Object-Oriented Programming in Python

Martin Wafula
Multimedia University of Kenya

April 23, 2025

Outline

Introduction to OOP

What is Object-Oriented Programming?

- A programming paradigm based on the concept of "objects"
- Objects contain:
  - **Data** (attributes/properties) - information the object stores
  - **Behavior** (methods/functions) - actions the object can perform
- Four fundamental principles:
  - **Encapsulation**: Bundling data with methods that operate on that data
  - **Abstraction**: Hiding complex implementation details
  - **Inheritance**: Creating hierarchical relationships between classes
  - **Polymorphism**: One interface with multiple implementations

Why Use OOP?

- **Modularity**: Self-contained objects make code organization easier
- **Reusability**: Inheritance promotes code reuse (don't repeat yourself)
- **Maintainability**: Clear structure makes changes easier
- **Scalability**: Better for large, complex systems
- **Flexibility**: Polymorphism enables adaptable code
- **Problem Solving**: Models real-world entities effectively

Real-world Analogy

Think of objects like real-world things. A car has properties (color, model) and behaviors (drive, brake). OOP lets us model programs this way.

Everything is an Object in Python

Python's Object-Oriented Nature

```python
# Even primitive types are objects
num = 42
print(type(num))  # <class 'int'>
# Lists are objects with methods
my_list = [1, 2, 3]
my_list.append(4)  # Calling a method
# Functions are objects
def greet():
    return "Hello"
print(type(greet))  # <class 'function'>
```

- Unlike some languages, Python treats everything as an object
- This means everything has attributes and methods
- Even basic operations like + are method calls behind the scenes

Classes and Objects

Basic Class Structure

```python
class ClassName:
    """Class docstring"""
    class_attribute = value  # Shared by all instances
    def __init__(self, parameters):
        """Initialize instance attributes"""
        self.instance_attribute = value
    def method(self, parameters):
        """Method docstring"""
        # Method implementation
```

Key components:

- `class` keyword - starts the class definition
- Class name (PascalCase convention) - e.g., MyClass
- Docstring - describes what the class does
- __init__ method (constructor) - initializes new objects
- Instance methods - define object behavior

The Account Class Example

```python
from decimal import Decimal

class Account:
    """Account class for maintaining bank balances"""
    def __init__(self, name, balance):
        """Initialize account with name and balance"""
        if balance < Decimal('0.00'):
            raise ValueError('Initial balance must be >= 0.00')
        self.name = name
        self.balance = balance
    def deposit(self, amount):
        """Add money to the account"""
        if amount < Decimal('0.00'):
            raise ValueError('Deposit amount must be positive')
        self.balance += amount
```

Explanation

- Uses Decimal for precise financial calculations
- ___init___ validates initial balance
- deposit method validates deposit amount

Constructor Expression

```python
from account import Account
from decimal import Decimal

# Create an Account object
account1 = Account('John Green', Decimal('50.00'))

# Access attributes
print(account1.name)      # 'John Green'
print(account1.balance)   # Decimal('50.00')

# Call methods
account1.deposit(Decimal('25.53'))
print(account1.balance)   # Decimal('75.53')
```

Key points

- Objects are created by calling the class name like a function
- __init__ is automatically called to initialize the object
- Methods are called using dot notation (object.method())
- Attributes are accessed using dot notation (object.attribute)

Understanding self

- `self` is a reference to the current object instance
- It's automatically passed as the first argument to methods
- Used to access instance attributes and other methods
- Example:
  - When you call `account1.deposit(25.53)`, Python actually calls `deposit(account1, 25.53)`
- Why self?
  - Makes it clear you're working with instance data
  - Allows multiple objects of the same class to maintain their own state

Important

You must include `self` as the first parameter in all instance methods (but don't pass it when calling the method)

Attributes and Properties

Instance vs Class Attributes

Instance Attributes

- Unique to each object instance
- Created in __init__ using self.attribute_name
- Each object has its own copy

```python
class MyClass:
    def __init__(self, value):
        self.instance_attr = value  # Unique to each instance

obj1 = MyClass(1)
obj2 = MyClass(2)
print(obj1.instance_attr)  # 1
print(obj2.instance_attr)  # 2
```

Class Attributes

- Shared by all instances of the class
- Defined directly in the class (not in methods)
- Useful for constants or shared data

```python
class MyClass:
    class_attr = 0  # Shared by all instances
print(MyClass.class_attr)  # 0
obj1 = MyClass()
obj2 = MyClass()
print(obj1.class_attr)  # 0
print(obj2.class_attr)  # 0
```

Caution

If you modify a class attribute through an instance, you actually create an instance attribute that shadows the class attribute!

