Object-Oriented Programming in Python

Complete Reference Guide

A Comprehensive Guide to OOP Concepts,
Classes, Inheritance, and Design Patterns in Python

Sujil S

sujil9480@gmail.com

December 25, 2025

Contents

1 Introduction to Object-Oriented Programming

1.1 What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects" that contain both data (attributes) and code (methods). OOP allows developers to create modular, reusable, and organized code.

1.1.1 Key Principles of OOP

1. **Encapsulation**: Bundling data and methods that operate on that data within a single unit (class)
2. **Inheritance**: Creating new classes from existing ones, inheriting attributes and methods
3. **Polymorphism**: Ability to present the same interface for different data types
4. **Abstraction**: Hiding complex implementation details and showing only necessary features

1.2 OOP vs Structured Programming

Object-Oriented Programming	**Structural Programming**
Based on objects containing data and methods	Based on functions and procedures
Bottom-up approach	Top-down approach
Provides data hiding through encapsulation	Does not provide data hiding
Can solve problems of any complexity	Can solve moderate problems
Code reusability through inheritance	Limited code reusability
More flexible and maintainable	Less flexible

1.3 Why Use OOP?

- **Modularity**: Code is organized into objects, making it easier to understand
- **Reusability**: Classes can be reused across programs
- **Maintainability**: Changes in one part don't affect other parts
- **Security**: Data hiding protects internal state
- **Scalability**: Easy to add new features without breaking existing code

2 Classes and Objects

2.1 Understanding Classes and Objects

- **Class**: A blueprint or template for creating objects. It defines attributes and methods.
- **Object**: An instance of a class. It's a concrete entity based on the class blueprint.

Important Note

Think of a class as a cookie cutter and objects as the cookies made from it. The class defines the shape and structure, while each object is a unique instance.

2.2 Creating a Class

2.2.1 Basic Class Syntax

```
class ClassName:
    # class body
    pass
```

Naming Convention: Use PascalCase (each word capitalized) for class names.

Simple Class Example

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

    def greet(self):
        print(f"Hello, my name is {self.name}")

# Creating objects
person1 = Person("John", 30)
person2 = Person("Alice", 25)

person1.greet()  # Output: Hello, my name is John
person2.greet()  # Output: Hello, my name is Alice
```

2.3 The Constructor: __init__()

The `__init__()` method is a special method called a **constructor**. It's automatically called when an object is created.

2.3.1 Types of Constructors

1. **Default Constructor**: No parameters (except self)

```
class Example:
    def __init__(self):
        self.value = 0

```

2. **Parameterized Constructor**: Accepts parameters

```python
class Example:
    def __init__(self, value):
        self.value = value

```

2.4 The self Parameter

self represents the instance of the class. It's used to access variables and methods of the class.

Important Note

- `self` is NOT a keyword; it's a convention (you can use any name, but don't!)
- `self` must be the first parameter in instance methods
- When calling methods, Python automatically passes the instance as `self`

Understanding self

```python
class Atm:
    def __init__(self):
        print(f"Object created at: {id(self)}")
        self.pin = ''
        self.balance = 0

obj1 = Atm()  # Object created at: 140289660099024
print(id(obj1))  # 140289660099024 (same as self)

obj2 = Atm()  # Object created at: 140289660586384
print(id(obj2))  # 140289660586384 (different object)
```

3 Attributes and Methods

3.1 Instance Attributes

Instance attributes are variables that belong to a specific object. Each object has its own copy.

```
class Person:
    def __init__(self, name, country):
        self.name = name        # instance attribute
        self.country = country  # instance attribute

p1 = Person('Nitish', 'India')
p2 = Person('Steve', 'Australia')

print(p1.name)      # Nitish
print(p2.name)      # Steve
print(p1.country)   # India
print(p2.country)   # Australia
```

3.2 Class Attributes (Static Variables)

Class attributes are shared by all instances of the class. They're defined directly in the class body.

Class Attributes Example

```
class Atm:
    # Class attribute (shared by all instances)
    __counter = 1

    def __init__(self):
        self.pin = ''
        self.balance = 0
        # Use class attribute to assign unique ID
        self.cid = Atm.__counter
        Atm.__counter += 1

    @staticmethod
    def get_counter():
        return Atm.__counter

c1 = Atm()
print(c1.cid)  # 1

c2 = Atm()
print(c2.cid)  # 2

c3 = Atm()
print(c3.cid)  # 3

print(Atm.get_counter())  # 4
```

3.3 Instance Attributes vs Class Attributes

Instance Attributes	**Class Attributes**
Unique to each object	Shared by all objects
Defined in __init__() method	Defined in class body
Accessed via self.attribute	Accessed via ClassName.attribute
Different values for each instance	Same value for all instances
Use for object-specific data	Use for shared data

3.4 Creating Attributes from Outside the Class

Python allows dynamic attribute creation (though not recommended for production code).

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

p = Person("John")
print(p.name)  # John

# Creating new attribute from outside
p.gender = 'male'
print(p.gender)  # male

# This attribute exists only for this instance
p2 = Person("Alice")
# print(p2.gender)  # AttributeError!
```

4 Methods in Python

4.1 Types of Methods

Python has three types of methods:

1. **Instance Methods**: Operate on instance data (require `self`)
2. **Class Methods**: Operate on class data (use `@classmethod` decorator)
3. **Static Methods**: Don't operate on instance or class data (use `@staticmethod`)

4.2 Instance Methods

Instance methods are the most common type. They can access and modify instance and class attributes.

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

    # Instance method
    def greet(self):
        print(f"Hello, I'm {self.name}")

    # Instance method with parameters
    def birthday(self):
        self.age += 1
        print(f"Happy birthday! Now {self.age} years old")

p = Person("John", 30)
p.greet()       # Hello, I'm John
p.birthday()    # Happy birthday! Now 31 years old
```

4.3 Class Methods

Class methods receive the class as the first argument (conventionally named `cls`).

Class Method Example

```
class Person:
    count = 0  # Class attribute

    def __init__(self, name):
        self.name = name
        Person.count += 1

    @classmethod
    def get_count(cls):
        """Class method - receives class as first argument"""
        return cls.count

    @classmethod
    def create_anonymous(cls):
        """Factory method - creates instance"""
        return cls("Anonymous")

```

```python
p1 = Person("John")
p2 = Person("Alice")
print(Person.get_count())  # 2

# Using factory method
p3 = Person.create_anonymous()
print(p3.name)  # Anonymous
print(Person.get_count())  # 3
```

4.4 Static Methods

Static methods don't receive `self` or `cls`. They're utility functions that belong to the class namespace.

Static Method Example

