Object-Oriented Programming for Python with Turtle and Pygame

Updated June 29, 2025

Contents

Prologue

This guide offers a concise overview of object-oriented programming (OOP) in Python. It also introduces Pygame, a library used to create graphics games in Python. Before delving into the topics covered in this guide, it's highly recommended to master the concepts covered in the Intro to Python guide.

Use a Python interpreter like `programiz.com` to test the presented code blocks. Note that Pygame requires Python 3, particularly Python 3.6 and above. Environments running Python 2, such as the free version of Trinket, will not work.

NOTE: This compact guide is not a textbook, not comprehensive, and may not cover all the details in the topics presented. If you have any questions about any topic in this guide, feel free to search the internet or ask a coach.

1 Object-Oriented Programming (OOP)

1.1 Before we Begin...

Imagine you're creating a fun animal simulator game! In this game, you can have different pets like dogs, cats, and birds.

Each animal has its own special traits:

- Some animals are fast, others are slow
- Some animals make loud sounds, others are quiet
- Some animals like to play, others prefer to sleep
- Each animal has a favorite food

Instead of writing separate code for every single animal, OOP helps us organize our code by grouping similar animals together. This makes our code much easier to understand and work with!

1.2 What is OOP?

Object-oriented programming, or **OOP** for short, is a widely used programming paradigm that organizes code using objects. Just as we perceive the characteristics and behaviors of people, objects, or surroundings in reality, OOP mimics these perspectives to make code more understandable to both developers and others.

Since it's likely that understanding OOP will be challenging at this point, let's proceed to the next section.

2 Classes

To start using OOP, developers dive into **classes**. A class is a "blueprint" that is used to create objects.

Recall functions from the Intro to Python guide. To utilize a function, we first *define* it with a header that consists of a name and arguments, a body of algorithm, and a series of return-statements. Then, we call the function to actually use it.

Now, the procedure of definition and utilization is seen at a much larger scope, where a class falls under the definition step.

A class, like functions, starts with a header:

```
class <ClassName>:
```

where `<ClassName>` is the name of the class. Unlike names of variables or functions, by common practice, we capitalize each word in the name and do *not* use "`_`" to represent the space.

In general, a class consists of *class variables, instance variables, an `__init__()` method, and instance methods.*

2.1 Class and Instance Variables

Remember our animal simulator from the previous section. Each pet has different traits that can be organized into categories. For example, all pets have names, ages, and favorite foods. By organizing these traits, we can easily manage each pet in our program.

Instance variables are unique to each object created from a class. Since a class doesn't represent a single object, instance variables can also depend on the parameters provided when creating an object. For instance, an instance variable named `favorite_food` can be defined. Then, when creating objects for pets, each pet's actual favorite food can be initialized with a specific value.

Class variables are similar to instance variables, except that all objects created from the class share the same value. It is also known as static variables in other languages.

One way to distinguish between instance and class variables is to observe the presence of the `self` tag in front of the variable name, assuming that the variable is within the scope of a class. If the `self` syntax is included in a variable or function, it means that such will be unique to each object. Table 1 at page 6 summarizes the access tag to be used for each case.

A code example will be shown in the next section.

2.2 `__init__()` Method

An `__init__()` method, also known as the constructor, is a function that is run when an affiliated object is *initialized*. In general, `__init__()` methods take optional arguments for the object creation and assign them into the corresponding instance variables. For some cases, such arguments are taken in as a dependent for actions taken when creating an object. A class should have exactly one `__init__()` method.

The format to define an `__init__()` method is the following:

```
def  __init__(self, <arguments>):
```

where `<arguments>` are replaced with the actual arguments.

The `self` argument, like the `self` tag for instance variables, implies unique execution of the `__init__()` method for each object creation. When taking object creation parameters, the first parameter matches the *second* argument of the `__init__()` method, ignoring the `self` argument.

