Object-Oriented Programming in Python

Building Better Code with Classes and Objects

What is OOP?

Object-Oriented Programming is a paradigm that organizes code around **objects** that contain:

- **Data** (attributes/properties)
- **Behavior** (methods/functions)

Four Pillars of OOP:

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

Classes and Objects

A **class** is a blueprint, an **object** is an instance of that class.

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

    def bark(self):
        return f"{self.name} says Woof!"

    def get_info(self):
        return f"{self.name} is {self.age} years old"

# Creating objects
buddy = Dog("Buddy", 3)
luna = Dog("Luna", 5)

print(buddy.bark())      # Buddy says Woof!
print(luna.get_info())   # Luna is 5 years old
```

Example: Music Tune Class

```
class Tune:
    def __init__(self, title, tune_type, key, abc_notation):
        self.title = title
        self.tune_type = tune_type
        self.key = key
        self.abc_notation = abc_notation
        self.play_count = 0

    def play(self):
        self.play_count += 1
        return f"Playing {self.title} (a {self.tune_type})"

    def get_stats(self):
        return f"'{self.title}' has been played {self.play_count} times"

# Create a tune
banshee = Tune("The Banshee", "reel", "E minor", "B2 E2...")
print(banshee.play())
print(banshee.get_stats())
```

Encapsulation

Hide internal data and provide controlled access

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return f"Deposited €{amount}"
        return "Invalid amount"

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return f"Withdrew €{amount}"
        return "Insufficient funds"
    def get_balance(self):
```

Encapsulation Example

```
account = BankAccount("Bryan", 1000)

# These work (controlled access)
account.deposit(500)
print(account.get_balance())  # 1500

account.withdraw(300)
print(account.get_balance())  # 1200

# This doesn't work (private attribute)
# print(account.__balance)  # AttributeError
```

Benefits:

- Protects data integrity
- Controls how data is accessed/modified
- Makes code easier to maintain

Inheritance

Create new classes based on existing ones

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

    def play(self):
        return f"Playing the {self.name}"

    def tune(self):
        return f"Tuning the {self.name}"

class Flute(Instrument):
    def __init__(self, name, material, keys):
        super().__init__(name, "woodwind")
        self.material = material
        self.keys = keys
    def play(self):  # Override parent method
```

Inheritance Example Continued

```
class Guitar(Instrument):
    def __init__(self, name, strings, electric=False):
        super().__init__(name, "string")
        self.strings = strings
        self.electric = electric

    def strum(self):
        type_str = "electric" if self.electric else "acoustic"
        return f"Strumming the {type_str} {self.name}"

# Create instances
irish_flute = Flute("Irish flute", "wood", 0)
acoustic = Guitar("Martin D-28", 6, False)

print(irish_flute.play())
print(acoustic.strum())
print(acoustic.tune())  # Inherited from Instrument
```

Game Development: Character Hierarchy

```
class Character:
    def __init__(self, name, health, attack_power):
        self.name = name
        self.health = health
        self.attack_power = attack_power

    def attack(self, target):
        damage = self.attack_power
        target.take_damage(damage)
        return f"{self.name} attacks {target.name} for {damage} damage!"

    def take_damage(self, damage):
        self.health -= damage
        if self.health < 0:
            self.health = 0

    def is_alive(self):
        return self.health > 0
```

Character Subclasses

```
class Warrior(Character):
    def __init__(self, name):
        super().__init__(name, health=120, attack_power=25)
        self.armor = 10

    def take_damage(self, damage):
        actual_damage = max(0, damage - self.armor)
        super().take_damage(actual_damage)

    def shield_bash(self, target):
        stun_damage = self.attack_power * 1.5
        target.take_damage(stun_damage)
        return f"{self.name} shield bashes {target.name}!"

class Mage(Character):
    def __init__(self, name):
        super().__init__(name, health=80, attack_power=35)
        self.mana = 100
```

Mage Powers

```
class Mage(Character):
    def __init__(self, name):
        super().__init__(name, health=80, attack_power=35)
        self.mana = 100

    def cast_spell(self, target):
        mana_cost = 20
        if self.mana >= mana_cost:
            self.mana -= mana_cost
            spell_damage = self.attack_power * 1.8
            target.take_damage(spell_damage)
            return f"{self.name} casts fireball on {target.name}!"
        return f"{self.name} is out of mana!"

    def regenerate_mana(self):
        self.mana = min(100, self.mana + 30)
        return f"{self.name} regenerates mana"
```

Battle Simulation

```
# Create characters
warrior = Warrior("Cú Chulainn")
mage = Mage("Merlin")

print(f"Warrior HP: {warrior.health}, Mage HP: {mage.health}")

# Combat
print(warrior.attack(mage))
print(f"Mage HP: {mage.health}")

print(mage.cast_spell(warrior))
print(f"Warrior HP: {warrior.health}")  # Reduced by armor

print(warrior.shield_bash(mage))
print(f"Mage HP: {mage.health}")

print(f"Warrior alive: {warrior.is_alive()}")
print(f"Mage alive: {mage.is_alive()}")
```

Polymorphism

Same interface, different implementations

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

    def process(self, signal):
        return signal

class Reverb(AudioEffect):
    def __init__(self, room_size):
        super().__init__("Reverb")
        self.room_size = room_size

    def process(self, signal):
        return f"{signal} -> [Reverb: {self.room_size}]"

class Delay(AudioEffect):
    def __init__(self, delay_time):
        super().__init__("Delay")
        self.delay_time = delay_time
```