```python
class Atm:
    __water_source = "well in the circus"

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

    @staticmethod
    def get_water_source():
        """Static method - no self or cls parameter"""
        return Atm.__water_source

    @staticmethod
    def is_valid_pin(pin):
        """Utility function"""
        return len(str(pin)) == 4

# Can call without creating instance
print(Atm.get_water_source())  # well in the circus
print(Atm.is_valid_pin(1234))  # True
print(Atm.is_valid_pin(123))   # False
```

4.5 Method Comparison Table

Feature	Instance	Class	Static
First parameter	self	cls	None
Decorator	None	@classmethod	@staticmethod
Access instance data	Yes	No	No
Access class data	Yes	Yes	Yes (via ClassName)
Modify instance	Yes	No	No
Modify class	Yes	Yes	No
Call from class	No	Yes	Yes
Call from instance	Yes	Yes	Yes

4.6 When to Use Each Method Type

- **Instance Methods**: When you need to access/modify instance-specific data
- **Class Methods**: When you need to access/modify class-level data, or create factory methods
- **Static Methods**: When you need utility functions that don't need access to instance or class data

5 Encapsulation and Access Modifiers

5.1 What is Encapsulation?

Encapsulation is the bundling of data and methods that operate on that data within a single unit (class), and restricting direct access to some components.

5.1.1 Benefits of Encapsulation

- **Data Protection**: Prevents accidental modification of data
- **Flexibility**: Internal implementation can change without affecting external code
- **Maintainability**: Clear interface between object and outside world
- **Control**: Validation can be added when setting values

5.2 Access Modifiers in Python

Python uses naming conventions to indicate access levels:

1. **Public**: `attribute` (no underscore)
2. **Protected**: `_attribute` (single underscore)
3. **Private**: `__attribute` (double underscore)

5.3 Public Attributes

Public attributes can be accessed from anywhere.

```
class Person:
    def __init__(self, name):
        self.name = name  # Public attribute

p = Person("John")
print(p.name)  # Accessible
p.name = "Alice"  # Can be modified
```

5.4 Protected Attributes

Protected attributes (single underscore) are a convention indicating "use with caution." Python doesn't enforce this.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age  # Protected (by convention)

    def get_age(self):
        return self._age

p = Person("John", 30)
print(p.name)      # OK
print(p._age)      # Works, but shouldn't be used directly
print(p.get_age()) # Preferred way
```

5.5 Private Attributes (Name Mangling)

Private attributes (double underscore) undergo name mangling. Python changes the name internally.

Private Attributes with Name Mangling

```python
class Atm:
    def __init__(self):
        self.pin = ''
        self.__balance = 0  # Private attribute

    def get_balance(self):
        """Getter method for private attribute"""
        return self.__balance

    def set_balance(self, new_value):
        """Setter method with validation"""
        if type(new_value) == int and new_value >= 0:
            self.__balance = new_value
        else:
            print('Invalid balance amount')

obj = Atm()
# print(obj.__balance)  # AttributeError!

# Access through getter/setter
obj.set_balance(1000)
print(obj.get_balance())  # 1000

# Name mangling: __balance becomes _Atm__balance
print(obj._Atm__balance)  # 1000 (not recommended!)
```

5.6 Checking Attributes with dir()

The `dir()` function lists all attributes and methods of an object.

```python
class Test:
    def __init__(self):
        self.foo = 11       # Public
        self._bar = 23      # Protected
        self.__baz = 42     # Private

t = Test()
print(dir(t))

# Output includes:
# 'foo'           <- Public attribute (unchanged)
# '_bar'          <- Protected attribute (unchanged)
# '_Test__baz'    <- Private attribute (name mangled!)
```

5.7 Getters and Setters

Use getter and setter methods to control access to private attributes.

Complete Encapsulation Example

```python
class BankAccount:
    def __init__(self, account_number, balance=0):
        self.__account_number = account_number
        self.__balance = balance

    # Getter methods
    def get_account_number(self):
        return self.__account_number

    def get_balance(self):
        return self.__balance

    # Setter with validation
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited: ${amount}")
        else:
            print("Invalid amount")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrawn: ${amount}")
        else:
            print("Insufficient funds or invalid amount")

account = BankAccount("123456", 1000)
print(account.get_balance())  # 1000
account.deposit(500)          # Deposited: $500
print(account.get_balance())  # 1500
account.withdraw(200)         # Withdrawn: $200
print(account.get_balance())  # 1300
```

6 Magic Methods (Dunder Methods)

6.1 What are Magic Methods?

Magic methods (also called **dunder methods** - double underscore) are special methods with double underscores before and after their names. They enable operator overloading and provide special functionality.

6.2 Common Magic Methods

Method	Purpose	Example Usage
__init__	Constructor	Called when object created
__str__	String representation	print(obj)
__repr__	Developer representation	repr(obj)
__len__	Length	len(obj)
__add__	Addition	obj1 + obj2
__sub__	Subtraction	obj1 - obj2
__mul__	Multiplication	obj1 * obj2
__truediv__	Division	obj1 / obj2
__eq__	Equality	obj1 == obj2
__lt__	Less than	obj1 < obj2
__gt__	Greater than	obj1 > obj2
__call__	Make callable	obj()
__getitem__	Indexing	obj[key]
__setitem__	Assignment	obj[key] = value
__del__	Destructor	del obj

6.3 __str__() vs __repr__()

Both methods return string representations, but serve different purposes.

__str__ vs __repr__

```python
import datetime

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

    def __str__(self):
        """User-friendly representation"""
        return f"{self.name} ({self.age} years old)"

    def __repr__(self):
        """Developer-friendly representation"""
        return f"Person(name='{self.name}', age={self.age})"

p = Person("John", 30)
print(str(p))   # John (30 years old)
print(repr(p))  # Person(name='John', age=30)

# Built-in example
now = datetime.datetime.now()
```

```
print(str(now))    # 2024-12-16 15:46:52.007475
print(repr(now))   # datetime.datetime(2024, 12, 16, 15, 46, 52,
    7475)
```

Important Note

str vs repr in summary:

- `__str__()`: For end users - meant to be readable
- `__repr__()`: For developers - meant to be unambiguous and reproducible
- If only one is defined, Python uses it for both
- repr() should ideally return valid Python code to recreate the object

6.4 Operator Overloading

Operator overloading allows custom classes to use standard Python operators.