Now, let's create a `Pet` class that will be used to create our virtual pets as objects.

```python
class Pet:
    # Like if-statements, while- or for-loops, the class header that ends with a ":" is followed with the body which are indented.

    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

        print(f"{self.name} is a {self.age} year old {self.animal_type} who loves {self.favorite_food}!")

        Pet.pet_count += 1

# Since the code below is at the same indentation level as the class header, it is now outside the scope of the class.
buddy = Pet("Buddy", 3, "dog", "chicken")
print(f"Total pets: {Pet.pet_count}")
print(f"Pet name: {buddy.name}, Age: {buddy.age} years old")
```

```
Buddy is a 3 year old dog who loves chicken!
Total pets: 1
Pet name: Buddy, Age: 3 years old
```

Here, the Pet class is created, and the __init__() method does the following:

1. It accepts name, age, animal_type, and favorite_food as parameters and sets each into instance variables.
2. Next, a print statement prints what kind of new pet is created.
3. Outside __init__() method, a class variable pet_count is initialized to 0, and the __init__() method updates the pet_count for each creation of Pet object.

To avoid misrepresentations, class variables are defined outside the scope of __init__() method, whereas instance variables are defined inside.

Creating an object of Pet class named buddy at line 17 executes the __init__() method. Then, lines 18 and 19 print the class and instance variables of buddy, respectively. Notice that, being outside the scope of the class, the class variables have the tag that is the name of the class, and instance variables have the tag of the object name.

When the British coder wrote a Python class, they named the constructor def __innit__(self): because it's always initializing, *innit*? 🙃

(Do not try this; it will not work like a constructor...)

2.3 Instance Methods

Instance methods are functions that are unique to each object created from a class, just like instance variables. In general, such uniqueness depend on the instance variables of the same object.

In our pet simulator example, while instance variables represent the characteristics of each pet, instance methods will represent the behaviors of each pet.

The function header of an instance method is defined as following:

```
def <fn_name>(self, <arguments>...):
```

where <fn_name> represents the function name, and <arguments> denotes the sequence of arguments. Like instance variables and the __init__() method, the self tag indicates that these methods are exclusive to each object and is disregarded when comparing parameters from function calls.

Let's look at an example of instance method. Assume that the same variables and the __init__() method from last example were used.

Scope	Member Type	Access Tag	Example
Inside class (outside methods)	Class variable	Class name	`MyClass.var`
	Instance variable	Invalid	N/A
	Instance method	Invalid	N/A
Inside instance method (including `__init__()` method)	Class variable	Class name	`MyClass.var`
	Instance variable	`self`	`self.var`
	Instance method	`self`	`self.method()`
Outside class	Class variable	Class name	`MyClass.var`
	Instance variable	Instance name	`obj.var`
	Instance method	Instance name	`obj.method()`

Table 1: Access tags for class and instance variables and instance methods. Assume `MyClass` is the name of the class, and `obj` is the name of the object created from `MyClass`.

```python
import random   # See Intro to Python guide

class Pet:
    # Class and instance variables and __init__() method placed here

    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 a 3 year old dog who loves chicken!
Buddy is playing happily!
Buddy loves eating chicken!
Buddy doesn't really like broccoli.
```

Here, we have two instance methods: `play` which doesn't take any arguments, and `eat` which takes a `food` argument. The `play` method checks the pet's age, while the `eat` method compares the given food with the pet's favorite food.

The access tags for instance methods are the same as instance variables. Table 1 at page 6 summarizes the access tag to be used for each case.

3 Objects

3.1 Object Creation (Instantiation)

Sections 2.2 and 2.3 briefly explored an example of object creation. The formal term for object creation is called **instantiation**. Instantiation means creating an *instance* of a class, which, in this context, refers to an object.

Like how we call a function using the parameters fed in the same order as corresponding arguments, instantiating an object requires the same order of parameters as the arguments required by the `__init__()` method of the class.

3.2 Classes as Data Types, Objects as Variables

From the Intro to Python guide, we learned about **primitive data types** that contain a single piece of data, such as integers, floats, and Boolean. We also learned **non-primitive data types**, such as strings, contain multiple pieces of data. They also include lists, dictionaries, and other kinds of data structures.

Both categories of data types are defined using classes, and the variables created with such types are objects. In other words, classes can be viewed as “custom data types”.

