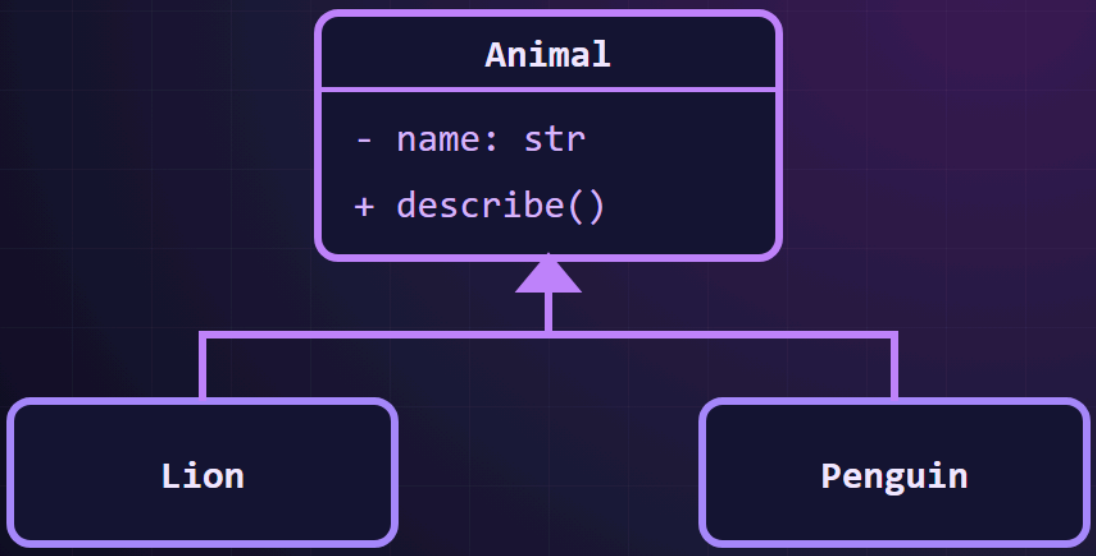


CLASSES · OBJECTS · DESIGN

Object-Oriented Programming Made Simple

Classes, objects, and design — explained with practical examples

From your first class to SOLID and design patterns — with exercises and solutions in every chapter

CLASSES · INHERITANCE · SOLID · PATTERNS

THENEXBOOKS

Object-Oriented Programming Made Simple

Classes, Objects, and Design Explained with Practical Examples

From Your First Class to SOLID Principles and Design Patterns

First Edition

THENEXBOOKS

Object-Oriented Programming Made Simple
Classes, Objects, and Design Explained with Practical Examples

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples and exercise solutions may be used freely for learning and in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Examples use Python 3.10 or later, with short comparisons in other languages; always consult the official documentation for the versions you use.

Python is a registered trademark of the Python Software Foundation. Java is a trademark of Oracle and/or its affiliates. C# and TypeScript are associated with Microsoft. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement or affiliation.

Mossgate Zoo, the running example in this book, is fictional. Any resemblance to real organizations or people is coincidental.

Preface

Object-oriented programming is one of the most widely taught ideas in software — and one of the most often misunderstood. Many learners can write a class that compiles but are not sure *why* it should exist, what belongs inside it, or how several classes should work together. The vocabulary does not help: encapsulation, polymorphism, abstraction, composition, interfaces, SOLID, design patterns. Each word sounds more intimidating than the idea behind it.

This book makes those ideas simple without making them shallow. We start from the problem OOP solves — keeping data and the rules that govern it together — and build up one concept at a time, always with runnable Python code and a realistic example. When a concept looks different in Java, C#, or TypeScript, we show a short side-by-side comparison so the idea carries over to whatever language you use next.

Every chapter ends with graded exercises and worked solutions. Along the way you will design and build the software for a small zoo, a text adventure game, and a shop inventory system. By the end you will not only know what the words mean — you will be able to design classes that are easy to understand, test, and change.