Fraction Class with Operator Overloading

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.num = numerator
        self.den = denominator

    def __str__(self):
        return f'{self.num}/{self.den}'

    def __add__(self, other):
        """Overload + operator"""
        new_num = self.num * other.den + other.num * self.den
        new_den = self.den * other.den
        return f'{new_num}/{new_den}'

    def __sub__(self, other):
        """Overload - operator"""
        new_num = self.num * other.den - other.num * self.den
        new_den = self.den * other.den
        return f'{new_num}/{new_den}'

    def __mul__(self, other):
        """Overload * operator"""
        new_num = self.num * other.num
        new_den = self.den * other.den
        return f'{new_num}/{new_den}'

    def __truediv__(self, other):
        """Overload / operator"""
        new_num = self.num * other.den
        new_den = self.den * other.num
        return f'{new_num}/{new_den}'

    def convert_to_decimal(self):
```

```
        """Convert fraction to decimal"""
        return self.num / self.den

# Usage
fr1 = Fraction(3, 4)
fr2 = Fraction(1, 2)

print(fr1 + fr2)  # 10/8
print(fr1 - fr2)  # 2/8
print(fr1 * fr2)  # 3/8
print(fr1 / fr2)  # 6/4
print(fr1.convert_to_decimal())  # 0.75
```

6.5 The Destructor: __del__()

The destructor is called when an object is about to be destroyed.

```
class Example:
    def __init__(self, name):
        self.name = name
        print(f'Constructor called for {self.name}')

    def __del__(self):
        print(f'Destructor called for {self.name}')

obj1 = Example("Object1")  # Constructor called
obj2 = obj1  # Reference copy (no new object)

del obj1  # Doesn't call destructor (obj2 still references it)
del obj2  # Now destructor is called
```

Important Note

The destructor is called when:

- The last reference to an object is deleted
- The program terminates
- The object goes out of scope

Python uses garbage collection, so you rarely need to define __del__().

7 Inheritance

7.1 What is Inheritance?

Inheritance allows a class (child/derived class) to inherit attributes and methods from another class (parent/base class). This promotes code reusability and establishes a relationship between classes.

7.1.1 Benefits of Inheritance

- **Code Reusability**: Don't repeat code from parent class
- **Extensibility**: Add new features while keeping existing ones
- **Maintainability**: Changes in parent affect all children
- **Hierarchy**: Models real-world relationships

7.1.2 Limitations of Inheritance

- Increases execution time (jumping between classes)
- Creates tight coupling between parent and child
- Modifications require changes in both parent and child
- Needs careful implementation to avoid errors

7.2 Basic Inheritance Syntax

```
# Parent class
class Parent:
    def __init__(self):
        self.attribute = "value"

    def method(self):
        print("Parent method")

# Child class inherits from Parent
class Child(Parent):
    pass  # Inherits everything from Parent

# Usage
c = Child()
c.method()  # Inherited from Parent
```

7.3 What Gets Inherited?

1. **Constructor**: If child doesn't define __init__(), parent's constructor is used
2. **Public Methods**: All public methods are inherited
3. **Public Attributes**: All public attributes are inherited
4. **Protected Members**: Single underscore members (_attribute) are inherited

Warning

Private members (__attribute) are NOT directly accessible in child classes!
They undergo name mangling and become _ParentClassName__attribute.

Inheritance Example

```
class Phone:
    def __init__(self, price, brand, camera):
        print("Inside Phone constructor")
        self.__price = price  # Private
        self.brand = brand    # Public
        self.camera = camera  # Public

    def buy(self):
        print("Buying a phone")

    def get_price(self):
        return self.__price

class SmartPhone(Phone):
    pass

# Create SmartPhone instance
s = SmartPhone(20000, "Apple", 13)
# Output: Inside Phone constructor

s.buy()  # Inherited method works
print(s.brand)  # Inherited public attribute
# print(s.__price)  # AttributeError! Private attribute
print(s.get_price())  # Works! Accessing through method
```

7.4 Constructor Inheritance

7.4.1 Case 1: Child Has No Constructor

If child class doesn't define __init__(), parent's constructor is automatically used.

```
class Phone:
    def __init__(self, price, brand):
        self.price = price
        self.brand = brand

class SmartPhone(Phone):
    pass  # No constructor

s = SmartPhone(20000, "Apple")
print(s.price)  # 20000 (from parent constructor)
```

7.4.2 Case 2: Child Has Its Own Constructor

If child defines __init__(), parent's constructor is NOT automatically called.

```
class Phone:
    def __init__(self, price, brand):
        self.price = price
```

```
        self.brand = brand

class SmartPhone(Phone):
    def __init__(self, os, ram):
        self.os = os
        self.ram = ram

s = SmartPhone("Android", 4)
print(s.os)      # Android
# print(s.price)  # AttributeError! Parent constructor not called
```

7.5 The super() Function

`super()` allows calling parent class methods, especially the constructor.

Using super() with Constructor

```
class Phone:
    def __init__(self, price, brand, camera):
        print("Inside Phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera

class SmartPhone(Phone):
    def __init__(self, price, brand, camera, os, ram):
        print("Inside SmartPhone constructor")
        # Call parent constructor
        super().__init__(price, brand, camera)
        # Add child-specific attributes
        self.os = os
        self.ram = ram

s = SmartPhone(20000, "Samsung", 12, "Android", 4)
# Output:
# Inside SmartPhone constructor
# Inside Phone constructor

print(s.brand)  # Samsung
print(s.os)     # Android
```

7.6 Method Overriding

Child classes can override parent methods to provide specific implementations.

Method Overriding

```
class Phone:
    def __init__(self, price, brand):
        self.price = price
        self.brand = brand

    def buy(self):
        print("Buying a phone")

```

```python
class SmartPhone(Phone):
    # Override the buy method
    def buy(self):
        print("Buying a smartphone")
        # Optionally call parent method
        super().buy()

s = SmartPhone(20000, "Apple")
s.buy()
# Output:
# Buying a smartphone
# Buying a phone
```

Important Note

Key Points about super():

- Can only be used inside a class method
- Commonly used in __init__() to call parent constructor
- Can call any parent method, not just __init__()
- Returns a proxy object that delegates calls to parent class

8 Types of Inheritance

8.1 1. Single Inheritance

One child class inherits from one parent class (simple linear inheritance).

Structure: Parent → Child

Single Inheritance Example

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

    def eat(self):
        print(f"{self.name} is eating")

class Dog(Animal):
    def bark(self):
        print(f"{self.name} is barking")

# Dog inherits from Animal
dog = Dog("Buddy")
dog.eat()   # Inherited from Animal
dog.bark()  # Dog's own method
```

8.2 2. Multilevel Inheritance

Classes inherit in a chain: Grandparent → Parent → Child

Multilevel Inheritance Example

```python
class Product:
    def review(self):
        print("Product customer review")

class Phone(Product):
    def __init__(self, price, brand):
        self.price = price
        self.brand = brand

    def buy(self):
        print("Buying a phone")

class SmartPhone(Phone):
    def apps(self):
        print("Installing apps")

# SmartPhone inherits from Phone, which inherits from Product
s = SmartPhone(20000, "Apple")
s.review()  # From Product (2 levels up)
s.buy()     # From Phone (1 level up)
s.apps()    # From SmartPhone (own method)
```

8.3 3. Hierarchical Inheritance

Multiple child classes inherit from one parent class.