What does this mean? Classes can be cared like data types, and objects can be cared like variables. For example, lists in Python contain items consisting of data of a certain type, or a mix of data from different data types. This includes objects created from custom classes.

```python
class Toy:
    def __init__(self, name, color, size):
        self.name = name
        self.color = color
        self.size = size
        print(f"Created a {self.color} {self.name} that is {self.size} size!")

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

# Create toy objects
teddy = Toy("teddy bear", "brown", "large")
ball = Toy("ball", "red", "small")
robot = Toy("robot", "blue", "medium")

# Put toys in a list and play with each one
toy_box = [teddy, ball, robot]
for toy in toy_box:
    toy.play_with()
```

```
Created a brown teddy bear that is large size!
Created a red ball that is small size!
Created a blue robot that is medium size!
Playing with the brown teddy bear!
Playing with the red ball!
```

```
Playing with the blue robot!
```

Understanding classes as data types and objects as variables will significantly enhance your comprehension of OOP.

4 Principles of OOP

There are four principles revolving OOP. The majority of these principles further allows us avoid repeating writing similar codes. In Python, only inheritance and polymorphism are used frequently throughout the development; however, it is good idea to have understandings in abstraction and encapsulation as well.

4.1 Inheritance

Inheritance allows a class to inherit properties and methods from another class. The class that inherits is called the child class or derived class, and the class being inherited from is called the parent class or base class.

In Python, inheritance is defined by putting the parent class name in parentheses after the child class name. The `super()` function is used to call methods from the parent class.

```python
# Base class
class Vehicle:
    def __init__(self, name, speed):
        self.name = name
        self.speed = speed
        print(f"Created a {self.name} that goes {self.speed} mph!")

    def move(self):
        print(f"The {self.name} is moving at {self.speed} mph!")

# Child classes
class Car(Vehicle):
    def __init__(self, name, speed, doors):
        super().__init__(name, speed)
        self.doors = doors
        print(f"This car has {self.doors} doors.")

    def honk(self):
        print(f"The {self.name} goes BEEP BEEP!")

class Bicycle(Vehicle):
    def __init__(self, name, speed, has_bell):
        super().__init__(name, speed)
        self.has_bell = has_bell

    def ring_bell(self):
        if self.has_bell:
            print(f"The {self.name} goes RING RING!")
        else:
```

```
            print(f"The {self.name} doesn't have a bell.")

# Create vehicle objects
my_car = Car("Red Car", 60, 4)
my_bike = Bicycle("Blue Bike", 15, True)

# Use inherited and new methods
my_car.move()  # From Vehicle class
my_car.honk()  # Car's own method

my_bike.move()  # From Vehicle class
my_bike.ring_bell()  # Bicycle's own method
```

```
Created a Red Car that goes 60 mph!
This car has 4 doors.
Created a Blue Bike that goes 15 mph!
The Red Car is moving at 60 mph!
The Red Car goes BEEP BEEP!
The Blue Bike is moving at 15 mph!
The Blue Bike goes RING RING!
```

4.2 Polymorphism

Polymorphism means "many forms". It allows objects of different classes to be treated as objects of a common base class. In Python, polymorphism is achieved when different classes have methods with the same name but different implementations.

```
class Dog:
    def make_sound(self):
        return "Woof!"

class Cat:
    def make_sound(self):
        return "Meow!"

class Duck:
    def make_sound(self):
        return "Quack!"

# Create different animals
animals = [Dog(), Cat(), Duck()]

# Same method name, different sounds
for animal in animals:
    print(animal.make_sound())
```

```
Woof!
Meow!
Quack!
```

Here, each animal class has a `make_sound()` method, but each implements it differently. The same method name produces different behaviors based on the object type.

4.3 Abstraction

Abstraction hides complex implementation details and shows only the essential features. In Python, abstraction is achieved through abstract classes and methods using the `abc` module. However, for beginners, it's enough to understand that abstraction means hiding complexity and providing simple interfaces.

```python
# Simple abstraction example - user doesn't need to know how microwave works
    inside
class Microwave:
    def __init__(self):
        self.power = 0
        self.time = 0

    def heat_food(self, food, minutes):
        self._set_power(800)      # Hidden complexity
        self._set_timer(minutes) # Hidden complexity
        self._start_heating()     # Hidden complexity
        print(f"Your {food} is ready!")

    def _set_power(self, watts):  # Private method (hidden)
        self.power = watts
        print(f"Power set to {watts} watts")

    def _set_timer(self, minutes):  # Private method (hidden)
        self.time = minutes
        print(f"Timer set to {minutes} minutes")

    def _start_heating(self):  # Private method (hidden)
        print("Heating started...")

# User only needs to know this simple interface
microwave = Microwave()
microwave.heat_food("pizza", 2)
```

```
Power set to 800 watts
Timer set to 2 minutes
Heating started...
Your pizza is ready!
```