Attribute Access Control

- Python doesn't have true private attributes like some languages
- Convention: prefix with _ for "internal use"

```
self._internal_attr = value  # Hint: don't access directly
```

- Name mangling: __name becomes _ClassName__name
  - Python changes the name to make it harder to access accidentally
  - Not truly private, just harder to access

```
self.__private_attr = value  # Becomes _MyClass__private_attr
```

- Properties provide controlled access to attributes
  - Allow validation when getting/setting values
  - Maintain consistent interface if implementation changes

Properties Example


```python
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour   # Uses hour property setter
        self.minute = minute
        self.second = second

    @property
    def hour(self):
        """Get the hour"""
        return self._hour

    @hour.setter
    def hour(self, hour):
        """Set the hour with validation"""
        if not (0 <= hour < 24):
            raise ValueError(f'Hour ({hour}) must be 0-23')
        self._hour = hour
    # Similar properties for minute and second
```

Properties Example [**cont...**]

Explanations

- @property defines a getter method
- @attribute.setter defines a setter method
- Allows validation when setting values

Property Access

```python
t = Time(hour=12, minute=30, second=45)
# Getting values (calls @property)
print(t.hour)  # 12
# Setting values (calls @hour.setter)
t.hour = 15
print(t.hour)  # 15
# Invalid assignment raises ValueError
t.hour = 25  # ValueError: Hour (25) must be 0-23
```

Benefits of Properties

- Validation when setting values
- Computed attributes (calculate values on the fly)
- Maintains consistent interface if implementation changes
- Allows you to add behavior to attribute access

Inheritance

Inheritance Basics

Syntax

```python
class BaseClass:
    # Base class implementation
class DerivedClass(BaseClass):
    # Derived class implementation
```

Key Concepts

- "is-a" relationship (DerivedClass is a BaseClass)
- Subclass inherits all attributes and methods
- Can override or extend base class functionality
- `super()` accesses base class methods

Real-world Example

A `Car` is a `Vehicle`. A `Student` is a `Person`. Inheritance models these "is-a" relationships.

CommissionEmployee Hierarchy


```python
class CommissionEmployee:
    """Base class for commission employees"""
    def __init__(self, first_name, last_name, ssn,
                 gross_sales, commission_rate):
        self._first_name = first_name
        self._last_name = last_name
        self._ssn = ssn
        self.gross_sales = gross_sales
        self.commission_rate = commission_rate
    def earnings(self):
        return self.gross_sales * self.commission_rate
    def __repr__(self):
        return (f'CommissionEmployee: {self.first_name}'
                f'{self.last_name}\nssn: {self.ssn}\n'
                f'gross sales: {self.gross_sales:.2f}\n'
                f'commission rate: {self.commission_rate
    :.2f}')
```

- Base class for employees paid by commission
- Contains common attributes and methods

SalariedCommissionEmployee Subclass

```python
class SalariedCommissionEmployee(CommissionEmployee):
    """Employee with salary plus commission"""
    def __init__(self, first_name, last_name, ssn,
                 gross_sales, commission_rate,
    base_salary):
        super().__init__(first_name, last_name, ssn,
                         gross_sales, commission_rate)
        self.base_salary = base_salary
    def earnings(self):
        return super().earnings() + self.base_salary
    def __repr__(self):
        return ('Salaried' + super().__repr__()+
                f'\nbase salary:{self.base_salary:.2f}')
```

- Inherits from CommissionEmployee
- Adds base_salary attribute
- Overrides earnings() to include base salary
- Uses super() to call parent class methods

Using the Hierarchy

Creating and Using Objects

```python
from decimal import Decimal
# Base class object
ce = CommissionEmployee('Sue', 'Jones', '333-33-3333',
    Decimal('10000.00'), Decimal('0.06'))
print(ce.earnings())  # 600.00
# Subclass object
sce = SalariedCommissionEmployee('Bob', 'Lewis', '
    444-44-4444',Decimal('5000.00'), Decimal('0.04'),
    Decimal('300.00'))
print(sce.earnings())  # 500.00
```

Key points:

- Subclass inherits all base class attributes/methods
- Can override methods (earnings, __repr__)
- `super()` calls base class implementation

Common Relationships

- **CommunityMember** hierarchy (single inheritance)
- **Shape** hierarchy (multiple levels)
- "is-a" vs "has-a" (inheritance vs composition)

- Single inheritance: class inherits from one base class
- Multiple inheritance: class inherits from multiple base classes (advanced)
- Composition: class contains objects of other classes ("has-a" relationship)

Polymorphism

Polymorphism Concepts

- **Definition**: Ability to present the same interface for different underlying forms
- Two forms in Python:
  - **Inheritance-based**: Method overriding in subclasses
  - **Duck typing**: Objects with compatible interfaces
- Enables "programming in the general"
- Makes systems extensible

Real-world Analogy

The "drive" operation works differently for cars, trucks, and motorcycles, but they all can be "driven". Polymorphism lets us treat them all as "drivable" objects.

Inheritance-Based Polymorphism


```python
employees = [
    CommissionEmployee('Sue', 'Jones', '333-33-3333',
                       Decimal('10000.00'), Decimal('0.06')),
    SalariedCommissionEmployee('Bob', 'Lewis', '444-44-4444', Decimal('5000.00'), Decimal('0.04'), Decimal('300.00'))]
for employee in employees:
    print(employee)
    print(f'Earnings: {employee.earnings():.2f}\n')
```

Output:

- Different string representations
- Different earnings calculations
- Same interface (earnings() method)
- We can process different types uniformly
- Each object responds appropriately to the same method call

Duck Typing

Definition

"If it walks like a duck and quacks like a duck, it must be a duck"

```python
class WellPaidDuck:
    def __repr__(self):
        return 'I am a well-paid duck'
    def earnings(self):
        return Decimal('1_000_000.00')
# Works with same interface
employees.append(WellPaidDuck())
for employee in employees:
    print(employee)
    print(f'{employee.earnings():.2f}\n')
```

- Python doesn't care about inheritance if the interface matches
- Any object with earnings() can be used polymorphically
- More flexible than inheritance-based polymorphism

Polymorphism Benefits

- **Flexibility**: New types can be added without changing existing code
- **Maintainability**: Changes are localized
- **Extensibility**: Easy to add new functionality
- **Readability**: Code expresses intent clearly
- **Reusability**: Generic algorithms work with many types

Key Point

Polymorphism lets you write code that works with objects at a higher level of abstraction, making your code more general and reusable.

Special Methods

Method	Purpose
__init__	Object initialization
__repr__	Official string representation
__str__	Informal string representation
__format__	Custom string formatting
__eq__	== operator
__lt__, __le__	<, <= operators
__gt__, __ge__	>, >= operators
__add__	+ operator
__sub__	- operator
__iadd__	+= operator
__isub__	-= operator

- Special methods start and end with double underscores
- They enable operator overloading and other Python features
- Called automatically by Python in specific situations

Complex Number Class


```python
class Complex:
    """Complex number with overloaded operators"""
    def __init__(self, real, imaginary):
        self.real = real
        self.imaginary = imaginary
    def __add__(self, right):
        """Overload + operator"""
        return Complex(self.real + right.real,
                       self.imaginary + right.imaginary)
    def __iadd__(self, right):
        """Overload += operator"""
        self.real += right.real
        self.imaginary += right.imaginary
        return self
    def __repr__(self):
        """Return string representation"""
        return (f'({self.real} ' +
                ('+' if self.imaginary >= 0 else '-') +
                f' {abs(self.imaginary)}i)')
```

Complex Number Operations

```python
x = Complex(2, 4)
y = Complex(5, -1)
print(x)          # (2 + 4i)
print(y)          # (5 - 1i)
z = x + y         # Calls __add__
print(z)          # (7 + 3i)

x += y            # Calls __iadd__
print(x)          # (7 + 3i)
```

Benefits:

- Natural syntax for mathematical operations
- Consistent with built-in types
- Makes classes more intuitive to use

String Representation Methods

__repr__ vs __str__

- __repr__: Official, unambiguous representation
  - Used by repr() function
  - Should look like constructor call
  - Fallback for __str__ if not defined
- __str__: Informal, readable representation
  - Used by str() and print()
  - More user-friendly

```python
class Card:
    def __repr__(self):
        return f"Card(face='{self.face}', suit='{self.suit}')"
    def __str__(self):
        return f'{self.face} of {self.suit}'
    def __format__(self, format_spec):
        return f'{str(self):{format_spec}}'
```