Structure:

```
      Parent
     /      \
Child1      Child2
```

Hierarchical Inheritance Example

```python
class Phone:
    def __init__(self, price, brand):
        self.price = price
        self.brand = brand

    def buy(self):
        print("Buying a phone")

# Multiple children from same parent
class SmartPhone(Phone):
    def apps(self):
        print("Smartphone apps")

class FeaturePhone(Phone):
    def basic_features(self):
        print("Basic phone features")

# Both inherit from Phone independently
s = SmartPhone(20000, "Apple")
f = FeaturePhone(1000, "Nokia")

s.buy()  # Inherited
f.buy()  # Inherited
s.apps()  # SmartPhone specific
f.basic_features()  # FeaturePhone specific
```

8.4 4. Multiple Inheritance

A child class inherits from multiple parent classes.

Structure: Child(Parent1, Parent2)

Multiple Inheritance Example

```python
class Phone:
    def __init__(self, price, brand):
        self.price = price
        self.brand = brand

    def buy(self):
        print("Buying a phone")

class Product:
    def review(self):
        print("Customer review")

```

```python
# SmartPhone inherits from both Phone and Product
class SmartPhone(Phone, Product):
    pass

s = SmartPhone(20000, "Apple")
s.buy()      # From Phone
s.review()   # From Product
```

8.5 5. Hybrid Inheritance

Combination of multiple inheritance types.

Hybrid Inheritance Example

```python
class Animal:
    def __init__(self, name):
        self.name = name
        print(f"Animal.__init__ called for {name}")

    def sound(self):
        print(f"{self.name} makes a sound")

class Mammal(Animal):
    def __init__(self, name):
        super().__init__(name)
        print(f"Mammal.__init__ called for {name}")

    def feed_milk(self):
        print(f"{self.name} feeds milk")

class Bird(Animal):
    def __init__(self, name):
        super().__init__(name)
        print(f"Bird.__init__ called for {name}")

    def fly(self):
        print(f"{self.name} is flying")

# Bat inherits from both Mammal and Bird
class Bat(Mammal, Bird):
    def __init__(self, name):
        super().__init__(name)
        print(f"Bat.__init__ called for {name}")

    def nocturnal(self):
        print(f"{self.name} is nocturnal")

bat = Bat("Bruce")
# Initialization follows MRO: Bat -> Mammal -> Bird -> Animal

bat.sound()       # From Animal
bat.feed_milk()   # From Mammal
bat.fly()         # From Bird
bat.nocturnal()   # From Bat
```

9 Method Resolution Order (MRO)

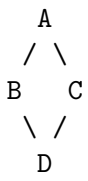
9.1 What is MRO?

Method Resolution Order (MRO) is the order in which Python searches for methods and attributes in a class hierarchy, especially in multiple inheritance.

9.2 The Diamond Problem

The diamond problem occurs when a class inherits from two classes that share a common ancestor.

Structure:



Problem: If D inherits from B and C, and both B and C inherit from A, which path should Python follow?

9.3 How Python Resolves the Diamond Problem

Python uses **C3 Linearization Algorithm** to determine MRO:

1. **Depth-first**: Go deep before going wide
2. **Left-to-right**: Check parents from left to right
3. **No duplicates**: Each class appears only once
4. **Monotonicity**: Subclass always comes before superclass

For class D(B, C), the MRO is: **D** → **B** → **C** → **A** → **object**

9.4 Three Cases of super() Chain Behavior

9.4.1 Case 1: C Does NOT Call super() - Chain Breaks at C

Case 1: Broken Chain at C

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

    def greet(self):
        print(f"Hello from A, {self.name}.")

class B(A):
    def greet(self):
        print(f"Hello from B, {self.name}.")
        super().greet()  # Continues to next in MRO (C)

class C(A):
    def greet(self):
        print(f"Hello from C, {self.name}.")
```

```
        # NO super() call - CHAIN BREAKS HERE!

class D(B, C):
    def greet(self):
        print(f"Hello from D, {self.name}.")
        super().greet()  # Starts MRO chain (calls B)

d = D("Frank")
print("MRO:", [cls.__name__ for cls in D.__mro__])
# MRO: ['D', 'B', 'C', 'A', 'object']

d.greet()
# Output:
# Hello from D, Frank.
# Hello from B, Frank.
# Hello from C, Frank.
# (A is NEVER called - chain broken at C!)

# Execution Flow:
# D -> B -> C -> X (stops, A never executes)
```

Warning

In Case 1, even though A is next in the MRO after C, it never executes because C doesn't call `super()`. The chain stops at C.

9.4.2 Case 2: Both B and C Call super() - Complete Chain (CORRECT)

Case 2: Complete Chain - Best Practice

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

    def greet(self):
        print(f"Hello from A, {self.name}.")

class B(A):
    def greet(self):
        print(f"Hello from B, {self.name}.")
        super().greet()  # Continues to next in MRO (C)

class C(A):
    def greet(self):
        print(f"Hello from C, {self.name}.")
        super().greet()  # Continues to next in MRO (A)

class D(B, C):
    def greet(self):
        print(f"Hello from D, {self.name}.")
        super().greet()  # Starts MRO chain (calls B)

d = D("Frank")
print("MRO:", [cls.__name__ for cls in D.__mro__])
```

```
# MRO: ['D', 'B', 'C', 'A', 'object']

d.greet()
# Output:
# Hello from D, Frank.
# Hello from B, Frank.
# Hello from C, Frank.
# Hello from A, Frank.

# Execution Flow:
# D -> B -> C -> A (complete chain!)
```

Important Note

Case 2 is the CORRECT implementation! All classes in the MRO chain execute because every intermediate class calls `super()`.