4.4 Encapsulation

Encapsulation bundles data and methods into a single unit (class) and restricts access to some components. In Python, encapsulation is achieved using private attributes and methods by prefixing them with underscore(s).

```python
class PiggyBank:
    def __init__(self):
        self._coins = 0          # Protected attribute
        self.__secret_code = 1234  # Private attribute

    def add_coin(self):
        self._coins += 1
```

```python
        print(f"Added a coin! Total coins: {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!")
            return self._coins
        else:
            print("Wrong code! Bank stays locked!")
            return 0

    def __count_money(self):  # Private method
        return self._coins * 25  # Each coin worth 25 cents

bank = PiggyBank()
bank.add_coin()
bank.add_coin()
bank.add_coin()
print(f"Current coins: {bank.get_coins()}")
bank.open_bank(1234)  # Correct code
bank.open_bank(9999)  # Wrong code
```

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

5 import and Cross-File Programming

When Python programs become complex, organizing code across multiple files becomes essential. The `import` statement allows you to use code from other Python files and libraries.

5.1 import Methods

There are several ways to import modules in Python:

```python
# Method 1: Import entire module
import math
print(math.sqrt(25))  # Output: 5.0
print(math.pi)        # Output: 3.141592653589793

# Method 2: Import specific functions
from math import sqrt, pi
print(sqrt(25))  # Output: 5.0
print(pi)        # Output: 3.141592653589793

# Method 3: Import with alias (nickname)
```

```python
import random as r
print(r.randint(1, 10))  # Random number between 1 and 10

# Method 4: Import everything (not recommended)
from math import *
print(sqrt(25))  # Output: 5.0
```

```
5.0
3.141592653589793
5.0
3.141592653589793
7
5.0
```

To import your own Python files, simply use the filename without the `.py` extension:

```python
# If you have a file called "my_games.py"
from my_games import GuessGame, DiceGame

# Or import the entire file
import my_games
game = my_games.GuessGame()
```

5.2 Libraries Continued

Python's strength lies in its vast ecosystem of libraries. Libraries are collections of pre-written code that solve common programming problems.

5.2.1 Documentations

Every good Python library comes with **documentation** that explains how to use its functions and classes. Reading documentation is a crucial skill for programmers.

Key places to find Python documentation:

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

```python
import math
help(math.sqrt)  # Shows documentation for sqrt function

# You can also get help on any function
help(print)      # Shows how to use print function
```

5.2.2 sys and os

Two important built-in libraries for system operations:

sys - System-specific parameters and functions:

```python
import sys

print("Python version:", sys.version)     # Python version info
print("Platform:", sys.platform)          # Your computer type
print("Path:", sys.path[0])               # Where Python looks for files
```

```
Python version: 3.9.7 (default, Sep 16 2021, 08:50:36)
Platform: darwin
Path: /Users/student/python_projects
```

os - Operating system interface:

```python
import os

print("Current folder:", os.getcwd())          # Where you are now
print("Files here:", os.listdir("."))          # List files in current folder
print("Does 'test.txt' exist?", os.path.exists("test.txt"))  # Check if file
    exists

# Create a new folder (be careful!)
# os.mkdir("my_new_folder")
```

```
Current folder: /Users/student/python_projects
Files here: ['main.py', 'my_game.py', 'README.txt']
Does 'test.txt' exist? False
```

Here's a fun example using `random` for a simple guessing game:

```python
import random

# Simple number guessing game
secret_number = random.randint(1, 10)
print("I'm thinking of a number between 1 and 10!")

guess = int(input("What's your guess? "))
if guess == secret_number:
    print("You got it! Great job!")
else:
    print(f"Sorry! The number was {secret_number}. Try again!")
```

```
I'm thinking of a number between 1 and 10!
What's your guess? 7
Sorry! The number was 3. Try again!
```

Now that you know how to import libraries and use different Python modules, you're ready to explore more exciting programming! The next sections will show you how to create visual programs that draw pictures and games on the screen.

