Python Classes and Inheritance

Part 1: Understanding Classes

What is a Class?

A **class** is a blueprint for creating objects in Python. Think of it as a template that defines what attributes (data) and methods (functions) an object will have. Classes allow you to organize related data and functionality together, making your code more modular and reusable.

Class Example

Here's a simple example of a class definition:

```
class Item:
    def __init__(self):
        self.name = ""
        self.quantity = 0

    def set_name(self, name):
        self.name = name

    def set_quantity(self, quantity):
        self.quantity = quantity

    def display(self):
        print(f"Item: {self.name}, Quantity: {self.quantity}")
```

Key components:

- `__init__()`: The constructor method that initializes new instances
- `self`: A reference to the instance itself (automatically passed as the first argument)
- **Instance attributes**: `name` and `quantity` store data specific to each instance
- **Methods**: Functions that operate on the instance's data

Using a Class

```
# Create an instance
item1 = Item()

# Set attributes
item1.set_name("Smith Cereal")
item1.set_quantity(9)

# Call methods
item1.display()  # Output: Item: Smith Cereal, Quantity: 9
```

Why Use Classes?

Classes help you:

- **Organize code** by grouping related data and functions

- **Reuse code** by creating multiple instances from one template
- **Model real-world concepts** in your programs
- **Extend functionality** through inheritance (covered next!)

Part 2: Inheritance - Extending Classes

What is Inheritance?

Inheritance allows you to create a new class based on an existing class. The new class (called a **derived class** or **subclass**) automatically gains all the attributes and methods of the original class (called the **base class** or **superclass**), and you can add or modify functionality as needed.

Think of it this way: If you're building a store inventory system, you might have a general `Item` class. But fruits and vegetables need an expiration date—something regular items don't have. Rather than duplicating all the `Item` code, you can create a `Produce` class that *inherits* from `Item` and adds the expiration date feature.

Creating a Derived Class

Here's how to create a class that inherits from another:

```
class Item:
    def __init__(self):
        self.name = ""
        self.quantity = 0

    def set_name(self, name):
        self.name = name

    def set_quantity(self, quantity):
        self.quantity = quantity

    def display(self):
        print(f"Item: {self.name}, Quantity: {self.quantity}")


class Produce(Item):  # Produce inherits from Item
    def __init__(self):
        Item.__init__(self)  # Call the base class constructor
        self.expiration = ""

    def set_expiration(self, expiration):
        self.expiration = expiration

    def get_expiration(self):
        return self.expiration
```

Key syntax: `class Produce(Item):`
The base class name appears in parentheses after the derived class name.

Using the Derived Class

```
# Create a Produce instance
item2 = Produce()
```

```
# Use methods from both Item and Produce
item2.set_name("Apples")            # From Item
item2.set_quantity(40)               # From Item
item2.set_expiration("May 5, 2012") # From Produce

# Call inherited method
item2.display()  # Output: Item: Apples, Quantity: 40

# Call Produce-specific method
print(f"Expires: {item2.get_expiration()}")  # Output:
Expires: May 5, 2012
```

What the Derived Class Inherits

When you create a `Produce` instance, it automatically has:

- **All class attributes** from `Item` (methods like `set_name()`, `set_quantity()`, `display()`)
- **All attributes defined in `Produce`** (methods like `set_expiration()`, `get_expiration()`)
- **Instance attributes** from both classes (once constructors are called)

Understanding the Constructor Call

Notice this line in `Produce.__init__()`:

```
Item.__init__(self)
```

This **explicitly calls the base class constructor**, which is essential for creating the `name` and `quantity` attributes in your `Produce` instance.

Here's what happens when you create a `Produce` instance:

1. `Produce.__init__(self)` is called
2. It immediately calls `Item.__init__(self)`
3. The `Item` constructor creates `name` and `quantity` attributes
4. Control returns to `Produce.__init__()`
5. The `expiration` attribute is created

Important: Class attributes (methods) are automatically inherited, but instance attributes must be created by calling the base class constructor.