9.4.3 Case 3: Neither B nor C Call super() - Chain Breaks at B

Case 3: Broken Chain at B

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

    def greet(self):
        print(f"Hello from A, {self.name}.")

class B(A):
    def greet(self):
        print(f"Hello from B, {self.name}.")
        # NO super() call - CHAIN BREAKS HERE!

class C(A):
    def greet(self):
        print(f"Hello from C, {self.name}.")
        # NO super() call (won't be reached anyway)

class D(B, C):
    def greet(self):
        print(f"Hello from D, {self.name}.")
        super().greet()  # Starts MRO chain (calls B)

d = D("Frank")
print("MRO:", [cls.__name__ for cls in D.__mro__])
# MRO: ['D', 'B', 'C', 'A', 'object']

d.greet()
# Output:
# Hello from D, Frank.
# Hello from B, Frank.
# (Both C and A are NEVER called - chain broken at B!)

# Execution Flow:
```

```
# D -> B -> X (stops, C and A never execute)
```

Warning

In Case 3, the chain breaks immediately at B. Since B doesn't call `super()`, neither C nor A ever execute, even though they're both in the MRO.

9.5 Side-by-Side Comparison

Aspect	Case 1	Case 2	Case 3
D calls super()?	YES	YES	YES
B calls super()?	YES	YES	NO
C calls super()?	NO	YES	NO
A executes?	NO	YES	NO
Output	D, B, C	D, B, C, A	D, B
Chain Status	Broken at C	Complete	Broken at B

Table 1: Comparison of super() chain behaviors

9.6 Visual Representation

```
MRO for all cases: D -> B -> C -> A

Case 1 (C breaks chain):
    D [OK] -> B [OK] -> C [OK] -> A [NO]
                            ^
                    Chain breaks here
                    (C doesn't call super)

Case 2 (Complete chain - CORRECT):
    D [OK] -> B [OK] -> C [OK] -> A [OK]
                                    ^
                                All execute!
                            (Everyone calls super)

Case 3 (B breaks chain):
    D [OK] -> B [OK] -> C [NO] -> A [NO]
                ^
            Chain breaks here
            (B doesn't call super)
```

9.7 Key Takeaways

Important Note

Critical Points About Cooperative super():

- **Cooperative Inheritance**: For the full MRO chain to execute, EVERY intermediate class must call `super()`.

- **Breaking the Chain**: If ANY class doesn't call `super()`, the chain stops there, and all subsequent classes in the MRO won't execute.
- **Correct Implementation**: Case 2 is the only correct implementation. In multiple inheritance, all intermediate classes should call `super()`.
- **Diamond Problem Solution**: `super()` ensures the base class (A) is called only ONCE, even though it's reachable through multiple paths.
- **Exception**: Only the final base class should NOT call `super()`, because there's nothing after it in the MRO.

9.8 Best Practice Pattern

```python
# GOOD - Always call super()
def method(self):
    # Do your work first
    print("Doing work in this class")
    # Then call super() to continue the chain
    super().method()

# BAD - Missing super() call
def method(self):
    # Do your work
    print("Doing work in this class")
    # No super() call - breaks the chain!
```

9.9 Practical Example: Why This Matters

Real-World Mixin Example

```python
class DatabaseMixin:
    def save(self):
        print("Saving to database...")
        super().save()  # Must call super()!

class CacheMixin:
    def save(self):
        print("Updating cache...")
        super().save()  # Must call super()!

class Model:
    def save(self):
        print("Validating data...")

class User(DatabaseMixin, CacheMixin, Model):
    def save(self):
        print("Saving user...")
        super().save()

# MRO: User -> DatabaseMixin -> CacheMixin -> Model

user = User()
user.save()
```

```
# Output:
# Saving user...
# Saving to database...
# Updating cache...
# Validating data...

# If any mixin forgets super().save(), the chain breaks!
# You might save to database but forget to update cache,
# causing hard-to-track bugs.
```

9.10 Viewing MRO

```
# Three ways to view MRO:

# 1. Using __mro__ attribute (tuple)
print(D.__mro__)

# 2. Using mro() method (list)
print(D.mro())

# 3. Human-readable format
print([cls.__name__ for cls in D.__mro__])
# Output: ['D', 'B', 'C', 'A', 'object']
```

Warning

Common Mistake: Forgetting to call `super()` in mixin classes or intermediate classes in a diamond inheritance hierarchy. This breaks the cooperative nature of multiple inheritance and can lead to subtle bugs where parts of your code silently fail to execute.

10 Polymorphism

10.1 What is Polymorphism?

Polymorphism means "many forms." It's the ability of different objects to respond to the same method call in different ways.

10.1.1 Types of Polymorphism in Python

1. **Method Overriding**: Child class provides different implementation
2. **Method Overloading**: Same method name, different parameters (limited in Python)
3. **Operator Overloading**: Custom behavior for operators

10.2 Method Overriding

Method overriding allows a child class to provide a specific implementation of a method already defined in parent class.

Method Overriding Example

```
class Animal:
    def sound(self):
        print("Animal makes a sound")

class Dog(Animal):
    def sound(self):
        print("Dog barks")

class Cat(Animal):
    def sound(self):
        print("Cat meows")

# Polymorphic behavior
animals = [Animal(), Dog(), Cat()]

for animal in animals:
    animal.sound()

# Output:
# Animal makes a sound
# Dog barks
# Cat meows
```

10.3 Method Overloading (Simulated)

Python doesn't support traditional method overloading. The last defined method overrides previous ones. However, we can simulate it using default parameters.

Simulating Method Overloading

```
class Shape:
    def area(self, length, width=None):
        """
```

```
        Calculate area
        - If only length: treat as square (length * length)
        - If length and width: treat as rectangle
        """
        if width is None:
            # Square: area = length^2
            return length * length
        else:
            # Rectangle: area = length * width
            return length * width

s = Shape()
print(s.area(5))        # Square: 25
print(s.area(5, 10))    # Rectangle: 50
```

10.4 Operator Overloading

Already covered in Magic Methods section. Here's a summary:

Operator	Magic Method	Expression
+	__add__	obj1 + obj2
-	__sub__	obj1 - obj2
*	__mul__	obj1 * obj2
/	__truediv__	obj1 / obj2
==	__eq__	obj1 == obj2
<	__lt__	obj1 < obj2
>	__gt__	obj1 > obj2

11 Abstraction

11.1 What is Abstraction?

Abstraction means hiding complex implementation details and showing only essential features. It's achieved through abstract classes and interfaces.

11.2 Abstract Base Classes (ABC)

Python's `abc` module provides infrastructure for defining abstract base classes.

Abstract Class Example