Who This Book Is For

- **Beginners who know the basics** — variables, conditions, loops, functions, and lists — and want to learn OOP properly.
- **Students** in a programming or software-design course who want clearer explanations and more practice.
- **Self-taught developers** who use classes from frameworks but want to design their own with confidence.
- **Developers moving between languages** who want the concepts, not just one language's syntax.

What You Will Learn

- What classes and objects are, and how to design a class with clear responsibilities.
- Encapsulation, abstraction, inheritance, and polymorphism — and when to prefer composition.
- Abstract classes, interfaces, protocols, special methods, and class-level members.
- How objects work together, identity and equality, immutability, and exceptions in object-oriented code.
- Class diagrams, the SOLID principles, cohesion and coupling, and common design patterns.
- Testing, refactoring, persistence, comparisons across languages, and complete projects.

How This Book Is Organized

Part I — Thinking in Objects (Chapters 1–5) introduces classes, objects, and encapsulation. **Part II — The Four Pillars and Beyond** (Chapters 6–10) covers abstraction, inheritance, polymorphism, composition, and interfaces. **Part III — Working with Objects** (Chapters 11–15) explores special methods, class members, relationships, equality, and exceptions. **Part IV — Designing Good Object-Oriented Code** (Chapters 16–20) teaches diagrams, principles, and patterns. **Part V — OOP in Practice** (Chapters 21–25) covers testing, refactoring, other languages, persistence, and common mistakes. **Part VI — Projects and Case Study** (Chapters 26–28) builds three complete programs. The appendices provide a cheat sheet, templates and a pattern reference, a design review checklist, and a glossary.

Each chapter ends with **Practice Exercises**, **Selected Solutions**, and a **Key Takeaways** box.

The Running Example: Mossgate Zoo

Throughout the book we build software for **Mossgate Zoo**, a fictional small zoo. Animals, enclosures, keepers, feeding schedules, visitor tickets, and a gift shop give us natural examples of objects with state and behavior, of inheritance and composition, and of objects that must cooperate. In the final chapter these classes come together as a complete zoo management system.

Conventions

Code appears in `monospace`, with program output shown in comments or a separate block. Examples use Python 3.10 or later; short Java, C#, and TypeScript snippets are labeled. Exercises are marked **[Easy]**, **[Medium]**, or **[Challenge]**.

> **TIP**
> A recommended practice, or a "Try It" prompt to pause and experiment.

> **NOTE**
> Background information or a useful aside.

> **WARNING**
> A common design or coding mistake to avoid.

