions between two individual rects:

```
if spaceship.rect.colliderect(asteroid.rect):
    print("Collision!")
```

6.5 Displaying Text

Pygame uses **fonts** to render text as images, then blits them to the screen:

```
font = pygame.font.Font(None, 36)       # Default font, size 36
text_surface = font.render("Score: 100", True, (255, 255, 255))
screen.blit(text_surface, (10, 10))
```

To center text:

```
text_surface = font.render("GAME OVER", True, (255, 0, 0))
text_rect = text_surface.get_rect(center=(400, 300))
screen.blit(text_surface, text_rect)
```

6.6 Keyboard Input

Pygame handles keyboard input in two ways:

Events — for single key presses (fires once per press):

```
for event in pygame.event.get():
    if event.type == pygame.KEYDOWN:
```

```
        if event.key == pygame.K_SPACE:
            print("Fire!")
```

Continuous — for held-down keys (checked every frame):

```
keys = pygame.key.get_pressed()
if keys[pygame.K_LEFT]:
    ship_x -= 5
if keys[pygame.K_RIGHT]:
    ship_x += 5
```

Use events for actions like shooting or pausing, and continuous input for smooth movement. Unlike Turtle, Pygame doesn't need `listen()` — input handling is built into the game loop.

6.7 Putting It All Together

Here's a mini space game that combines sprites, collision detection, text, and input:

```
import pygame
import random

pygame.init()
screen = pygame.display.set_mode((800, 600))
pygame.display.set_caption("Space Dodge!")
clock = pygame.time.Clock()
font = pygame.font.Font(None, 36)

class Ship(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        self.image = pygame.Surface((40, 30))
        self.image.fill((0, 255, 0))
        self.rect = self.image.get_rect(center=(400, 550))

    def update(self):
        keys = pygame.key.get_pressed()
        if keys[pygame.K_LEFT]:
            self.rect.x -= 5
        if keys[pygame.K_RIGHT]:
            self.rect.x += 5
        self.rect.clamp_ip(screen.get_rect())

class Rock(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        self.image = pygame.Surface((20, 20))
        self.image.fill((150, 75, 0))
        self.rect = self.image.get_rect()
        self.reset()

    def reset(self):
        self.rect.x = random.randint(0, 780)
        self.rect.y = random.randint(-200, -20)
        self.speed = random.randint(3, 7)

```

```
    def update(self):
        self.rect.y += self.speed
        if self.rect.top > 600:
            self.reset()

# Create sprites
ship = Ship()
all_sprites = pygame.sprite.Group(ship)
rocks = pygame.sprite.Group()
for i in range(8):
    rock = Rock()
    all_sprites.add(rock)
    rocks.add(rock)

score = 0
running = True

while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    all_sprites.update()
    score += 1

    # Check collisions
    if pygame.sprite.spritecollide(ship, rocks, False):
        running = False  # Game over!

    # Draw
    screen.fill((0, 0, 30))
    all_sprites.draw(screen)
    text = font.render(f"Score: {score}", True, (255, 255, 255))
    screen.blit(text, (10, 10))
    pygame.display.flip()
    clock.tick(60)

# Game over screen
screen.fill((0, 0, 0))
game_over_text = font.render(f"Game Over! Final Score: {score}", True, (255, 0, 0)
    )
text_rect = game_over_text.get_rect(center=(400, 300))
screen.blit(game_over_text, text_rect)
pygame.display.flip()
pygame.time.wait(3000)
pygame.quit()
```

This game demonstrates nearly every OOP concept from this guide: **classes** (Ship, Rock), **inheritance** (from `pygame.sprite.Sprite`), **polymorphism** (each sprite's `update()` behaves differently), **encapsulation** (each sprite manages its own state), and **abstraction** (you call `all_sprites.update()` without worrying about the details).

Try it yourself! Add a `Bullet` class that the ship fires when the player presses the spacebar. Check for collisions between bullets and rocks to let the player destroy them.

Epilogue

Remember that this guide covers only the fundamental topics of OOP, Turtle graphics, and Pygame. For questions not covered by this guide, search the internet or read the official documentation. Happy coding!