PROGRAMMING PRINCIPLES

Object-Oriented Programming in Python

The chapter is grounded in the Week 10 lecture slides, including the OOP introduction, class/object definitions, methods, self, __init__, inheritance, access conventions, encapsulation, polymorphism, operator overloading and special methods. It also incorporates the supplied Section 25 material on modules, members, empty classes, special members, association, inheritance and the Person/Member/Student/Staff examples. The GoCard references are included only to connect the chapter to the Week 10 workshop application of these concepts.

Learning Outcomes

After studying this chapter, you should be able to:

- Explain why object-oriented programming is useful for modelling related data and behaviour.
- Distinguish clearly between a class, an object and an instance.
- Identify attributes and methods in a class.
- Write a class with an __init__ method.
- Explain exactly what self represents.
- Create several independent objects from one class.
- Access and modify object attributes and call methods.
- Distinguish instance members from class members.
- Explain the intended meaning of public, protected and private naming conventions in Python.
- Explain encapsulation and information hiding.
- Use one object as an attribute of another object through association.
- Define a subclass and understand what it inherits from a superclass.
- Use super().__init__() to initialise inherited state.
- Override an inherited method.
- Explain polymorphism using a simple example.
- Recognise operator overloading and common special methods.
- Use __str__ to provide a useful representation of an object.
- Place class definitions in a module and import them into another program.
- Design a class from a programming problem rather than starting from syntax alone.
- Trace object-oriented code by tracking each object's state separately.

How to Study This Chapter

1. Read the explanation before copying the code.
2. Predict what each example will do before looking at the output.
3. Run the examples yourself and change one value at a time.
4. Draw a small object-state table when tracing code.
5. When you see inheritance, first identify the 'is-a' relationship before writing code.
6. At the end, complete the revision questions without looking at the answers.

From Real-World Objects to Software Objects

We start with familiar objects such as students, people, staff, managers, cars, bikes, trucks and buses. Each has properties and actions. For example, a Student can have a name, age and GPA and can perform actions such as studying; a Vehicle can have make, model, colour and mileage and can drive or update its mileage.

OOP uses this same way of thinking when designing software. Instead of treating every variable and function as unrelated, we group information and operations that belong together.

Real-world concept	**Python/OOP concept**	**Example**
Type of thing	Class	Student
One actual thing	Object / instance	student1
Property	Attribute	student1.age
Action	Method	student1.get_age()
Relationship	Association or inheritance	Person has Address; Student is a Person

Ask: What does this object know? What can this object do? What other objects is it related to? These questions lead naturally to attributes, methods and relationships.

Class, Object and Instance

We define a class as a template or blueprint and an object as an instance of a class. A class defines the members of the objects that will be made as instances.

```
class Student:
    pass
```

This defines the type. It does not yet create a Student object.

```
student1 = Student()
student2 = Student()
```

Now two separate instances exist.

Term	**Meaning**	**Example**
Class	Definition/blueprint	Student
Object	One concrete software entity	student1
Instance	Another way to describe an object created from a class	student2 is an instance of Student

Why create a class?

Without a class, a program might need separate variables and functions for every student. A class gives one reusable definition from which many student objects can be created.

Empty Classes and Attributes

A class can be empty. An empty class is not necessarily useful by itself, but Python objects can have attributes added after creation.

```
class Empty:
    """An empty class."""
```

```
    pass

mt = Empty()
mt.x = 1
mt.y = 2

print(mt.x, mt.y)
```

This works because attributes can be attached to an object. However, for most structured programs, it is clearer to define the expected attributes in __init__, so every object starts with a predictable state.

> **Good practice:** Although Python permits attributes to be added dynamically, use __init__ when an attribute is part of the normal design of the class.

Attributes and Object State

Attributes are the data members that represent the state of an object. We describe attributes or fields as storing information, or the state of the object.

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

student1 = Student("David", 18)
student2 = Student("Kim", 19)
```

Object	self.name	self.age
student1	David	18
student2	Kim	19

The important point is that self.name means 'the name attribute belonging to this particular object'.