Contents

Preface

PART I — THINKING IN OBJECTS

1 Why Object-Oriented Programming? From Procedures to Objects

2 Objects and Classes: The Big Idea

3 Your First Class: Attributes, Methods, and __init__

4 Designing a Class: Responsibilities, State, and Behavior

5 Encapsulation: Protecting State with Methods and Properties

PART II — THE FOUR PILLARS AND BEYOND

6 Abstraction: Hiding Complexity Behind Simple Interfaces

7 Inheritance: Reusing and Extending Classes

8 Polymorphism: One Interface, Many Forms

9 Composition over Inheritance: Building Objects from Objects

10 Abstract Classes, Interfaces, and Protocols

PART III — WORKING WITH OBJECTS

11 Special Methods: Making Objects Behave Like Built-ins

12 Class Attributes, Static Methods, and Class Methods

13 Objects Working Together: Relationships, Collections, and Associations

14 Identity, Equality, Copying, and Immutability (with dataclasses)

15 Exceptions in Object-Oriented Code: Custom Errors and Safe Objects

PART IV — DESIGNING GOOD OBJECT-ORIENTED CODE

16 Class Diagrams and UML Made Simple

17 The SOLID Principles in Plain Language

18 Cohesion, Coupling, and Keeping Objects Independent

19 Design Patterns I: Creating Objects (Factory, Builder, Singleton)

20 Design Patterns II: Structural and Behavioral (Adapter, Decorator, Strategy, Observer, Command)

PART V — OOP IN PRACTICE

21 Testing Object-Oriented Code

22 Refactoring Procedural Code into Objects

23 OOP Across Languages: Python, Java, C#, and TypeScript Compared

24 Saving and Loading Objects: Files, JSON, and Databases

25 Common OOP Mistakes and How to Avoid Them

PART VI — PROJECTS AND CASE STUDY

26 Project: A Text-Based Adventure Game with Objects

27 Project: A Shop Inventory and Ordering System

28 Case Study: Building the Mossgate Zoo Management System

APPENDICES

A OOP Cheat Sheet

B Class Design Templates and Pattern Reference

C Object-Oriented Design Review Checklist

D Glossary

PART I

Thinking in Objects

Why objects exist, what classes and objects are, how to write and design your first classes, and how encapsulation protects their state.

1 · Why Object-Oriented Programming? From Procedures to Objects
2 · Objects and Classes
3 · Your First Class
4 · Designing a Class
5 · Encapsulation

CHAPTER 1 · PART I — THINKING IN OBJECTS

Why Object-Oriented Programming? From Procedures to Objects

Before we learn any new syntax, we need to understand the problem that object-oriented programming solves. In this chapter we write a small program for a zoo the way most beginners do, watch it become harder to change as it grows, and then see how organizing the same code around objects keeps it tidy.

Meet Mossgate Zoo

Throughout this book we will build software for **Mossgate Zoo**, a small fictional zoo. The zoo has animals, enclosures where the animals live, keepers who look after them, feeding schedules, visitor tickets, and a gift shop. It is a friendly, familiar domain: you already know what a lion is, what a ticket is, and what it means for an enclosure to be full. That lets us concentrate on the programming ideas instead of on the subject matter.

A few residents will appear again and again:

- **Leo** the lion, 8 years old, eats meat.
- **Pip** the penguin, 3 years old, eats fish.
- **Tessa** the tortoise, 62 years old, eats plants.
- **Ruby** the red panda, 4 years old, eats plants (mostly bamboo).

Visitors buy tickets: an adult ticket costs $18.00, a child ticket costs $9.00, and a family ticket costs $45.00. Keepers have staff IDs such as `"K07"`. By the last chapter of this book, all of these pieces will fit together into one well-designed system. For now we start small, with just the animals.

A First Version: The Procedural Way

Suppose the head keeper asks for a tiny program that lists the animals and tells the kitchen how much food each one needs per day. If you know variables, lists, dictionaries, loops, and functions, you can already write it. A very natural first attempt stores each animal as a dictionary and writes functions that work on those dictionaries.

zoo_procedural.py

```
animals = [
    {"name": "Leo", "species": "lion", "age": 8, "diet": "meat"},
    {"name": "Pip", "species": "penguin", "age": 3, "diet": "fish"},
    {"name": "Tessa", "species": "tortoise", "age": 62, "diet": "plants"},
]
```

```
def describe_animal(animal):
    return f"{animal['name']} the {animal['species']}, age {animal['age']}"


def daily_food_kg(animal):
    if animal["species"] == "lion":
        return 7.0
    elif animal["species"] == "penguin":
        return 1.0
    elif animal["species"] == "tortoise":
        return 0.5
    return 0.0


def have_birthday(animal):
    animal["age"] = animal["age"] + 1


for animal in animals:
    print(describe_animal(animal), "-", daily_food_kg(animal), "kg/day")

have_birthday(animals[0])
print(describe_animal(animals[0]))
```

Running it prints:

```
Leo the lion, age 8 - 7.0 kg/day
Pip the penguin, age 3 - 1.0 kg/day
Tessa the tortoise, age 62 - 0.5 kg/day
Leo the lion, age 9
```

There is nothing wrong with this program. It is short, it works, and it is easy to read. This style is called **procedural programming**: the program is a sequence of steps (procedures, which Python calls functions) that act on data stored somewhere else. The data and the functions are two separate things. The dictionaries do not "know" anything; they just hold values. The functions do all the work, and each function must be handed the data it needs.