6 Turtle Basics

Turtle is a fun Python library that lets you draw pictures and create graphics using simple commands. It's like having a digital pen that you can control with code! The turtle starts in the center of the screen and follows your commands to draw lines, shapes, and colorful patterns.

Turtle comes built-in with Python, so you don't need to install anything extra. It's perfect for learning programming because you can see your code come to life as drawings on the screen.

6.1 Basic Turtle Commands

Let's start with the most important turtle commands. Think of the turtle as a little robot that can move around and draw:

```python
import turtle

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

# Basic movement commands
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()
```

This code creates a simple path that the turtle draws. When you run it, you'll see a window open with lines showing where the turtle moved!

6.2 Drawing Shapes

One of the most fun things about turtle is drawing shapes. Let's create some basic shapes:

```python
import turtle

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

# Draw a square
for i in range(4):
    artist.forward(100)
    artist.right(90)

# Move to a new position without drawing
artist.penup()            # Lift the pen
artist.goto(150, 0)       # Move to position (150, 0)
```

```python
artist.pendown()          # Put the pen down

# Draw a triangle
for i in range(3):
    artist.forward(80)
    artist.right(120)

screen.exitonclick()
```

Here we use loops to draw shapes efficiently. The square uses 4 sides with 90-degree turns, and the triangle uses 3 sides with 120-degree turns.

6.3 Colors and Pen Control

Let's make our drawings more colorful and interesting:

```python
import turtle

# Create turtle
painter = turtle.Turtle()
screen = turtle.Screen()
screen.bgcolor("lightblue")  # Set background color

# Set turtle properties
painter.color("red")         # Set pen color
painter.pensize(5)           # Make lines thicker
painter.speed(3)             # Set drawing speed (1-10)

# Draw a colorful flower
for i in range(6):
    painter.circle(50)       # Draw a circle with radius 50
    painter.right(60)        # Turn to create petal pattern

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

screen.exitonclick()
```

```
# This creates a beautiful flower drawing with:
# - Light blue background
# - Red flower petals (6 circles)
# - Green stem
# - Thick lines for better visibility
```

6.4 Turtle and Object-Oriented Programming

Here's where turtle connects to what we've learned about OOP! Each turtle is actually an object with its own properties and methods:

```python
import turtle

class DrawingTurtle:
    def __init__(self, name, color, size):
        self.name = name
        self.turtle = turtle.Turtle()
        self.turtle.color(color)
        self.turtle.pensize(size)
        self.turtle.speed(5)
        print(f"Created turtle named {self.name} with {color} color!")

    def draw_square(self, side_length):
        print(f"{self.name} is drawing a square!")
        for i in range(4):
            self.turtle.forward(side_length)
            self.turtle.right(90)

    def draw_circle(self, radius):
        print(f"{self.name} is drawing a circle!")
        self.turtle.circle(radius)

    def move_to(self, x, y):
        self.turtle.penup()
        self.turtle.goto(x, y)
        self.turtle.pendown()

# Create turtle objects
artist1 = DrawingTurtle("Pablo", "blue", 3)
artist2 = DrawingTurtle("Penny", "purple", 5)

# Set up screen
screen = turtle.Screen()
screen.setup(800, 600)

# Use our turtle objects
artist1.draw_square(80)
artist1.move_to(200, 100)
artist1.draw_circle(40)

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

screen.exitonclick()
```

```
Created turtle named Pablo with blue color!
Created turtle named Penny with purple color!
Pablo is drawing a square!
Pablo is drawing a circle!
Penny is drawing a square!
```

