Programming for Engineers

Lecture 9 - Practical Programming: OOP and PyGame

Radoslav Škoviera

Table of contents

1 OOP and PyGame

1.1 Decorators in Python

A decorator is a function that wraps another function and modifies its behavior. It is a useful tool for adding extra behavior to existing functions without modifying their source code.

```python
def decorator(func):
    def wrapper():
        print("Before")
        func()
        print("After")
    return wrapper

@decorator
def say_hello():
    print("Hello")
```

The `@` symbol is used to apply the decorator to the function `say_hello`.

```
say_hello()
```

```
Before
Hello
After
```

Decorators can be also applied to classes. For example, the `@dataclass` decorator is used to turn a class into **a data class**.

2 Object-oriented programming refresher

A class is a blueprint. An object is a concrete instance created from that blueprint. The class tells us what the object knows and what it can do. Consider a very small example:

```
class Counter:
    def __init__(self, start: int = 0) -> None:
        self.value = start

    def increment(self) -> None:
        self.value += 1


counter = Counter(start=10)
counter.increment()
print(counter.value)
```

```
11
```

This example demonstrates three important OOP principles. First, `Counter` is a **class**, whereas `counter` is an **object**. Second, the **attribute** `value` *belongs to the object*, not to the class definition as plain global data. Third, the **method** `increment` *changes the object's internal state*. The parameter `self` is the object that receives the method call.

```
from dataclasses import dataclass

@dataclass
class Point2:
    x: float = 0.0
    y: float = 0.0
```

The `@dataclass` decorator removes boilerplate for simple data-centric classes. It automatically creates an initializer (`__init__` with the defined attributes) and a readable representation (`__repr__`, attributes). This is especially useful when many classes mainly store state.

```
p = Point2(1, 2)
print(p)
print(f"x: {p.x}, y: {p.y}")
p.x = 3
p.y *= 2.5
print(p)
```

```
Point2(x=1, y=2)
x: 1, y: 2
Point2(x=3, y=5.0)
```

The `@dataclass` can be combined with "custom" methods.

```
from dataclasses import dataclass, field
from math import atan2, cos, sin, sqrt, degrees

@dataclass
class Vector2:
    x: float = 0.0
    y: float = 0.0

    def magnitude(self) -> float:
        return sqrt(self.x ** 2 + self.y ** 2)

    def __add__(self, other: "Vector2") -> "Vector2":
        return Vector2(self.x + other.x, self.y + other.y)

    def __mul__(self, scalar: float) -> "Vector2":
        return Vector2(self.x * scalar, self.y * scalar)

    def __truediv__(self, scalar: float) -> "Vector2":
        return Vector2(self.x / scalar, self.y / scalar)

    @classmethod
    def from_polar(cls, magnitude: float, angle_radians: float) -> "Vector2":
        return cls(magnitude * cos(angle_radians), magnitude * sin(angle_radians))

    @staticmethod
    def normalize(vector: "Vector2") -> "Vector2":
        return vector / vector.length

    @property
    def angle(self) -> float:
        return degrees(atan2(self.y, self.x))

    @property
    def length(self) -> float:
        return sqrt(self.x ** 2 + self.y ** 2)
```

The `__add__` and `__mul__` are "magic methods" (sometimes called "dunder methods" for "double underscore") the serve a special role, for example, in this case, they are used with the `+` and `*` operators. The `@classmethod from_polar` is an alternative constructor. This method is "attached" to the class, not the object. The `@staticmethod normalize` is a function, attached to the class but unlike `classmethod`, it has no access to the "cls" variable. It is basically "a standalone" function. The `@property angle` looks like an attribute when used from outside, but internally it is computed like a method. Properties are useful when a value is conceptually part of the object's interface, even though it is derived from other state.

```
v1 = Vector2(1, 2)
v2 = Vector2(3, 4)
v3 = v1 + v2
v4 = v1 * 2.5
print(f"{v1} + {v2} =\t{v3}")
print(f"{v1} * {2.5} =\t\t{v4}")

v5 = Vector2.from_polar(10, 0.5)
print(f"from_polar(10, 0.5): {v5}")

v6 = Vector2.normalize(v1)
print(f"normalize(v1): {v6}")
print(f"v6.length = {v6.length}")

print(f"v1.angle = {v1.angle:.2f} degrees")
```

```
Vector2(x=1, y=2) + Vector2(x=3, y=4) = Vector2(x=4, y=6)
Vector2(x=1, y=2) * 2.5 =       Vector2(x=2.5, y=5.0)
from_polar(10, 0.5): Vector2(x=8.775825618903728, y=4.79425538604203)
normalize(v1): Vector2(x=0.4472135954999579, y=0.8944271909999159)
v6.length = 0.9999999999999999
v1.angle = 63.43 degrees
```