Procedural programming is a perfectly good way to write software. Many small scripts, command-line tools, and data-processing jobs are best written this way. The question is not "is procedural code bad?" but "what happens as the program grows?"

The Program Grows

Software rarely stays small. A month later the zoo asks for more features: keepers, enclosures, feeding times, medical checks, and tickets. Every new feature adds more dictionaries and more functions. Let us imagine a few of the changes and watch where the pain appears.

Problem 1: Every function must know the shape of the data

Each function above reaches inside the dictionary with keys such as `"name"` and `"age"`. If we later decide to rename the key `"species"` to `"kind"`, we must find every function in every file that uses that key. Nothing in the language helps us find them all. Miss one, and the program crashes only when that particular function runs, which might be weeks later.

A typo has the same effect. Python happily runs this line until the moment it executes:

```
print(animals[1]["Age"])   # capital A by mistake
# KeyError: 'Age'
```

Problem 2: Nothing protects the data

Any line of code anywhere in the program can change any dictionary in any way. One function might set an age to `-5`. Another might store the age as the string `"eight"`. A third might delete the `"diet"` key entirely. The data has no guardian. When something goes wrong, you have to search the whole program to find who broke it.

```
animals[2]["age"] = -5          # nobody stops this
del animals[1]["diet"]          # or this
print(describe_animal(animals[2]))
# Tessa the tortoise, age -5
```

Problem 3: Decisions are spread out in long if-chains

Look again at `daily_food_kg`. It asks "which species is this?" and then chooses an answer. Now imagine the zoo adds a red panda. We must edit `daily_food_kg`. Then we notice that `feeding_time`, `enclosure_temperature`, and `vet_check_interval` all have their own `if species == ...` chains, so we must edit those too. Knowledge about "what a red panda needs" ends up scattered across many functions in many files.

```
def feeding_time(animal):
    if animal["species"] == "lion":
        return "16:00"
    elif animal["species"] == "penguin":
        return "10:30"
    elif animal["species"] == "tortoise":
        return "09:00"
    return "12:00"


def enclosure_temperature(animal):
    if animal["species"] == "lion":
        return 24
    elif animal["species"] == "penguin":
        return 4
    # ... and so on, for every species, in every function
```

Problem 4: Related things drift apart

In a larger procedural program, the animal data might live in `data.py`, the food functions in `kitchen.py`, the birthday logic in `records.py`, and the display code in `reports.py`. To understand "what is an animal in this program?" you need to read all four files. There is no single place that answers the question.

None of these problems is a disaster on its own. Together they make a growing program slow and risky to change. Each new feature takes a little longer than the last one, and each change has a slightly higher chance of breaking something far away. Object-oriented programming was developed largely to fight exactly this kind of slow decay.

> **NOTE**
> Programmers sometimes describe code that tangles like this as "spaghetti". It is not a technical term, but it captures the feeling: pull one strand and something on the other side of the plate moves.

The Object-Oriented Idea

The central idea of object-oriented programming (OOP) fits in one sentence:

Keep data together with the code that works on that data, inside units called objects.

Instead of a dictionary that holds an animal's values and a pile of separate functions that manipulate dictionaries, we create an **Animal** object that holds its own values *and* knows how to describe itself, how to have a birthday, and how much food it needs. The data and the behavior travel together.

Here is the same zoo program rewritten in an object-oriented style. Do not worry about every keyword yet. The words `class`, `self`, and `__init__` are explained slowly in Chapters 2 and 3. For now, just compare the *shape* of this code with the procedural version.

zoo_objects.py

```
class Animal:
    def __init__(self, name, species, age, diet):
        self.name = name
        self.species = species
        self.age = age
        self.diet = diet

    def describe(self):
        return f"{self.name} the {self.species}, age {self.age}"

    def have_birthday(self):
        self.age = self.age + 1


leo = Animal("Leo", "lion", 8, "meat")
pip = Animal("Pip", "penguin", 3, "fish")
```

```
print(leo.describe())
leo.have_birthday()
print(leo.describe())
print(pip.describe())
```