Advanced Topics

Data Classes (Python 3.7+)

Key Features

- Autogenerate __init__, __repr__, __eq__
- Concise syntax with type hints
- Default values &type annotations
- Frozen (immutable) instances

Data Classes (Python 3.7+)

```python
from dataclasses import dataclass
from typing import ClassVar, List
@dataclass
class Card:
    FACES: ClassVar[List[str]] = ['Ace', '2', '3', ...,
    'King']
    SUITS: ClassVar[List[str]] = ['Hearts', 'Diamonds',
    ...]
    face: str
    suit: str

    @property
    def image_name(self):
        return f'{self.face_of_{self.suit}.png'
```

- @dataclass decorator does the magic
- Class attributes use ClassVar
- Data attributes use type hints

Data Class Example

```
from carddataclass import Card
# Autogenerated __init__
c1 = Card(Card.FACES[0], Card.SUITS[3])  # Ace of Spades

# Autogenerated __repr__
print(c1)  # Card(face='Ace', suit='Spades')
# Autogenerated __eq__
c2 = Card(Card.FACES[0], Card.SUITS[3])
print(c1 == c2)  # True
# Custom property
print(c1.image_name)  # 'Ace_of_Spades.png'
```

- Less boilerplate code
- Clear, concise syntax
- Still full Python classes with all capabilities

Data Class Advantages

- **Less Boilerplate**: Autogenerated methods
- **Type Hints**: Better documentation and IDE support
- **Immutability**: Frozen instances
- **Flexibility**: Can add methods and properties
- **Interoperability**: Works with existing code
- **Performance**: Optimized attribute access

When to Use Data Classes

- When you need a class mainly to store data
- When you want automatic string representations
- When you want to reduce repetitive code

```python
def maximum(value1, value2, value3):
    """Return the maximum of three values.
    >>> maximum(3, 1, 2)
    3
    >>> maximum(1.5, 2.5, 1.0)
    2.5
    >>> maximum('a', 'c', 'b')
    'c'
    """
    max_value = value1
    if value2 > max_value:
        max_value = value2
    if value3 > max_value:
        max_value = value3
    return max_value

if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

Unit Testing with doctest

- Tests in docstrings
- »> followed by expected output
- doctest.testmod() runs the tests
- Great for simple unit tests

- **LEGB Rule**:
  - Local - inside current function
  - Enclosing - for nested functions
  - Global - module level
  - Built-in - Python built-ins
- **Class Namespace**: Contains class attributes
- **Object Namespace**: Contains instance attributes
- **Module Namespace**: Global to the module
- **Built-in Namespace**: Python's built-in names

Key Point

Python searches namespaces in LEGB order when looking up names

Key Steps

```python
import pandas as pd
from scipy import stats
import seaborn as sns
# Load data
nyc = pd.read_csv('ave_hi_nyc_jan_1895-2018.csv')
nyc.columns = ['Date', 'Temperature', 'Anomaly']
# Clean data
nyc.Date = nyc.Date.floordiv(100)
# Calculate regression
slope, intercept = stats.linregress(
    x=nyc.Date, y=nyc.Temperature)[:2]
# Predict future temperature
predicted = slope * 2019 + intercept
```

- Uses pandas for data handling
- Uses scipy.stats for regression
- Uses seaborn for visualization

- Uses pandas for data handling
- Uses scipy.stats for regression
- Uses seaborn for visualization

- OOP is a powerful paradigm for organizing complex programs
- Python provides flexible and expressive OOP capabilities
- Key concepts: encapsulation, inheritance, polymorphism
- Special methods enable operator overloading
- Advanced features like data classes reduce boilerplate
- Proper design leads to maintainable, extensible code

Next Steps

- Practice creating your own classes
- Experiment with inheritance hierarchies
- Try implementing operator overloading
- Explore data classes for simple data containers

Further Reading

- Python Documentation: Classes
- "Fluent Python" by Luciano Ramalho
- "Python Cookbook" by David Beazley and Brian K. Jones
- "Design Patterns in Python" by Peter Ullrich
- Python Standard Library: dataclasses, abc, enum

Online Resources

- Real Python OOP Tutorials
- Python Official Documentation
- Stack Overflow for specific questions

Thank You!
Questions?