6.5 Interactive Turtle Programs

We can make turtle respond to keyboard input, creating interactive drawings:

```python
import turtle

class ControllableTurtle:
    def __init__(self):
        self.turtle = turtle.Turtle()
        self.turtle.color("orange")
        self.turtle.pensize(3)
        self.turtle.speed(6)

        # Set up the screen
        self.screen = turtle.Screen()
        self.screen.setup(600, 600)
        self.screen.title("Control the Turtle with Arrow Keys!")

        # Bind keys to methods
        self.screen.onkey(self.move_up, "Up")
        self.screen.onkey(self.move_down, "Down")
        self.screen.onkey(self.move_left, "Left")
        self.screen.onkey(self.move_right, "Right")
        self.screen.onkey(self.change_color, "space")

        # Listen for key presses
        self.screen.listen()

        self.colors = ["red", "blue", "green", "purple", "orange", "yellow"]
        self.color_index = 0

    def move_up(self):
        self.turtle.setheading(90)   # Point up
        self.turtle.forward(20)

    def move_down(self):
        self.turtle.setheading(270)  # Point down
        self.turtle.forward(20)

    def move_left(self):
        self.turtle.setheading(180)  # Point left
        self.turtle.forward(20)

    def move_right(self):
        self.turtle.setheading(0)    # Point right
        self.turtle.forward(20)

    def change_color(self):
        self.color_index = (self.color_index + 1) % len(self.colors)
        self.turtle.color(self.colors[self.color_index])
        print(f"Color changed to {self.colors[self.color_index]}!")

# Create and use the controllable turtle
my_turtle = ControllableTurtle()
print("Use arrow keys to move, spacebar to change colors!")
my_turtle.screen.exitonclick()
```

```
Use arrow keys to move, spacebar to change colors!
Color changed to blue!
Color changed to green!
# (Output appears as you press keys)
```

6.6 Creating Art with Turtle

Let's combine everything we've learned to create a beautiful spiral pattern:

```python
import turtle
import random

class SpiralArtist:
    def __init__(self):
        self.turtle = turtle.Turtle()
        self.screen = turtle.Screen()
        self.screen.bgcolor("black")
        self.screen.setup(800, 800)
        self.turtle.speed(0)  # Fastest speed

        self.colors = ["red", "orange", "yellow", "green", "blue", "purple", "pink
            ", "cyan"]

    def draw_colorful_spiral(self):
        for i in range(200):
            # Pick a random color
            color = random.choice(self.colors)
            self.turtle.color(color)

            # Draw and turn
            self.turtle.forward(i * 2)
            self.turtle.right(91)  # Slightly more than 90 degrees creates spiral

    def draw_rainbow_flower(self, petals=12):
        for i in range(petals):
            # Use different colors for each petal
            color = self.colors[i % len(self.colors)]
            self.turtle.color(color)

            # Draw petal
            self.turtle.circle(100)
            self.turtle.right(360 / petals)

# Create spiral art
artist = SpiralArtist()
print("Creating colorful spiral art...")
artist.draw_colorful_spiral()

# Reset position for flower
artist.turtle.home()
artist.turtle.clear()

print("Creating rainbow flower...")
artist.draw_rainbow_flower()

artist.screen.exitonclick()
```

```
Creating colorful spiral art...
Creating rainbow flower...
```

Turtle graphics provide an excellent bridge between basic programming concepts and visual creativity. You can see your code come to life as colorful drawings, making it perfect for understanding how programming instructions translate into visual results.

The turtle library demonstrates many OOP concepts we've learned:

- Each turtle is an **object** with its own state (position, color, direction)
- Turtle methods like `forward()`, `color()`, and `circle()` are **instance methods**
- We can create custom turtle classes that **inherit** from or use turtle objects
- Different turtle objects can have different behaviors (**polymorphism**)

This foundation in turtle graphics prepares you perfectly for the more advanced game development concepts we'll explore with Pygame!

7 Pygame Basics

Now that you've learned how to create drawings with Turtle graphics, you're ready for the next level: game development! **Pygame** is a Python library for creating games and multimedia applications. It provides tools for graphics, sound, and game logic. While Turtle is great for simple drawings, Pygame lets you create interactive games with moving objects, sound effects, and complex animations.

Pygame uses object-oriented programming extensively, making it perfect for applying all the OOP concepts you've learned in this guide.