```python
from abc import ABC, abstractmethod

class BankApp(ABC):
    """Abstract base class"""

    def database(self):
        """Concrete method - implemented"""
        print("Connected to database")

    @abstractmethod
    def security(self):
        """Abstract method - must be implemented by subclass"""
        pass

    @abstractmethod
    def display(self):
        """Abstract method - must be implemented by subclass"""
        pass

# This will cause error - cannot instantiate abstract class
# obj = BankApp()  # TypeError!

class MobileApp(BankApp):
    """Concrete class implementing abstract methods"""

    def security(self):
        print("Mobile security layer")

    def display(self):
        print("Mobile display interface")

    def mobile_login(self):
        print("Mobile login")

# Now we can create instance
app = MobileApp()
app.database()       # Inherited concrete method
app.security()       # Implemented abstract method
app.display()        # Implemented abstract method
app.mobile_login()   # Own method
```

11.3 Key Points About Abstraction

- **Abstract class**: Class with one or more abstract methods
- **Cannot instantiate**: Cannot create objects of abstract class
- **Must implement**: Subclasses must implement all abstract methods
- **Can have concrete methods**: Abstract classes can have regular methods too
- **Purpose**: Define contract that subclasses must follow

Warning

If a subclass doesn't implement all abstract methods, it also becomes abstract and cannot be instantiated!

12 Aggregation (Has-A Relationship)

12.1 What is Aggregation?

Aggregation is a "has-a" relationship where one class contains references to objects of another class. Unlike inheritance ("is-a"), aggregation represents composition.

Aggregation Example

```
class Address:
    def __init__(self, city, pin, state):
        self.__city = city
        self.pin = pin
        self.state = state

    def get_city(self):
        return self.__city

    def edit_address(self, new_city, new_pin, new_state):
        self.__city = new_city
        self.pin = new_pin
        self.state = new_state

class Customer:
    def __init__(self, name, gender, address):
        self.name = name
        self.gender = gender
        self.address = address  # Has-A relationship

    def print_address(self):
        print(f"{self.address.get_city()}, "
              f"{self.address.pin}, "
              f"{self.address.state}")

    def edit_profile(self, new_name, new_city, new_pin,
    new_state):
        self.name = new_name
        self.address.edit_address(new_city, new_pin, new_state)

# Create Address object
add1 = Address('Gurgaon', 122011, 'Haryana')

# Create Customer with Address
cust = Customer('Nitish', 'male', add1)
cust.print_address()  # Gurgaon, 122011, Haryana

# Edit through Customer
cust.edit_profile('Ankit', 'Mumbai', 111111, 'Maharashtra')
cust.print_address()  # Mumbai, 111111, Maharashtra
```

12.2 Aggregation vs Inheritance

Aggregation (Has-A)	**Inheritance (Is-A)**
Composition relationship	Parent-child relationship
Contains objects of another class	Extends another class
Example: Car has Engine	Example: Car is Vehicle
More flexible (can change at runtime)	More rigid (fixed at compile time)
Looser coupling	Tighter coupling
Preferred for "has-a" relationships	Preferred for "is-a" relationships

13 Reference Variables and Object Behavior

13.1 Reference Variables

In Python, variables are references to objects, not the objects themselves.

Reference Variables Example

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

# Create object
p1 = Person("John")
print(id(p1))  # 140655538334992

# Create another reference to same object
p2 = p1
print(id(p2))  # 140655538334992 (same ID!)

# Modifying through p2 affects p1
p2.name = "Alice"
print(p1.name)  # Alice (both point to same object)
print(p2.name)  # Alice
```

13.2 Pass by Reference

Python passes references to objects, not copies.

Pass by Reference Example

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

def modify_person(person):
    """Function receives reference to object"""
    print(f"Inside function, ID: {id(person)}")
    person.name = "Modified"
    return person

p = Person("Original")
print(f"Before: {p.name}")  # Original
print(f"Outside function, ID: {id(p)}")

modified_p = modify_person(p)

print(f"After: {p.name}")  # Modified
print(f"Returned object, ID: {id(modified_p)}")

# All IDs are the same!
```

13.3 Object Mutability

Objects in Python are mutable - their state can be changed after creation.

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

p = Person("John")
print(id(p))  # 12345

# Modify object (same ID)
p.name = "Alice"
print(id(p))  # 12345 (same object, modified state)

# Reassign variable (different ID)
p = Person("Bob")
print(id(p))  # 67890 (new object!)
```

13.4 Collections of Objects

Objects can be stored in collections (lists, dictionaries, sets).

List of Objects

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

# Create multiple objects
p1 = Person('Nitish', 'male')
p2 = Person('Ankit', 'male')
p3 = Person('Ankita', 'female')

# Store in list
people = [p1, p2, p3]

# Iterate through objects
for person in people:
    print(f"{person.name}: {person.gender}")
```

Dictionary of Objects

```python
# Store objects in dictionary
people_dict = {
    'person1': p1,
    'person2': p2,
    'person3': p3
}

# Access through dictionary
for key in people_dict:
    person = people_dict[key]
    print(f"{key}: {person.name}")
```

14 Advanced OOP Concepts

14.1 Decorators in OOP

Decorators are functions that modify the behavior of other functions or methods.

Method Decorator Example

```python
def swipe_decorator(func):
    """Decorator to swap arguments if needed"""
    def wrapper(first, second):
        if first < second:
            first, second = second, first
        return func(first, second)
    return wrapper

@swipe_decorator
def divide(first, second):
    print(f"The result is: {first/second}")

divide(4, 16)   # Swaps to: 16/4 = 4.0
divide(16, 4)   # No swap needed: 16/4 = 4.0
```

14.2 isinstance() and issubclass()

14.2.1 isinstance()

Checks if an object is an instance of a class.

```python
class Animal:
    pass

class Dog(Animal):
    pass

dog = Dog()
print(isinstance(dog, Dog))      # True
print(isinstance(dog, Animal))   # True (inheritance)
print(isinstance(dog, str))      # False
```

14.2.2 issubclass()

Checks if a class is a subclass of another.