2.1 Instance attributes versus class attributes

There are *instance attributes* and *class attributes*. An instance attribute belongs to one object. A class attribute is shared at the class level.

```
class GameObject:
    created_count = 0  # class attribute

    def __init__(self, name: str) -> None:
        self.name: str = name  # instance attribute
        self.position: Vector2 = Vector2()  # instance attribute
        self.velocity: Vector2 = Vector2()  # instance attribute
        type(self).created_count += 1  # update class attribute

player = GameObject("Player")
```

```
print(f"Created {player.created_count} objects")
print(player.name)
print(player.position)
print(player.velocity)
```

```
Created 1 objects
Player
Vector2(x=0.0, y=0.0)
Vector2(x=0.0, y=0.0)
```

The attributes `name`, `position`, and `velocity` belong to each separate object. By contrast, `created_count` is attached to the class, and it counts how many objects have been created in that class hierarchy.

2.2 Composition in OOP

Instead of creating a large class for everything, a better design is often *composition*: build one object out of smaller objects with distinct functionalities.

For example, the health of a game object could be a *component*:

```
@dataclass
class Health:
    maximum: int
    current: int | None = None

    def __post_init__(self) -> None:
        if self.current is None:
            self.current = self.maximum

    def damage(self, amount: int) -> None:
        self.current = max(0, self.current - amount)

    def heal(self, amount: int) -> None:
        self.current = min(self.maximum, self.current + amount)
```

Then a game object *contains* a `Health` object, which holds the "health functionality". This can be re-used in different objects and if necessary, changes are contained to smaller scope.

```
@dataclass
class GameObject:
    name: str
    position: Vector2 = field(default_factory=Vector2)
    velocity: Vector2 = field(default_factory=Vector2)
    hit_points: Health = field(default_factory=lambda: Health(100))

go = GameObject("Player")
print(go.hit_points)
```

```
go.hit_points.damage(10)
print(go.hit_points)
go.hit_points.heal(20)
print(go.hit_points)
```

```
Health(maximum=100, current=100)
Health(maximum=100, current=90)
Health(maximum=100, current=100)
```

```
@dataclass
class Health:
    maximum: int
    current: int | None = None
    shield: int = 10

    def __post_init__(self) -> None:
        if self.current is None:
            self.current = self.maximum

    def damage(self, amount: int) -> None:
        if self.shield > 0:
            diff = self.shield - amount
            self.shield = max(0, diff)
            if diff >= 0:
                return
            amount = -diff
        self.current = max(0, self.current - amount)

    def heal(self, amount: int) -> None:
        self.current = min(self.maximum, self.current + amount)

go = GameObject("Player")
print(go.hit_points)
go.hit_points.damage(15)
print(go.hit_points)
go.hit_points.heal(20)
print(go.hit_points)
```

```
Health(maximum=100, current=100, shield=10)
Health(maximum=100, current=95, shield=0)
Health(maximum=100, current=100, shield=0)
```

2.3 Inheritance and polymorphism

Inheritance lets one class reuse and extend another. Polymorphism means that different subclasses may be treated through a common interface (method or attribute).

```python
@dataclass
class Player(GameObject):
    score: int = 0

    def draw_symbol(self) -> str:
        return "P"


@dataclass
class Enemy(GameObject):
    damage_points: int = 10

    def draw_symbol(self) -> str:
        return "E"
```

Both `Player` and `Enemy` are special cases (subclasses) of `GameObject`. A scene can therefore hold a list of `GameObject` references, even when the objects are actually a mixture of players and enemies.

```python
class Scene:
    def __init__(self, objects: list[GameObject] | None = None) -> None:
        self.objects = list(objects or [])

    def update(self, dt: float) -> None:
        for obj in self.objects:
            obj.update(dt)

    def symbols(self) -> str:
        return " ".join(obj.draw_symbol() for obj in self.objects)

scene = Scene()
scene.objects.append(Player("Player"))
scene.objects.append(Enemy("Dragon"))
print(scene.symbols())
```

```
P E
```

This is an example of polymorphism: the scene does not need to ask whether an object is a `Player` or an `Enemy` before calling `draw_symbol`. It simply relies on the common interface.