```
student1.age = 19

print(student1.age)
print(student2.age)
```

Output:

```
19
19
```

In this particular example the two values become equal, but they are still separate attributes belonging to separate objects. Changing student1.age does not modify student2.age.

Methods: What an Object Can Do

A method is a function defined inside a class. Methods describe operations that can be performed using an object. We state that all instance methods have a special first parameter, self.

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

    def get_age(self):
        return self.age
```

```
    def report(self):
        print(self.name, self.age)

student = Student("David", 18)

print(student.get_age())
student.report()
```

A method can read attributes, change attributes, return a value, print information, or perform other operations related to the object.

Understanding self

The first parameter of an instance method should conventionally be named self. It is a reference to the current object. We note compare this idea with this in Java and emphasise that attributes inside a method should be qualified with self.

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

    def add(self, amount):
        self.value += amount

a = Counter(5)
b = Counter(10)

a.add(2)
b.add(3)
```

Object	Before	Operation	After
a	5	a.add(2)	7
b	10	b.add(3)	13

Conceptually, a.add(2) supplies a as self and 2 as amount. Therefore self.value means a.value during that call.

> **Common exam question:** If a method changes self.value, ask which object was used to call the method. That object is self.

__init__: Initialising an Object

The __init__ method is a predefined special method commonly redefined to initialise objects.

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

student = Student("David", 18)
```

When Student("David", 18) is evaluated, Python creates a Student instance and uses __init__ to initialise it with the supplied values.

Code	Meaning
Student	The class being instantiated.

Student("David", 18)	Construct an object with two supplied values.
self	The newly created object during initialisation.
self.name = name	Store the supplied name in the object.
self.age = age	Store the supplied age in the object.

Do not confuse: name is a parameter/local name inside __init__; self.name is an attribute stored in the object.

Accessing Attributes and Calling Methods

```
student1 = Student("David", 18)

print(student1.name)        # attribute
print(student1.get_age())   # method
```

We demonstrate the same distinction: student1.full_name accesses an attribute, while student1.get_age() calls a method.

Expression	What it does
student1.name	Reads an attribute value.
student1.age = 19	Changes an attribute value.
student1.get_age()	Calls a method and receives its return value.
student1.report()	Calls a method that performs an action.

Adding New Attributes to an Existing Instance

We show that a new attribute can be added directly to an existing instance, for example student1.sex and student1.address.

```
student1.sex = "Male"
student1.address = "1 University Drive"

print(student1.sex)
print(student1.address)
```

This demonstrates Python's flexibility, but it also shows why a well-designed class should establish expected state consistently in __init__ when possible.

Student takeaway: Python allows dynamic attributes, but your programs are easier to understand when objects created from the same class have a predictable set of attributes.

Class Members and Instance Members

We distinguish class/static members from instance members. Class members belong to the class itself; instance members exist in each instantiated object. Each object can have different instance values, while class attributes are shared.

```
class Student:
    university = "Griffith"

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

```
student1 = Student("David")
student2 = Student("Kim")
```

Member	Type	Where it belongs
university	Class attribute	Student class
name	Instance attribute	Each Student object

Student.university and student1.university can both access the class attribute when it has not been shadowed by an instance attribute.

> **Potential pitfall:** Class attributes are shared. Do not use a mutable class attribute such as [] for per-object data unless you intentionally want all objects to share the same list.

Public, Protected and Private Naming

We distinguish public, protected and private members by naming convention.

Form	Name	Interpretation
No underscore	name	Public; normal external use.
One underscore	_name	Protected by convention; subclasses/internal code may use it.
Two underscores	__name	Private-style name; Python applies name mangling.

```
class MyClass:
    def __init__(self, a, b, c):
        self.public = a
        self._protected = b
        self.__private = c
```

We show x.public and x._protected can be read and changed directly, while x.__private raises AttributeError.

We warn against relying on private attributes of another class because implementation details can change.

> **Important nuance:** Protected is mainly a communication convention in Python, not a hard access restriction. Double underscores invoke name mangling; they do not make data cryptographically or absolutely inaccessible.

Encapsulation and Information Hiding

We describe encapsulation as bundling data with the methods that access that data and controlling access to protect the data. Getters and setters can help prevent an invalid or inconsistent state.

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance

    def get_balance(self):
        return self._balance

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

Instead of allowing every part of the program to change the balance arbitrarily, the class provides a controlled operation.

Without control	With encapsulation
Any code may directly change internal state.	Methods can enforce rules.
Invalid states are easier to create.	Validation can be centralised.
Implementation details leak outside.	Users depend mainly on the public interface.

Complete Beginner Example: Puppy

We use a Puppy example to put several ideas together: class, __init__, attributes, method, and multiple objects.

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

    def bark(self):
        print("I am", self.color, self.name)

puppy1 = Puppy("Max", "brown")
puppy1.bark()

puppy2 = Puppy("Ruby", "black")
puppy2.bark()
```