```python
class Animal:
    pass

class Dog(Animal):
    pass

print(issubclass(Dog, Animal))    # True
print(issubclass(Animal, Dog))    # False
print(issubclass(Dog, Dog))       # True
```

14.3 Single vs Double Underscores

Pattern	Meaning
_var	Protected (convention only) - internal use
var_	Avoid naming conflict with keywords
__var	Private (name mangling) - becomes _ClassName__var
__var__	Magic/dunder methods - special Python methods
_	Temporary variable or last result in REPL

Underscore Examples

```
class Example:
    def __init__(self):
        self.public = "Public"
        self._protected = "Protected"
        self.__private = "Private"

e = Example()
print(e.public)        # Works
print(e._protected)    # Works (but shouldn't use)
# print(e.__private)   # AttributeError!
print(e._Example__private)  # Works (name mangling)

# Using for keyword conflict
def make_object(name, class_):  # class_ avoids conflict
    pass
```

14.4 Hashing and Sets

Objects can be stored in sets if they define __hash__() and __eq__().

Custom Object in Set

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

    def __hash__(self):
        """Make object hashable"""
        return hash((self.name, self.age))

    def __eq__(self, other):
        """Define equality"""
        return self.name == other.name and self.age == other.
    age

# Can now use in sets
p1 = Person("John", 30)
p2 = Person("Alice", 25)
p3 = Person("John", 30)  # Same as p1

people_set = {p1, p2, p3}
print(len(people_set))  # 2 (p1 and p3 are considered equal)
```

Important Note

By default, custom objects are hashable (using id()), but if you override __eq__(), you must also override __hash__() to maintain the hash/equality contract.

14.5 enumerate() Function

The `enumerate()` function adds a counter to an iterable.

```
names = ['Alice', 'Bob', 'Charlie']

# Without enumerate
for i in range(len(names)):
    print(f"{i}: {names[i]}")

# With enumerate (better)
for index, name in enumerate(names):
    print(f"{index}: {name}")

# Start counting from 1
for index, name in enumerate(names, start=1):
    print(f"{index}: {name}")
```

15 Practical OOP Examples

15.1 ATM System

Complete ATM System

```
class Atm:
    """ATM system with encapsulation"""
    __counter = 1

    def __init__(self):
        self.pin = ''
        self.__balance = 0
        self.cid = Atm.__counter
        Atm.__counter += 1

    def get_balance(self):
        """Getter for private balance"""
        return self.__balance

    def set_balance(self, new_value):
        """Setter with validation"""
        if isinstance(new_value, int) and new_value >= 0:
            self.__balance = new_value
        else:
            print('Invalid balance amount')

    def create_pin(self, user_pin, initial_balance):
        """Create PIN and set initial balance"""
        self.pin = user_pin
        self.__balance = initial_balance
        print('PIN created successfully')

    def change_pin(self, old_pin, new_pin):
        """Change PIN with validation"""
        if old_pin == self.pin:
            self.pin = new_pin
            print('PIN changed successfully')
        else:
            print('Incorrect old PIN')

    def check_balance(self, user_pin):
        """Check balance with PIN verification"""
        if user_pin == self.pin:
            print(f'Your balance is: ${self.__balance}')
        else:
            print('Incorrect PIN')

    def withdraw(self, user_pin, amount):
        """Withdraw with PIN and balance verification"""
        if user_pin != self.pin:
            print('Incorrect PIN')
            return

        if amount <= 0:
            print('Invalid amount')
            return
```

```
52
53          if amount <= self.__balance:
54              self.__balance -= amount
55              print(f'Withdrawal successful. Balance: ${self.
    __balance}')
56          else:
57              print('Insufficient funds')
58
59      @staticmethod
60      def get_counter():
61          return Atm.__counter
62
63 # Usage
64 atm1 = Atm()
65 atm1.create_pin('1234', 1000)
66 atm1.check_balance('1234')  # Your balance is: $1000
67 atm1.withdraw('1234', 200)  # Withdrawal successful
68 atm1.check_balance('1234')  # Your balance is: $800
```

15.2 2D Point and Line System

Geometric Classes

```
1  class Point:
2      """Represents a 2D point"""
3      def __init__(self, x, y):
4          self.x_cod = x
5          self.y_cod = y
6
7      def __str__(self):
8          return f'<{self.x_cod}, {self.y_cod}>'
9
10     def euclidean_distance(self, other):
11         """Calculate distance between two points"""
12         dx = self.x_cod - other.x_cod
13         dy = self.y_cod - other.y_cod
14         return (dx**2 + dy**2)**0.5
15
16     def distance_from_origin(self):
17         """Calculate distance from origin"""
18         return (self.x_cod**2 + self.y_cod**2)**0.5
19
20 class Line:
21     """Represents a line: Ax + By + C = 0"""
22     def __init__(self, A, B, C):
23         self.A = A
24         self.B = B
25         self.C = C
26
27     def __str__(self):
28         return f'{self.A}x + {self.B}y + {self.C} = 0'
29
30     def point_on_line(self, point):
31         """Check if point lies on line"""
32         result = self.A * point.x_cod + self.B * point.y_cod +
```

```
    self.C
        return result == 0

    def shortest_distance(self, point):
        """Calculate shortest distance from point to line"""
        numerator = abs(self.A * point.x_cod +
                        self.B * point.y_cod + self.C)
        denominator = (self.A**2 + self.B**2)**0.5
        return numerator / denominator

# Usage
p1 = Point(1, 2)
p2 = Point(4, 6)
print(f"Point 1: {p1}")
print(f"Point 2: {p2}")
print(f"Distance: {p1.euclidean_distance(p2):.2f}")

line = Line(1, 1, -3)  # x + y - 3 = 0
print(f"Line: {line}")
print(f"Point on line: {line.point_on_line(p1)}")
print(f"Distance to line: {line.shortest_distance(p1):.2f}")
```

16 OOP Best Practices and Design Principles

16.1 SOLID Principles

16.1.1 S - Single Responsibility Principle

A class should have only one reason to change (one responsibility).

Single Responsibility

```python
# Bad: Multiple responsibilities
class User:
    def __init__(self, name):
        self.name = name

    def save_to_database(self):
        # Database logic
        pass

    def send_email(self):
        # Email logic
        pass

# Good: Separate responsibilities
class User:
    def __init__(self, name):
        self.name = name

class UserRepository:
    def save(self, user):
        # Database logic
        pass

class EmailService:
    def send(self, user, message):
        # Email logic
        pass
```

16.1.2 O - Open/Closed Principle

Classes should be open for extension but closed for modification.

Open/Closed Principle

```python
# Good: Extend through inheritance
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
```

```
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2
```

16.1.3 L - Liskov Substitution Principle

Subclasses should be substitutable for their base classes.

16.1.4 I - Interface Segregation Principle

Clients shouldn't be forced to depend on interfaces they don't use.

16.1.5 D - Dependency Inversion Principle

Depend on abstractions, not concretions.