Polymorphism in Action

```
class Compressor(AudioEffect):
    def __init__(self, ratio):
        super().__init__("Compressor")
        self.ratio = ratio

    def process(self, signal):
        return f"{signal} -> [Compress: {self.ratio}:1]"

# Create effect chain
effects = [
    Reverb("Large Hall"),
    Delay(250),
    Compressor(4)
]

# Process audio through all effects
audio_signal = "Clean Guitar"
for effect in effects:
    audio_signal = effect.process(audio_signal)
    print(audio_signal)
```

Data Science Example: Dataset Class

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

    def size(self):
        return len(self.data)

    def summary(self):
        return f"Dataset '{self.name}' contains {self.size()} records"

class TuneDataset(Dataset):
    def __init__(self, tunes):
        super().__init__("Irish Traditional Tunes", tunes)

    def get_by_type(self, tune_type):
        return [t for t in self.data if t['type'] == tune_type]

    def most_common_key(self):
        keys = [t['key'] for t in self.data]
        return max(set(keys), key=keys.count)
```

Using the Dataset Class

```
# Sample data
tunes = [
    {'title': 'The Banshee', 'type': 'reel', 'key': 'Emin'},
    {'title': 'Drowsy Maggie', 'type': 'reel', 'key': 'Emin'},
    {'title': 'The Silver Spear', 'type': 'reel', 'key': 'D'},
    {'title': 'The Butterfly', 'type': 'slip jig', 'key': 'Emin'},
]

dataset = TuneDataset(tunes)
print(dataset.summary())

reels = dataset.get_by_type('reel')
print(f"Found {len(reels)} reels")

print(f"Most common key: {dataset.most_common_key()}")
```

Godot-Style Node System

```
class Node:
    def __init__(self, name):
        self.name = name
        self.children = []
        self.parent = None

    def add_child(self, child):
        child.parent = self
        self.children.append(child)

    def get_children(self):
        return self.children

    def _ready(self):
        """Called when node enters scene"""
        pass

    def _process(self, delta):
        """Called every frame"""
        pass
```

Node System Example

```
class Sprite(Node):
    def __init__(self, name, texture):
        super().__init__(name)
        self.texture = texture
        self.position = [0, 0]

    def move(self, x, y):
        self.position[0] += x
        self.position[1] += y

class Player(Sprite):
    def __init__(self):
        super().__init__("Player", "player.png")
        self.health = 100
        self.score = 0

    def _process(self, delta):
        # Game logic here
        pass
```

Particle System Example

```
class Particle:
    def __init__(self, x, y, velocity_x, velocity_y):
        self.x = x
        self.y = y
        self.velocity_x = velocity_x
        self.velocity_y = velocity_y
        self.lifetime = 1.0

    def update(self, delta):
        self.x += self.velocity_x * delta
        self.y += self.velocity_y * delta
        self.lifetime -= delta

    def is_alive(self):
        return self.lifetime > 0
```

Particle Emitter

```
import random

class ParticleEmitter:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.particles = []

    def emit(self, count):
        for _ in range(count):
            vx = random.uniform(-100, 100)
            vy = random.uniform(-100, 100)
            particle = Particle(self.x, self.y, vx, vy)
            self.particles.append(particle)

    def update(self, delta):
        for particle in self.particles[:]:
            particle.update(delta)
            if not particle.is_alive():
                self.particles.remove(particle)
```

Practical: UN SDG Game Objects

```
class SDGChallenge:
    def __init__(self, goal_number, title, description):
        self.goal_number = goal_number
        self.title = title
        self.description = description
        self.completed = False
        self.score = 0

    def attempt(self, player_solution):
        """Override in subclasses"""
        pass

    def complete(self):
        self.completed = True
        return f"SDG {self.goal_number}: {self.title} - COMPLETED!"

class CleanWaterChallenge(SDGChallenge):
    def __init__(self):
        super().__init__(6, "Clean Water",
                         "Build a water filtration system")
```

Best Practices

1. **Single Responsibility**: Each class should have one clear purpose
2. **DRY (Don't Repeat Yourself)**: Use inheritance and composition
3. **Encapsulation**: Keep data private, expose through methods
4. **Clear naming**: Use descriptive class and method names
5. **Documentation**: Use docstrings to explain your classes

```
class Tune:
    """Represents an Irish traditional music tune.

    Attributes:
        title: The name of the tune
        tune_type: The type (reel, jig, etc.)
        key: The musical key
    """
```

Common Pitfalls to Avoid

❌ **Don't:** Make everything public

✅ **Do:** Use private attributes with getters/setters

❌ **Don't:** Create god classes (classes that do too much)

✅ **Do:** Follow single responsibility principle

❌ **Don't:** Deep inheritance hierarchies

✅ **Do:** Prefer composition over deep inheritance

❌ **Don't:** Ignore the `__init__` method

✅ **Do:** Initialize all attributes properly

Practice Exercise

Create a music production system with:

- `Track` class (parent)
- `AudioTrack` and `MIDITrack` (children)
- `Effect` class with subclasses (Reverb, EQ, Compressor)
- Methods to apply effects to tracks
- A `MixerChannel` that can hold multiple tracks

Challenge: Implement a method to export the final mix!

Summary

Key Takeaways:

- Classes are blueprints, objects are instances
- Encapsulation protects data and controls access
- Inheritance creates hierarchies and code reuse
- Polymorphism allows flexible, extensible code
- Use OOP to organize complex systems

Next Steps:

- Practice with real projects
- Read Python documentation
- Explore design patterns

Questions?

Resources:

- Python Official Docs: docs.python.org
- Real Python: realpython.com
- Godot Python: godot-python.readthedocs.io

Contact: bryan.duggan@tudublin.ie