2.4 Properties and validation

Accessor methods ("getters" and "setters") are often used to validate and control access to an attribute (so called access control). In Python, they are defined using the `@property` and `@<property>.setter` decorators. For example, the `value` attribute of a `BoundedValue` class cannot be set to any value, it must be between `minimum` and `maximum`.

```python
class BoundedValue:
    def __init__(self, minimum: float, maximum: float, value: float, clip: bool = True) -> None:
        self.minimum = minimum
        self.maximum = maximum
        self.value = value
        self.clip = clip

    @property
    def value(self) -> float:
        return self._value

    @value.setter
    def value(self, new_value: float) -> None:
        if not (self.minimum <= new_value <= self.maximum):
            if self.clip:
                new_value = max(min(new_value, self.maximum), self.minimum)
            else:
                raise ValueError("value out of bounds")
        self._value = new_value


bv = BoundedValue(0, 100, 50)
print(bv.value)
bv.value = 200
print(bv.value)
bv2 = BoundedValue(0, 100, 50, clip=False)
bv2.value = 200
```

```
50
100

ValueError: value out of bounds
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[28], line 27
     25 print(bv.value)
     26 bv2 = BoundedValue(0, 100, 50, clip=False)
---> 27 bv2.value = 200
Cell In[28], line 18, in BoundedValue.value(self, new_value)
     16         new_value = max(min(new_value, self.maximum), self.minimum)
     17     else:
---> 18         raise ValueError("value out of bounds")
     19 self._value = new_value
ValueError: value out of bounds
```

Outside the class, the code still looks like ordinary attribute access. Inside, extra functionality can be implemented.

3 PyGame

3.1 Structure of a PyGame program

Nearly every PyGame program contains the same conceptual stages:

1. Initialize PyGame and create the display.
2. Enter the main loop.
3. Process events.
4. Update the program state.
5. Draw the new frame and show it.

A minimal example:

```
import pygame

pygame.init()  # initialize PyGame
screen: pygame.Surface = pygame.display.set_mode((800, 600))  # create the display
clock = pygame.time.Clock()  # create the clock
running = True
position = pygame.Vector2(400, 300)

while running:  # main "game" loop
    # get the time since the last frame (time delta)
    dt = clock.tick(60) / 1000.0

    # process events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:  # close the window
            running = False

    screen.fill((25, 25, 35))  # clear the screen
    # draw the "player"
    pygame.draw.circle(screen, (80, 170, 255), position, 20)
    pygame.display.flip()  # update the display

pygame.quit()  # quit / deinitialize PyGame
```

3.2 Drawing

The screen is redrawn every frame. Typical order of operations is:

- `screen.fill(color)` to clear the background,
- `pygame.draw.circle(...)`, `pygame.draw.rect(...)`, and `pygame.draw.lines(...)` for shapes,
- text rendering with `pygame.font.SysFont(...)`,
- `pygame.display.flip()` to reveal the completed frame.

Game objects often carry their own drawing methods, that are called by the main game object. These drawing methods typically accept a `surface: pygame.Surface` as an argument. It could be the "main screen" or a sub-surface (that is later **blitted** to the main screen).

3.3 Updating objects with delta time

The game state can be viewed as a discrete approximation to a continuous process (e.g., a motion). Position changes according to velocity. Velocity may change according to acceleration. The elapsed time `dt` scales the update. Relying on frame taking a constant time (i.e., updating motion at a constant speed) can result in all sorts of issues.

```
self.velocity += acceleration * dt
self.position += self.velocity * dt
```

3.4 Event handling: keyboard, mouse, and text input

PyGame programs are event-driven. The event queue contains window-close events, key presses, mouse clicks, and text input.

For continuous movement, it is common to combine explicit event handling with `pygame.key.get_pressed()`.

```
keys = pygame.key.get_pressed()
if keys[pygame.K_LEFT]:
    player.x -= speed * dt
if keys[pygame.K_RIGHT]:
    player.x += speed * dt
```

For discrete actions such as "spawn a marker where the user clicked", it is better to react directly to mouse events.

```
def spawn_marker(self, pos):
    markers.append(Marker(position=pygame.Vector2(pos)))

...
    for event in pygame.event.get():
        if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:
            self.spawn_marker(event.pos)
...
```

3.5 A first structured PyGame program

The file `pygame_basics_demo.py` contains a small PyGame program ("mini game").

Its main elements are these:

- a `App` class owns the window, clock, fonts, and objects,
- a `Player` object, which moves its position from the keyboard,
- `Marker` objects are created with the mouse-click and expire after a short lifetime,
- the code is separated into `handle_events`, `update`, `draw`, and `run`.