Complete Guide to Object-Oriented Programming in Python

A Comprehensive Textbook for Students

September 21, 2025

Contents

Chapter 1

Introduction to Objects and Classes

1.1 Core Concept: Everything in Python is an Object

In Python, **everything is an object**. This fundamental principle means that every piece of data (numbers, strings, functions, classes) has:

- A **type** (class it belongs to)
- **Attributes** (data associated with it)
- **Methods** (functions that can be called on it)

Key Concept

A **class** is a blueprint for creating objects, while an **object** (or instance) is a specific realization of that class.

1.1.1 Basic Class Definition

```python
class Chai:
    pass

class ChaiTime:
    pass

# Demonstrating object types
print(type(Chai))           # <class ’type’>
ginger_tea = Chai()         # Creating an object instance
print(type(ginger_tea) is Chai)       # True
print(type(ginger_tea) is ChaiTime)   # False
```

1.1.2 Key Concepts

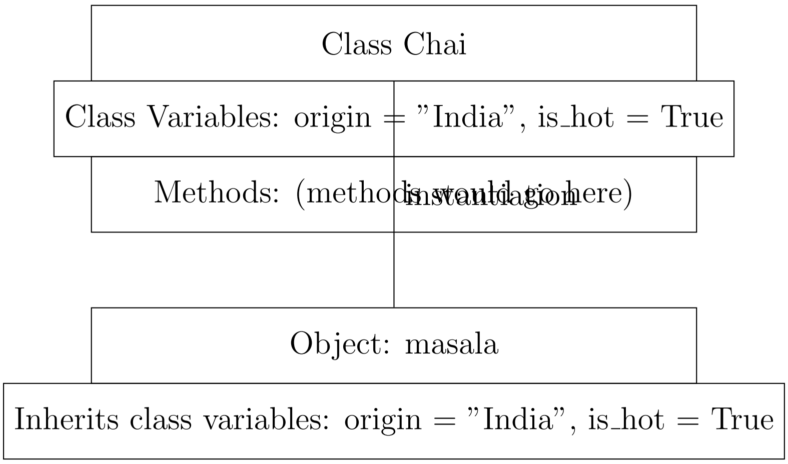
Concept	**Definition**	**Example**
Class	A blueprint for creating objects	`class Chai:`
Object/Instance	A specific realization of a class	`ginger_tea = Chai()`
Type	The class an object belongs to	`type(ginger_tea)`

Table 1.1: Fundamental OOP Concepts

Chapter 2

Classes and Objects Fundamentals

2.1 Class Structure


Class Chai
Class Variables: origin = "India", is_hot = True
Methods: (methods would go here)
instantiation
Object: masala
Inherits class variables: origin = "India", is_hot = True

2.1.1 Example: Basic Class with Attributes

```python
class Chai:
    origin = "India"  # Class variable

print(Chai.origin)    # Access class variable directly

# Adding class variables dynamically
Chai.is_hot = True
print(Chai.is_hot)    # True

# Creating object from class
masala = Chai()
print(f"Masala {masala.origin}")  # Masala India
print(f"Masala {masala.is_hot}")  # Masala True
```

Chapter 3

Namespaces in Object-Oriented Programming

3.1 Understanding Variable Scope

Namespaces define where variables can be accessed. In OOP, we have different levels:

Namespace Level	Description	Access Method
Class Level	Variables shared by all instances	`ClassName.variable`
Instance Level	Variables unique to each object	`object.variable`

Table 3.1: Namespace Levels in OOP

Example

```
class Chai:
    origin = "India"  # Class variable

masala = Chai()

# Instance variable shadows class variable
masala.is_hot = False

print("Class attribute", Chai.is_hot)      # True (class level)
print(f"Masala {masala.is_hot}")           # False (instance level)

# Adding instance-specific attribute
masala.flavor = "Masala"
print(masala.flavor)  # Masala (only exists for this instance)
```