Notice how puppy1 and puppy2 use the same class but have different attribute values.

Association: One Object Refers to Another

We explicitly shows that an object's attributes may be object references. Its Person class has a Name and an Address.

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

    def __str__(self):
        return str(self.name)
```

If name is a Name object and address is an Address object, Person is composed of references to other objects.

```
person.name.firstName()
person.address
```

Relationship test: Person has a Name and an Address. That is association. Student is a Person. That is inheritance.

Inheritance: Building a More Specific Class

We define inheritance as extending an existing class by inheriting all its members and adding or redefining some. The new class is the subclass; the old class is the superclass.

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

```
    def describe(self):
        return "I am a person"


class Student(Person):
    pass

student = Student("Alex")
print(student.describe())
```

Student inherits describe() from Person.

The supplied class diagram (on Section 25 page 22 of lecture material) represents Person at the top, Member beneath it, and Councillor, Staff, Student and Affiliate beneath Member. It visually communicates that a Student is a Member and a Member is a Person.

> **When to use inheritance:** Use it when the subclass genuinely represents a specialised form of the superclass. Do not use inheritance merely because two classes happen to share a few attributes.

Subclass __init__ and super()

The Member example gives Member an additional id while reusing Person's name and address initialisation.

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


class Member(Person):
    def __init__(self, name, address, member_id):
        super().__init__(name, address)
        self.id = member_id
```

The object therefore contains inherited state and subclass-specific state.

State	Initialised by
name	Person.__init__ via super()
address	Person.__init__ via super()
id	Member.__init__

> **Exam insight:** If a subclass defines __init__, the parent constructor is not automatically duplicated inside it. If the inherited initialisation is needed, explicitly call the superclass constructor, commonly with super().__init__(...).

Overriding an Inherited Method

A subclass can redefine an inherited method. The Member example overrides __str__ to include the member ID.

```
class Person:
    def __str__(self):
        return self.name


class Member(Person):
    def __str__(self):
        return self.id + " " + self.name
```

Both classes provide __str__, but Member's version is used for Member objects.

> **Trace rule:** When a method is called, look at the actual object's class first. If that class defines the method, that implementation is used; otherwise Python looks up the inheritance hierarchy.

Multi-Level Inheritance

The example continues from Person to Member and then creates Student and Staff as further subclasses.

```
class Member(Person):
    pass

class Student(Member):
    def exclude(self):
        self.writeALetter("Exclusion message")

class Staff(Member):
    def fire(self):
        self.writeALetter("Termination message")
```

Student can inherit members from both Member and, through Member, Person. Staff follows the same chain.

> **Inheritance chain:** Student → Member → Person. A Student therefore has access to appropriate inherited behaviour from both levels.

Polymorphism

We describe polymorphism as allowing an object or other entity to have more than one form in different contexts. It identifies overloading and overriding as two kinds.

```
class Student:
    def describe(self):
        return "student"


class Staff:
    def describe(self):
        return "staff"


people = [Student(), Staff()]

for person in people:
    print(person.describe())
```

The caller uses one operation, describe(), while different object types provide different results.

> **Practical meaning:** Polymorphism lets code focus on what an object can do rather than repeatedly checking exactly what class it is.

Operator Overloading

We explain that the same operator can have different meanings for different types. + performs numeric addition, string concatenation and list concatenation.

```
print(2 + 3)
print("Hello " + "Python")
print([1, 2] + [3, 4])
```

Classes can also define special methods that determine how operators behave.

Operator	Method
+	__add__
-	__sub__
*	__mul__
**	__pow__
/	__truediv__

The slides explain that p1 - p2 can result in a call to p1.__sub__(p2).