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

7.1 Sprites

A **sprite** in Pygame is a 2D image or animation that can be moved around the screen. In Pygame, sprites are implemented as classes that inherit from `pygame.sprite.Sprite`.

```python
import pygame

class Spaceship(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        # Create a simple colored rectangle for our spaceship
        self.image = pygame.Surface((40, 30))
        self.image.fill((0, 255, 0))  # Green spaceship

        # Get rectangle for positioning
        self.rect = self.image.get_rect()
        self.rect.center = (400, 500)  # Start at bottom center

    def update(self):
        # Move the spaceship with arrow keys
        keys = pygame.key.get_pressed()
```

```python
        if keys[pygame.K_LEFT]:
            self.rect.x -= 5
        if keys[pygame.K_RIGHT]:
            self.rect.x += 5
        if keys[pygame.K_UP]:
            self.rect.y -= 5
        if keys[pygame.K_DOWN]:
            self.rect.y += 5

        # Keep spaceship on screen
        if self.rect.left < 0:
            self.rect.left = 0
        if self.rect.right > 800:
            self.rect.right = 800
```

7.1.1 Sprite.rect

The `rect` attribute is crucial for sprite positioning and collision detection. It's a `pygame.Rect` object that represents the sprite's position and size.

```python
# Common rect properties for positioning
spaceship.rect.x = 100          # Left edge position
spaceship.rect.y = 50           # Top edge position
spaceship.rect.center = (400, 300)      # Center position
spaceship.rect.bottom = 600     # Bottom edge position
spaceship.rect.right = 800      # Right edge position

# Size properties
spaceship.rect.width = 40       # Width in pixels
spaceship.rect.height = 30      # Height in pixels
```

7.1.2 Sprite Groups

Sprite groups are containers that hold multiple sprites. They make it easy to update and draw many sprites at once.

```python
# Create sprite groups
all_sprites = pygame.sprite.Group()
asteroids = pygame.sprite.Group()

# Add sprites to groups
spaceship = Spaceship()
asteroid1 = Asteroid()
asteroid2 = Asteroid()

all_sprites.add(spaceship, asteroid1, asteroid2)
asteroids.add(asteroid1, asteroid2)

# Update all sprites in groups
all_sprites.update()

# Draw all sprites in groups
all_sprites.draw(screen)
```

7.2 Running Loops

Every Pygame game needs a **game loop** that continuously updates the game state and redraws the screen.

```python
import pygame

# Initialize Pygame
pygame.init()

# Set up the display
screen = pygame.display.set_mode((800, 600))
pygame.display.set_caption("Space Adventure!")
clock = pygame.time.Clock()

# Create sprite groups
all_sprites = pygame.sprite.Group()
spaceship = Spaceship()
all_sprites.add(spaceship)

# Game loop
running = True
while running:
    # Handle events (like closing the window)
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Update all sprites
    all_sprites.update()

    # Draw everything
    screen.fill((0, 0, 50))  # Dark blue space background
    all_sprites.draw(screen)

    # Update the display
    pygame.display.flip()
    clock.tick(60)  # 60 frames per second

pygame.quit()
```

7.3 Draw and Blit

Drawing and **blitting** are fundamental Pygame operations for displaying graphics.

Drawing creates shapes directly on surfaces:

```python
# Draw space objects on screen
pygame.draw.circle(screen, (255, 255, 0), (100, 100), 30)    # Yellow sun
pygame.draw.rect(screen, (255, 0, 0), (200, 200, 40, 60))    # Red asteroid
pygame.draw.line(screen, (255, 255, 255), (0, 0), (800, 600), 2)  # White laser
```

Blitting copies one surface onto another:

```python
# Load space images
spaceship_image = pygame.image.load("spaceship.png")
star_image = pygame.image.load("star.png")

# Blit (copy) images to screen
screen.blit(spaceship_image, (400, 500))  # Spaceship at bottom center
screen.blit(star_image, (50, 50))         # Star in top left

# You can also blit part of an image (useful for animation)
screen.blit(spaceship_image, (400, 500), (0, 0, 32, 32))  # Only 32x32 part
```

Here's a simple complete space game example:

```python
import pygame
import random

class Star(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        self.image = pygame.Surface((3, 3))
        self.image.fill((255, 255, 255))  # White star
        self.rect = self.image.get_rect()
        self.rect.x = random.randint(0, 800)
        self.rect.y = random.randint(0, 600)

# Create stars for background
stars = pygame.sprite.Group()
for i in range(50):
    star = Star()
    stars.add(star)

# In your game loop:
# stars.draw(screen)  # Draw twinkling stars
```

Pygame combines OOP principles with game development, allowing you to create engaging space adventures and other interactive applications using the concepts learned in this guide.

Epilogue

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