Chapter 4

Attribute Shadowing

4.1 Concept: Instance Variables Override Class Variables

When an instance variable has the same name as a class variable, the instance variable ”shadows” (hides) the class variable for that specific instance.

```python
class Chai:
    temperature = "hot"
    strength = "strong"

cutting = Chai()
print(cutting.temperature)  # "hot" (from class)

# Instance variable shadows class variable
cutting.temperature = "Mild"
print("After changing:", cutting.temperature)    # "Mild" (instance)
print("Inside the Class:", Chai.temperature)      # "hot" (class unchanged)

# Deleting instance variable reveals class variable
del cutting.temperature
print("After deleting:", cutting.temperature)    # "hot" (back to class)
```

Important Note

Understanding attribute shadowing is crucial for debugging and maintaining clean code architecture.

Chapter 5

The self Parameter

5.1 Understanding self: The Object Reference

The `self` parameter is a reference to the current instance of the class. It's automatically passed to instance methods.

```python
class Chaicup:
    size = 150  # ml

    def describe(self):
        return f"A {self.size}ml chai cup"

cup = Chaicup()
print(cup.describe())                # "A 150ml chai cup"
print(Chaicup.describe(cup))         # Same result, explicit self
```

5.1.1 Method Call Comparison

Call Method	**Syntax**	**self Parameter**
Instance Method	`cup.describe()`	Automatically passed
Class Method	`Chaicup.describe(cup)`	Manually passed

Table 5.1: Method Call Approaches

Chapter 6

Object Initialization with __init__

6.1 Constructor Method: __init__

The __init__ method is called automatically when an object is created. It initializes the object's attributes.

```python
class ChaiOrder:
    def __init__(self, type_, size):
        self.type = type_     # Instance variable
        self.size = size      # Instance variable

    def summary(self):
        return f"{self.size}ml of {self.type} chai"

# Creating objects with initialization
order = ChaiOrder("Masala", 200)
print(order.summary())  # "200ml of Masala chai"

order_two = ChaiOrder("Ginger", 220)
print(order_two.summary())  # "220ml of Ginger chai"
```

6.1.1 Object Creation Process

Step	Description	Code
1	Object creation	`order = ChaiOrder("Masala", 200)`
2	__init__ called	`__init__(order, "Masala", 200)`
3	Attributes set	`order.type = "Masala", order.size = 200`

Table 6.1: Object Creation Steps

Chapter 7

Inheritance and Composition

7.1 Single Inheritance

Inheritance allows a class to inherit attributes and methods from another class.

```python
class BaseChai:
    def __init__(self, type_):
        self.type = type_

    def prepare(self):
        print(f"Preparing {self.type} chai....")

class MasalaChai(BaseChai):  # Inherits from BaseChai
    def add_spices(self):
        print("Adding masala spices....")
```

7.2 Composition Pattern

Composition involves using objects of other classes as attributes.

```python
class ChaiShop:
    chai_cls = BaseChai  # Class variable

    def __init__(self):
        self.chai = self.chai_cls("Regular")  # Composition

    def serve(self):
        print(f"Serving {self.chai.type} chai in the shop")
        self.chai.prepare()

class FancyChaiShop(ChaiShop):
    chai_cls = MasalaChai  # Override class variable
```

7.2.1 Inheritance vs Composition

Aspect	**Inheritance**	**Composition**
Relationship	”is-a” relationship	”has-a” relationship
Coupling	Tight coupling	Loose coupling
Flexibility	Less flexible	More flexible
Example	`MasalaChai` is a `BaseChai`	`ChaiShop` has a `Chai`

Table 7.1: Inheritance vs Composition Comparison

Chapter 8

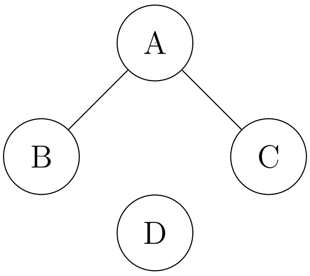
Method Resolution Order (MRO)