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __sub__(self, other):
        return Point(self.x - other.x,
                     self.y - other.y)
```

This example makes subtraction meaningful for Point objects.

Special Methods

Special methods have double underscores around their names. Python uses them to provide built-in behaviours, and classes can redefine them.

Special method	Triggered by / purpose
__init__	Object initialisation.
__str__	str(object) and print(object).
__len__	len(object).
__add__	object1 + object2.
__sub__	object1 - object2.
__mul__	object1 * object2.
__pow__	object1 ** object2.
__truediv__	object1 / object2.

__str__

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

    def __str__(self):
        return "Student: " + self.name

student = Student("Bob")
print(student)
```

The Name example uses __str__ to make printing a Name object produce its full preferred name.

Rule: __str__ must return a string.

Classes in Modules

We explain that a module is a script that can define functions and classes for another script to import. It also states that the first statement should be a docstring.

```
# student.py
"""A module that defines the Student class."""

class Student:
    """Represents a student."""

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

```
# main.py
from student import Student

student = Student("David", 18)
print(student.name)
```

The lecture material demonstrates this structure with name1.py, address1.py, person1.py, members1.py and separate test-driver programs.

Why separate files? It makes classes reusable and keeps the class definition separate from the program that tests or uses it.

Testing a Class Properly

The lecture material (Section 25) includes separate test-driver programs. This is a useful pattern: define the class in one module and write another program to create objects and exercise the methods. Create an object with normal values.

1. Check each important attribute.
2. Call each method at least once.
3. Create a second object and confirm its state is independent.
4. Test boundary or invalid values where the class has rules.
5. If inheritance is used, test both inherited and overridden methods.
6. If __str__ is defined, test print(object).

```
student = Student("David", 18)

assert student.name == "David"
assert student.age == 18
assert student.get_age() == 18
```

Assertions are one possible testing technique. For introductory exercises, ordinary print statements are also useful when you are learning to trace object state.

How to Design a Class from a Question

For programming questions, do not begin by guessing Python syntax. Extract the design from the English description.

1. Find the main noun representing the object. This is a candidate class name.
2. Find facts or properties that the object must remember. These are candidate attributes.
3. Find actions or verbs that the object must perform. These are candidate methods.
4. Identify information needed when the object is created. These become __init__ parameters.
5. Identify rules that must always be respected. Put the rule-checking logic inside appropriate methods.
6. Look for 'is a' relationships. These may indicate inheritance.
7. Look for 'has a' relationships. These may indicate association.
8. Look for repeated specialised behaviour. This may suggest overriding or polymorphism.
9. Test the design with two objects to make sure their state remains independent.

Example: GoCard

Requirement in problem	Design decision
A card has a balance	balance attribute
A card records transactions	transactions list
A card can ride	ride() method
A card can top up	top_up() method
A card can report balance	get_balance() method
A card can print history	print_statement() method
A pensioner card is a GoCard with different ride pricing	PensionerGoCard subclass
A regular card has a special ride cycle	RegularGoCard subclass with additional state

The Week 10 workshop asks students to identify the class, services and data for this exact style of design before implementing it.

A Reliable Method for Tracing OOP Code

OOP questions can look complicated because several objects may exist at once. Use an object-state table.

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

    def add(self, amount):
        self.value += amount

a = Counter(5)
b = Counter(10)
a.add(2)
b.add(4)
```

Object	Initial value	Call	Final value
a	5	a.add(2)	7
b	10	b.add(4)	14

Do not combine the state of a and b. Each object has its own instance attribute.

1. When an object is created, write its initial attribute values.
2. For every method call, identify self.
3. Change only the attributes of that self object.
4. For an overridden method, use the subclass version.
5. For super(), follow the parent method before continuing with subclass-specific statements.
6. For print(object), check __str__.