Output:

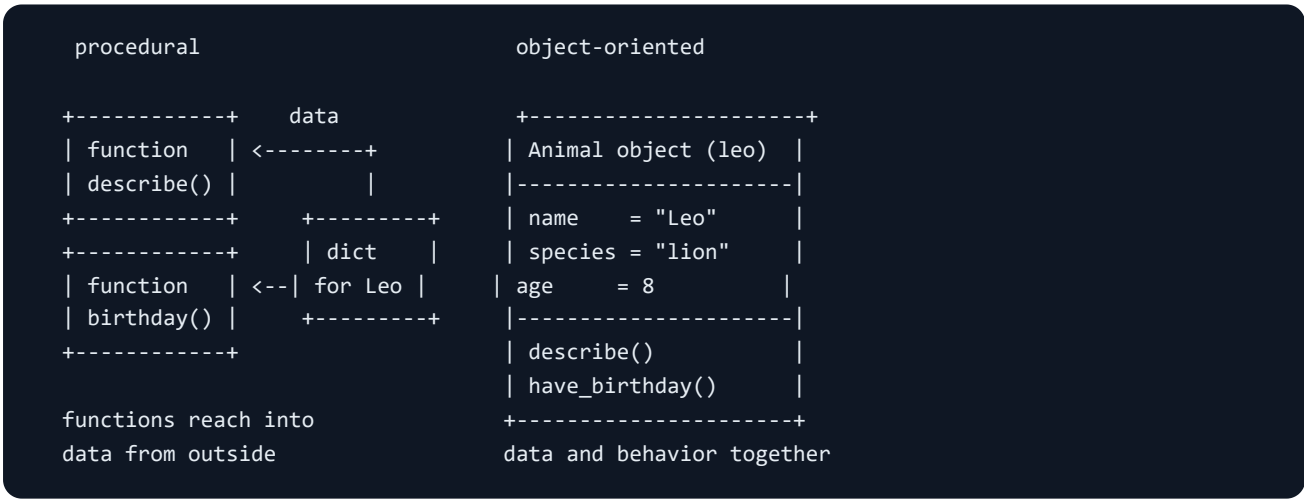
```
Leo the lion, age 8
Leo the lion, age 9
Pip the penguin, age 3
```

Notice three things:

1. **One place defines what an animal is.** The `class Animal:` block lists the data an animal has (name, species, age, diet) and the actions it can perform (describe, have a birthday). To answer "what is an animal in this program?" you read one block of code.
2. **We ask the object to do things.** Instead of `describe_animal(leo)` we write `leo.describe()`. The dot means "Leo, please describe yourself." The object uses its own data to answer.
3. **Attribute names are written once.** Inside the class, the code says `self.age`, and outside it says `leo.age`. There are no string keys to mistype in dozens of places. If you do mistype an attribute, Python raises an `AttributeError` that names the object type and the missing attribute, which makes the mistake easier to spot.

Objects send each other messages

A useful way to picture an object-oriented program is as a group of objects that talk to each other. When the code says `leo.have_birthday()`, we can read it as "send the message *have_birthday* to the object *leo*." The object receives the message and decides how to respond by running one of its methods. A **method** is simply a function that belongs to an object.



On the left, the functions reach into a separate data structure. On the right, the data and the functions that use it live inside one box. That box is an object.

Tracing one message step by step

Let us slow down and follow exactly what happens when Python runs `leo.have_birthday()`. Being able to trace code like this is the most useful skill you can build in the early chapters.

1. Python looks up the name `leo` and finds the `Animal` object created earlier, whose `age` is currently 8.
2. The dot tells Python to look for something called `have_birthday` that belongs to this object. It finds the method defined in the `Animal` class.
3. Python calls that method and automatically passes the object itself as the first argument, which the method calls `self`. Inside the method, `self` *is* Leo.
4. The line `self.age = self.age + 1` reads Leo's age (8), adds one, and stores 9 back into Leo's own `age`.
5. The method ends. Pip's age is untouched, because Pip is a different object with his own data.