8.1 Understanding Method Resolution

When multiple inheritance is used, Python follows a specific order to resolve method calls.

```python
class A:
    label = "A: Base class"

class B(A):
    label = "B: Masala blend"

class C(A):
    label = "C: Herbal blend"

class D(B, C):  # Multiple inheritance
    pass

cup = D()
print(cup.label)    # "B: Masala blend"
print(D.__mro__)    # Method Resolution Order
```

8.1.1 MRO Hierarchy



Resolution Order: D → B → C → A → object

8.1.2 MRO Rules

Rule	**Description**	**Example**
Depth-First	Go deep before wide	D → B → A before D → C → A
Left-to-Right	Left parent first	B (left) before C (right)
Linearization	No class appears before its parents	A appears after B and C

Table 8.1: Method Resolution Order Rules

Chapter 9

Static Methods and Class Methods

9.1 Static Methods

Static methods don't access class or instance data. They're utility functions that belong logically to the class.

```python
class ChaiUtils:
    @staticmethod
    def clean_ingredients(text):
        return [item.strip() for item in text.split(",")]

# Usage - no instance needed
raw = "  water, milk  , ginger , honey"
cleaned = ChaiUtils.clean_ingredients(raw)
print(cleaned)  # ['water', 'milk', 'ginger', 'honey']
```

9.2 Class Methods

Class methods receive the class as the first argument (`cls`) and can create instances or access class variables.

```python
class ChaiOrder:
    def __init__(self, tea_type, sweetness, size):
        self.tea_type = tea_type
        self.sweetness = sweetness
        self.size = size

    @classmethod
    def from_dict(cls, order_data):
        return cls(
            order_data["tea_type"],
            order_data["sweetness"],
            order_data["size"]
        )

    @classmethod
    def from_string(cls, order_string):
        tea_type, sweetness, size = order_string.split("-")
        return cls(tea_type, sweetness, size)

# Usage examples
```

```
order1 = ChaiOrder.from_dict({
    "tea_type": "Masala",
    "sweetness": "Medium",
    "size": "Large"
})

order2 = ChaiOrder.from_string("Ginger-Low-Small")
print(order1.tea_type, order1.sweetness, order1.size)
print(order2.tea_type, order2.sweetness, order2.size)
```

9.2.1 Method Types Comparison

Method Type	**Decorator**	**First Parameter**	**Access to**	**Use Case**
Instance	None	`self`	Instance & Class	Object operations
Class	`@classmethod`	`cls`	Class only	Alternative constructors
Static	`@staticmethod`	None	Neither	Utility functions

Table 9.1: Comparison of Method Types

Chapter 10

Properties and Encapsulation

10.1 Property Decorators

Properties allow method-like access to attributes with validation and transformation logic.

```python
class TeaLeaf:
    def __init__(self, age):
        self._age = age  # Private attribute convention

    @property
    def age(self):
        return self._age + 2  # Transformation logic

    @age.setter
    def age(self, age):
        if 1 <= age <= 5:
            self._age = age
        else:
            raise ValueError("Tea leaf age must be between 1 and 5 years")

# Usage
leaf = TeaLeaf(2)
print(leaf.age)    # 4 (2 + 2 transformation)

leaf.age = 3       # Uses setter with validation
print(leaf.age)    # 5 (3 + 2 transformation)
```

10.1.1 Property Benefits

Benefit	**Description**	**Example**
Validation	Check values before setting	Age must be 1-5 years
Transformation	Modify values on access	Add 2 to stored age
Encapsulation	Hide internal representation	_age is internal
Interface Stability	Change implementation without affecting usage	Attribute → property

Table 10.1: Benefits of Using Properties

Chapter 11

Summary and Best Practices

11.1 The Four Pillars of OOP

Principle	**Description**	**Python Implementation**
Encapsulation	Bundle data and methods together	Classes with private attributes (`_attribute`)
Inheritance	Create new classes based on existing ones	`class Child(Parent):`
Polymorphism	Same interface, different implementations	Method overriding, duck typing
Abstraction	Hide complex implementation details	Abstract classes, properties