Common Mistakes and How to Fix Them

Mistake	Why it is wrong	Fix
def get_age(age):	The method has no reference to the current object.	Use def get_age(self):
age = age in __init__	Does not create an instance attribute.	Use self.age = age
student1.get_age without ()	Refers to the method instead of calling it.	Use student1.get_age()
Calling Student.get_age() directly	No instance is supplied as self.	Normally use student.get_age()
Forgetting self when reading state	The method cannot identify which object's state to use.	Use self.attribute
Using a shared class list for per-object data	All objects may unintentionally share the same list.	Create the list in __init__
Duplicating parent initialisation	Makes inheritance harder to maintain.	Use super().__init__()
Confusing 'has a' and 'is a'	Leads to the wrong relationship.	Association for has-a; inheritance for is-a
Returning an int from __str__	print() expects __str__ to return a string.	Return a string
Assuming _name is truly inaccessible	Single underscore is mainly a convention.	Treat it as protected by convention

Week 10 GoCard: Putting the Ideas Together

The GoCard case study is particularly useful because one problem brings together attributes, methods, encapsulation, lists of transaction records, inheritance and overriding. The workshop extends a basic GoCard into RegularGoCard and PensionerGoCard with different ride behaviour.

```
class GoCard:
    def __init__(self, balance):
        self.balance = balance
        self.transactions = []

    def top_up(self, amount):
        self.balance += amount

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

```
class PensionerGoCard(GoCard):
    def ride(self, amount):
        self.balance -= amount * 0.95
```

Read this as:

- GoCard owns common account state.
- GoCard provides common operations.
- PensionerGoCard is a GoCard, so inheritance is appropriate.
- PensionerGoCard changes ride behaviour by overriding ride().
- A program can call account.ride(amount) without needing to know the exact subclass implementation.

Self-Check Questions

Q1. What is the difference between a class and an object?

Answer: A class is the definition/blueprint; an object is a concrete instance created from that class.

Q2. Why is self needed?

Answer: It refers to the current object, allowing an instance method to access or change that object's attributes.

Q3. What does __init__ do?

Answer: It initialises a newly created object with its required initial state.

Q4. Why do we write self.age rather than age?

Answer: self.age is stored in the object; age alone is a parameter/local variable.

Q5. What is an instance attribute?

Answer: An attribute belonging to one particular object.

Q6. What is a class attribute?

Answer: An attribute associated with the class and shared unless an instance provides its own value.

Q7. What is association?

Answer: A relationship where one object refers to or contains another object.

Q8. What is inheritance?

Answer: A subclass extends a superclass by inheriting and possibly adding or redefining members.

Q9. What does overriding mean?

Answer: A subclass supplies its own implementation of an inherited method.

Q10. What is polymorphism?

Answer: Different objects can respond to the same operation in different ways.

Q11. What is __str__ used for?

Answer: It provides a string representation used by str() and print().

Q12. What does super().__init__() achieve?

Answer: It calls the superclass constructor so inherited state can be initialised.

Final Revision Map

If the question says...	Think...
Define a blueprint/template	class
Create one actual entity	object / instance
Store information about an object	attribute
Perform an operation on an object	method
Initialise an object	__init__
Refer to the current object	self
One object contains another	association
A specialised type is also a general type	inheritance
Use the parent's initialisation	super().__init__()
Subclass changes inherited behaviour	override
Same method call, different object behaviour	polymorphism
Make +, -, etc. work for your class	operator overloading
Control internal state	encapsulation
Make print(object) useful	__str__
Reuse class definitions in another file	module/import

Exam strategy: Before coding, underline the nouns, verbs and relationships in the question. Nouns often suggest classes/attributes, verbs suggest methods, 'is a' suggests inheritance, and 'has a' suggests association.

Week 10 Final Checklist

- I can explain class, object and instance without confusing them.
- I can write a class with __init__ and instance attributes.
- I understand self and can trace which object it refers to.
- I can define and call methods.
- I can explain the difference between instance and class attributes.
- I understand public, protected and private naming conventions.
- I can explain encapsulation and why controlled access is useful.
- I can model a has-a relationship with object references.
- I can model an is-a relationship with inheritance.
- I can initialise inherited state with super().__init__().
- I can override a method in a subclass.
- I can explain polymorphism using a common method call.
- I understand operator overloading and special methods.
- I can define __str__ correctly.
- I understand how classes can be placed in modules and imported.
- I can design a class from an English problem description.
- I can trace multiple objects without mixing their state.
- I can identify and fix common OOP mistakes.

End of Week 10 Student Chapter