Inheritance Terminology

- **Base class (or superclass):** The class being inherited from (e.g., `Item`)
- **Derived class (or subclass):** The class that inherits (e.g., `Produce`)
- **Inheritance:** The mechanism by which a derived class gains attributes from its base class
- **Inheritance tree:** The hierarchy of classes from derived to base classes

Any class can serve as a base class—no special modifications are needed to its definition.

Part 3: Advanced Inheritance Concepts

Accessing Base Class Attributes

A derived class can access all base class attributes through normal attribute reference operations. When you reference an attribute (like `item2.set_name()`), Python searches for it using this order:

6. **Instance's namespace** (instance attributes)
7. **Class's namespace** (class attributes and methods)
8. **Base classes' namespaces** (recursively up the inheritance tree)

Example inheritance tree:

```
class TransportMode:
    def move(self):
        print("Moving...")

class MotorVehicle(TransportMode):
    def start_engine(self):
        print("Engine started")

class Motorcycle(MotorVehicle):
    def wheelie(self):
        print("Doing a wheelie!")

# Create instance
bike = Motorcycle()

# Python searches up the inheritance tree
bike.wheelie()        # Found in Motorcycle
bike.start_engine()   # Found in MotorVehicle
bike.move()           # Found in TransportMode (top of tree)
```

The search continues all the way up the inheritance tree until the attribute is found or a `AttributeError` is raised.

Overriding Methods

A derived class can define a method with the **same name** as a method in the base class. This is called **method overriding**, and the derived class's method will replace (override) the base class method.

Example:

```
class Item:
    def __init__(self):
        self.name = ""
        self.quantity = 0

    def display(self):
        print(f"Item: {self.name}, Quantity: {self.quantity}")


class Produce(Item):
    def __init__(self):
        Item.__init__(self)
```

```
        self.expiration = ""

    def display(self):  # Override Item's display()
        print(f"Produce: {self.name}, Quantity:
{self.quantity}, Expires: {self.expiration}")
```

Using the overridden method:

```
item1 = Item()
item1.name = "Cereal"
item1.quantity = 10
item1.display()  # Output: Item: Cereal, Quantity: 10

item2 = Produce()
item2.name = "Bananas"
item2.quantity = 25
item2.expiration = "June 1, 2023"
item2.display()  # Output: Produce: Bananas, Quantity: 25,
Expires: June 1, 2023
```

Why does this work?

When Python looks up the `display()` method, it finds it in the `Produce` class's namespace first, so it uses that version instead of looking further up the inheritance tree.

Extending (Not Replacing) Base Class Methods

Often you want to **extend** the base class method rather than completely replace it. You can call the base class method explicitly and then add additional functionality:

```
class Produce(Item):
    def __init__(self):
        Item.__init__(self)
        self.expiration = ""

    def display(self):
        Item.display(self)  # Call the base class method first
        print(f"Expiration: {self.expiration}")  # Add
additional behavior
```

Result:

```
item2 = Produce()
item2.name = "Strawberries"
item2.quantity = 15
item2.expiration = "April 20, 2023"
item2.display()
# Output:
# Item: Strawberries, Quantity: 15
# Expiration: April 20, 2023
```

This pattern lets you reuse the base class logic while adding specialized behavior.

Inheritance vs. Composition: Is-a vs. Has-a

It's important to understand when to use inheritance versus composition.

Inheritance (Is-a relationship): Use inheritance when the derived class **is a type of** the base class.

- A `Produce` **is an** `Item` ✓
- A `Motorcycle` **is a** `MotorVehicle` ✓
- A `Child` **is a** `Person` ✓

Composition (Has-a relationship): Use composition when an object **is made up of** other objects.