Table 11.1: Four Pillars of Object-Oriented Programming

11.2 Best Practices Checklist

✓ Use meaningful class and method names

✓ Follow the single responsibility principle

✓ Use `_attribute` for internal/private attributes

✓ Implement `__init__` for proper object initialization

✓ Use properties for attribute access control

✓ Prefer composition over inheritance when appropriate

✓ Document your classes with docstrings

✓ Use static methods for utility functions

✓ Use class methods for alternative constructors

Chapter 12

Practice Exercises

12.1 Exercise 1: Basic Class Creation

Create a `Book` class with attributes for title, author, and pages. Include methods to display book info and check if it's a long book (¿300 pages).

```python
# Your solution here
class Book:
    def __init__(self, title, author, pages):
        # Implement constructor
        pass

    def display_info(self):
        # Implement display method
        pass

    def is_long_book(self):
        # Implement long book check
        pass
```

12.2 Exercise 2: Inheritance

Create a `Vehicle` base class and derive `Car` and `Motorcycle` classes with specific attributes and methods.

```python
# Your solution here
class Vehicle:
    def __init__(self, make, model, year):
        # Implement base vehicle
        pass

class Car(Vehicle):
    def __init__(self, make, model, year, doors):
        # Implement car class
        pass

class Motorcycle(Vehicle):
    def __init__(self, make, model, year, engine_size):
        # Implement motorcycle class
        pass
```

12.3 Exercise 3: Properties

Implement a `BankAccount` class with balance property that prevents negative balances.

```python
# Your solution here
class BankAccount:
    def __init__(self, initial_balance):
        # Implement constructor
        pass

    @property
    def balance(self):
        # Implement balance getter
        pass

    @balance.setter
    def balance(self, amount):
        # Implement balance setter with validation
        pass
```

12.4 Exercise 4: Class Methods

Create a `Person` class with alternative constructors for creating objects from different data formats.

```python
# Your solution here
class Person:
    def __init__(self, name, age, email):
        # Implement constructor
        pass

    @classmethod
    def from_csv_string(cls, csv_string):
        # Implement CSV string constructor
        pass

    @classmethod
    def from_dict(cls, person_dict):
        # Implement dictionary constructor
        pass
```