16.2 General Best Practices

1. **Use descriptive names**: Class and method names should clearly indicate purpose
2. **Keep classes small**: Each class should do one thing well
3. **Encapsulate data**: Use private attributes with getters/setters
4. **Favor composition over inheritance**: Use "has-a" when appropriate
5. **Use abstract classes**: Define contracts for subclasses
6. **Document your code**: Use docstrings for classes and methods
7. **Follow PEP 8**: Python's style guide for consistency
8. **Write testable code**: Design for unit testing

16.3 Common Mistakes to Avoid

Warning

- **Deep inheritance hierarchies**: Keep inheritance tree shallow
- **God classes**: Avoid classes that do everything
- **Tight coupling**: Reduce dependencies between classes
- **Forgetting super()**: Always use super() in multiple inheritance
- **Mutable default arguments**: Don't use mutable objects as defaults
- **Not using __repr__**: Always implement for debugging
- **Breaking encapsulation**: Don't access private members from outside

16.4 Composition vs Inheritance

Use Inheritance When:	**Use Composition When:**
True "is-a" relationship exists	"has-a" relationship exists
Need to reuse implementation	Need to reuse functionality
Subclass extends parent behavior	Want flexibility to change behavior
Relationship is stable	Relationship may change
Example: Dog is an Animal	Example: Car has an Engine

17 Common Interview Questions

17.1 Conceptual Questions

17.1.1 Q1: What are the four pillars of OOP?

Answer:

1. **Encapsulation**: Bundling data and methods, hiding internal details
2. **Inheritance**: Creating new classes from existing ones
3. **Polymorphism**: Same interface, different implementations
4. **Abstraction**: Hiding complexity, showing only essentials

17.1.2 Q2: Explain the difference between `__str__` and `__repr__`

Answer:

- `__str__()`: User-friendly string representation, for end users
- `__repr__()`: Developer-friendly representation, for debugging
- `str()` is meant to be readable, `repr()` to be unambiguous
- If only one is defined, Python uses it for both

17.1.3 Q3: What is the diamond problem and how does Python solve it?

Answer:

- Diamond problem occurs in multiple inheritance when a class inherits from two classes that share a common ancestor
- Python solves it using MRO (Method Resolution Order) with C3 Linearization
- MRO ensures each class is called only once, in a consistent order
- Use `super()` to maintain proper MRO chain
- Check MRO with `ClassName.__mro__`

17.1.4 Q4: What's the difference between class method and static method?

Answer:

Feature	Class Method	Static Method
Decorator	@classmethod	@staticmethod
First parameter	cls (class)	None
Access class data	Yes	No
Modify class state	Yes	No
Use case	Factory methods	Utility functions

17.2 Coding Questions

17.2.1 Q5: Implement a class with private attributes

```python
class BankAccount:
    def __init__(self, account_number, balance=0):
        self.__account_number = account_number
        self.__balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return True
        return False

    def get_balance(self):
        return self.__balance
```

17.2.2 Q6: Create a class hierarchy with method overriding

```python
class Animal:
    def sound(self):
        print("Some sound")

class Dog(Animal):
    def sound(self):
        print("Bark")

class Cat(Animal):
    def sound(self):
        print("Meow")

# Polymorphism in action
animals = [Dog(), Cat(), Animal()]
for animal in animals:
    animal.sound()
```

18 Quick Reference Guide

18.1 Class Definition Syntax

Basic Class Template

```python
class ClassName:
    """Class docstring"""

    # Class attribute
    class_var = "shared"

    def __init__(self, param1, param2):
        """Constructor"""
        self.param1 = param1
        self.param2 = param2

    def instance_method(self):
        """Instance method"""
        pass

    @classmethod
    def class_method(cls):
        """Class method"""
        pass

    @staticmethod
    def static_method():
        """Static method"""
        pass

    def __str__(self):
        """String representation"""
        return f"ClassName({self.param1}, {self.param2})"
```

18.2 Inheritance Templates

Single Inheritance

```python
class Parent:
    def __init__(self, value):
        self.value = value

class Child(Parent):
    def __init__(self, value, extra):
        super().__init__(value)
        self.extra = extra
```

Multiple Inheritance

```
class Parent1:
    def method1(self):
        super().method1()

class Parent2:
    def method1(self):
        pass

class Child(Parent1, Parent2):
    def method1(self):
        super().method1()
```

18.3 Common Magic Methods

```
class MyClass:
    def __init__(self):             # Constructor
        pass

    def __str__(self):              # str(obj)
        return "string"

    def __repr__(self):             # repr(obj)
        return "representation"

    def __len__(self):              # len(obj)
        return 0

    def __add__(self, other):       # obj1 + obj2
        pass

    def __eq__(self, other):        # obj1 == obj2
        pass

    def __lt__(self, other):        # obj1 < obj2
        pass

    def __getitem__(self, key):     # obj[key]
        pass

    def __call__(self):             # obj()
        pass

    def __del__(self):              # Destructor
        pass
```

18.4 Access Modifiers Cheat Sheet

Syntax	Type	Accessibility
self.var	Public	Everywhere
self._var	Protected	Convention only
self.__var	Private	Name mangled

19 Conclusion

Object-Oriented Programming is a powerful paradigm that enables:

- **Modular code**: Organized into reusable classes
- **Data protection**: Through encapsulation
- **Code reuse**: Through inheritance
- **Flexibility**: Through polymorphism
- **Maintainability**: Through abstraction

19.1 Key Takeaways

1. **Classes and Objects**: Classes are blueprints, objects are instances
2. **Encapsulation**: Hide internal details, expose interfaces
3. **Inheritance**: Reuse code through parent-child relationships
4. **Polymorphism**: Same interface, different implementations
5. **Abstraction**: Hide complexity, show essentials
6. **MRO**: Python's method resolution order solves diamond problem
7. **Magic Methods**: Customize object behavior with dunder methods
8. **super()**: Essential for proper multiple inheritance

19.2 Next Steps

- Practice implementing classes for real-world problems
- Study design patterns (Factory, Singleton, Observer, etc.)
- Learn about metaclasses for advanced metaprogramming
- Explore dataclasses for simplified class creation
- Study type hints for better code documentation
- Practice with OOP coding challenges

19.3 Additional Resources

- Python Official Documentation: `https://docs.python.org/3/tutorial/classes.html`
- Real Python OOP Tutorials: `https://realpython.com/python3-object-oriented-programming/`
- Design Patterns in Python: `https://refactoring.guru/design-patterns/python`
- Clean Code by Robert C. Martin
- Fluent Python by Luciano Ramalho

End of Python OOP Complete Reference Guide