- A `Mother` **has a** `name` (string object)
- A `Mother` **has** `children` (list of Child objects)
- A `Car` **has an** `engine` (Engine object)

Example of composition (not inheritance):

```
class Mother:
    def __init__(self, name):
        self.name = name  # Has-a name (string)
        self.children = []  # Has-a list of children

    def add_child(self, child):
        self.children.append(child)
```

Example using both concepts:

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

class Mother(Person):  # Is-a Person (inheritance)
    def __init__(self, name, birthdate):
        Person.__init__(self, name, birthdate)
        self.children = []  # Has-a list of children
(composition)

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

class Child(Person):  # Is-a Person (inheritance)
    def __init__(self, name, birthdate):
        Person.__init__(self, name, birthdate)
```

Rule of thumb: If you can say "X is a Y," use inheritance. If you say "X has a Y," use composition.

Variations in Class Derivation

Inheritance offers several flexible patterns:

1. Chain of inheritance (a derived class can be a base class):

```
class Item:
    pass

class Produce(Item):
    pass
```

```
class Fruit(Produce):  # Fruit inherits from Produce, which
inherits from Item
    pass
```

2. Multiple derived classes from one base:

```
class Item:
    pass

class Produce(Item):
    pass

class Book(Item):  # Both Produce and Book inherit from Item
    pass
```

3. Multiple inheritance (inheriting from multiple base classes):

```
class Dwelling:
    pass

class Property:
    pass

class House(Dwelling, Property):  # Inherits from both
Dwelling and Property
    pass
```

Part 4: Multiple Inheritance and Mixins

Multiple Inheritance

A class can inherit from **more than one base class** simultaneously. The derived class inherits all class attributes and methods from every base class.

Syntax:

```
class VampireBat(WingedAnimal, Mammal):  # Inherits from both
classes
    def __init__(self):
        WingedAnimal.__init__(self)
        Mammal.__init__(self)
        # Additional initialization
```

The derived class has access to all methods and attributes from both `WingedAnimal` and `Mammal`.

Mixin Classes

A common and powerful use of multiple inheritance is **mixins**. A **mixin** is a class that:

- Provides additional behavior (methods)
- Is meant to be inherited from, **not instantiated directly**
- "Mixes in" new functionality to other classes

Example:

```
class LoggerMixin:
    """Mixin that adds logging capability to any class"""
    def log(self, message):
        print(f"[{self.__class__.__name__}] {message}")
```

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

class LoggedItem(Item, LoggerMixin):  # Mix in logging
functionality
    def __init__(self, name):
        Item.__init__(self, name)
        self.log(f"Created item: {name}")

# Usage
item = LoggedItem("Laptop")  # Output: [LoggedItem] Created
item: Laptop
item.log("Item updated")      # Output: [LoggedItem] Item
updated
```

Mixins let you add optional functionality to classes without creating complex inheritance hierarchies.

Additional Notes

Python-Specific Considerations

Getter/Setter Pattern: The examples use methods like `set_name()` and `get_expiration()` for clarity. In real Python code, you typically access attributes directly:

```
# Instead of:
item1.set_name("Hot Pockets")

# Use:
item1.name = "Hot Pockets"
```

For more control, use Python's `@property` decorator.

Access Levels: In languages like Java and C++, you explicitly set attributes as public, private, or protected. In Python, **all attributes are public by default**. Python does support a form of private variables using name mangling with double underscores (e.g., `self.__data`), but this is primarily used to prevent name collisions in complex inheritance trees, not for strict information hiding.

UML Notation: If you see symbols like +, –, or # in class diagrams:

- + = public (accessible by anyone)
- – = private (accessible only by the class itself)
- # = protected (accessible by the class and derived classes)

Summary

Classes provide a way to bundle data and functionality together. **Inheritance** allows you to:

- Reuse existing code by deriving new classes from existing ones
- Create specialized versions of general classes
- Organize code into logical hierarchies
- Override or extend base class behavior

Key takeaways:

- Derived classes inherit all class attributes from base classes
- Use `BaseClass.__init__(self)` to initialize inherited instance attributes
- Method overriding lets you customize inherited behavior
- Use inheritance for "is-a" relationships and composition for "has-a" relationships
- Multiple inheritance and mixins provide flexible ways to combine functionality