Appendix A

Complete Code Examples

A.1 Comprehensive Example: Chai Shop Management System

```python
class Ingredient:
    """Represents an ingredient used in chai preparation."""

    def __init__(self, name, quantity, unit):
        self.name = name
        self._quantity = quantity
        self.unit = unit

    @property
    def quantity(self):
        return self._quantity

    @quantity.setter
    def quantity(self, value):
        if value < 0:
            raise ValueError("Quantity cannot be negative")
        self._quantity = value

    def __str__(self):
        return f"{self.quantity} {self.unit} of {self.name}"

class Chai:
    """Base class for different types of chai."""

    def __init__(self, name, base_price):
        self.name = name
        self.base_price = base_price
        self.ingredients = []

    def add_ingredient(self, ingredient):
        self.ingredients.append(ingredient)

    def prepare(self):
        print(f"Preparing {self.name}...")
        for ingredient in self.ingredients:
            print(f"Adding {ingredient}")
        print(f"{self.name} is ready!")

    def calculate_cost(self):
        return self.base_price

    def __str__(self):
        return f"{self.name} - ${self.base_price:.2f}"

class MasalaChai(Chai):
    """Specialized chai with masala spices."""

    def __init__(self, spice_level="medium"):
        super().__init__("Masala Chai", 3.50)
        self.spice_level = spice_level
        self._add_default_ingredients()

    def _add_default_ingredients(self):
        self.add_ingredient(Ingredient("Tea leaves", 10, "grams"))
        self.add_ingredient(Ingredient("Milk", 200, "ml"))
        self.add_ingredient(Ingredient("Water", 150, "ml"))
        self.add_ingredient(Ingredient("Sugar", 15, "grams"))
        self.add_ingredient(Ingredient("Cardamom", 2, "pods"))
        self.add_ingredient(Ingredient("Ginger", 5, "grams"))

    def calculate_cost(self):
        base_cost = super().calculate_cost()
        spice_multipliers = {"mild": 1.0, "medium": 1.2, "hot": 1.5}
        return base_cost * spice_multipliers.get(self.spice_level, 1.0)

class GreenTea(Chai):
    """Light and healthy green tea option."""

    def __init__(self):
        super().__init__("Green Tea", 2.50)
        self._add_default_ingredients()

    def _add_default_ingredients(self):
        self.add_ingredient(Ingredient("Green tea leaves", 5, "grams"))
        self.add_ingredient(Ingredient("Hot water", 250, "ml"))

class ChaiOrder:
    """Represents a customer's chai order."""

    order_counter = 0

    def __init__(self, customer_name, chai):
        ChaiOrder.order_counter += 1
        self.order_id = ChaiOrder.order_counter
        self.customer_name = customer_name
        self.chai = chai
        self.status = "pending"

    @classmethod
    def get_order_count(cls):
        return cls.order_counter

    @staticmethod
    def validate_customer_name(name):
        return len(name) >= 2 and name.isalpha()

    def process_order(self):
        if self.status == "pending":
            print(f"Processing order #{self.order_id} for {self.customer_name}")
            self.chai.prepare()
            self.status = "completed"
            return True
        return False

    def get_total_cost(self):
        return self.chai.calculate_cost()

    def __str__(self):
        return f"Order #{self.order_id}: {self.customer_name} - {self.chai.name}"

class ChaiShop:
    """Manages the chai shop operations."""

    def __init__(self, shop_name):
        self.shop_name = shop_name
        self.orders = []
        self.menu = {
            "masala_mild": MasalaChai("mild"),
            "masala_medium": MasalaChai("medium"),
            "masala_hot": MasalaChai("hot"),
            "green_tea": GreenTea()
        }

    def display_menu(self):
        print(f"\n--- {self.shop_name} Menu ---")
        for key, chai in self.menu.items():
            print(f"{key}: {chai} - ${chai.calculate_cost():.2f}")

    def take_order(self, customer_name, chai_type):
        if not ChaiOrder.validate_customer_name(customer_name):
            raise ValueError("Invalid customer name")

        if chai_type not in self.menu:
            raise ValueError("Chai type not available")

        chai = self.menu[chai_type]
        order = ChaiOrder(customer_name, chai)
        self.orders.append(order)
        return order

    def process_all_pending_orders(self):
        pending_orders = [order for order in self.orders if order.status == "pending"]
        for order in pending_orders:
            order.process_order()

    def get_daily_revenue(self):
        return sum(order.get_total_cost() for order in self.orders if order.status == "completed")

    def __str__(self):
        return f"{self.shop_name} - {len(self.orders)} orders, ${self.
    get_daily_revenue():.2f} revenue"

# Demo usage
if __name__ == "__main__":
    # Create chai shop
    shop = ChaiShop("Aromatic Chai House")

    # Display menu
    shop.display_menu()

    # Take some orders
    order1 = shop.take_order("Alice", "masala_medium")
    order2 = shop.take_order("Bob", "green_tea")
    order3 = shop.take_order("Charlie", "masala_hot")

    # Process orders
    shop.process_all_pending_orders()

    # Show shop status
    print(f"\n{shop}")
    print(f"Total orders today: {ChaiOrder.get_order_count()}")
```

Appendix B

Glossary

Class A blueprint or template for creating objects that defines attributes and methods

Object/Instance A specific realization of a class with its own set of attribute values

Inheritance The mechanism by which a class can inherit attributes and methods from another class

Encapsulation The bundling of data and methods that operate on that data within a single unit

Polymorphism The ability of different classes to be treated as instances of the same type through inheritance

Abstraction The process of hiding complex implementation details while showing only essential features

Method Resolution Order (MRO) The order in which Python searches for methods in a hierarchy of classes

Static Method A method that belongs to a class but doesn't access class or instance data

Class Method A method that receives the class as the first argument and can be called on the class itself

Property A special kind of attribute that allows method-like access with validation and transformation

Index