The same sequence happens for every method call you will ever write: find the object, find the method, pass the object in as `self`, run the method. Chapter 3 explores `self` in much more detail, including the error messages you get when it is forgotten.

Compare this with what happens when you mistype an attribute name on an object. Instead of a vague `KeyError`, Python tells you which kind of object was involved:

```
print(leo.nmae)
# AttributeError: 'Animal' object has no attribute 'nmae'
```

The message says, in effect, "you asked an `Animal` for something it does not have." Recent versions of Python often add a helpful hint such as "Did you mean: 'name'?" at the end of the message. Error messages that name the class are one small but real everyday benefit of working with objects.

> **TRY IT**
>
> Type `zoo_objects.py` into a file and run it. Then create a third animal, `tessa = Animal("Tessa", "tortoise", 62, "plants")`, and print `tessa.describe()`. Call `have_birthday()` on Tessa twice and print her description again. Notice that Leo's age does not change when Tessa has a birthday.

Solving the If-Chain Problem

The procedural version had a `daily_food_kg` function full of `if species == ...` checks. OOP offers a cleaner way: let each kind of animal carry its own answer. Here is a small preview of a technique called **inheritance** (Chapter 7) combined with **polymorphism** (Chapter 8). You do not need to understand the details yet; just look at where the knowledge lives.

zoo_food_preview.py

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

    def daily_food_kg(self):
        return 0.0


class Lion(Animal):
    def daily_food_kg(self):
        return 7.0


class Penguin(Animal):
    def daily_food_kg(self):
        return 1.0


class Tortoise(Animal):
    def daily_food_kg(self):
        return 0.5


residents = [Lion("Leo", 8), Penguin("Pip", 3), Tortoise("Tessa", 62)]
total = 0.0
for animal in residents:
    total = total + animal.daily_food_kg()
print("Kitchen order:", total, "kg")
```

Output:

```
Kitchen order: 8.5 kg
```

The loop at the bottom never asks "what species is this?" It simply asks each animal for its daily food, and each animal answers in its own way. When the zoo adds a red panda, we write one new class, `RedPanda`, with its own `daily_food_kg`. We do not have to edit the loop or any existing class. Everything the program knows about red pandas sits in one place.

This ability to add new kinds of things without editing the old code is one of the biggest practical benefits of OOP. Later chapters show when to use it and, just as importantly, when a simpler approach is better.

What Exactly Is an Object?

We will define objects carefully in Chapter 2, but here is a first working definition. An **object** is a thing in your program that has:

Part	Meaning	Leo example
State	The data it holds right now	name "Leo", age 8
Behavior	What it can do (its methods)	describe, have a birthday
Identity	It is a distinct thing	Leo is not Pip

A **class** is the definition, or blueprint, from which objects are made. The class `Animal` describes what every animal has and can do; the objects `leo` and `pip` are two specific animals built from that blueprint. Programmers say that `leo` is an **instance** of the class `Animal`.

You Have Been Using Objects All Along

Here is a secret: if you have written any Python at all, you have already used objects. In Python, *every* value is an object. Strings, numbers, lists, and dictionaries are all objects with their own methods.

```
name = "mossgate zoo"
print(name.upper())          # MOSSGATE ZOO
print(name.title())          # Mossgate Zoo

species = ["lion", "penguin"]
species.append("tortoise")   # the list changes itself
print(species)               # ['lion', 'penguin', 'tortoise']

print(type(name))            # <class 'str'>
print(type(species))         # <class 'list'>
print(type(42))              # <class 'int'>
```

When you write `name.upper()`, you are sending the message *upper* to a string object. The string knows how to produce an upper-case copy of itself. When you write `species.append("tortoise")`, you are asking a list object to add an item to itself. The `type()` function even tells you the class each object belongs to: `str`, `list`, and `int` are classes built into Python.

So OOP is not a strange new world. It is the same pattern you already use with built-in types, extended so that you can create your *own* types, such as `Animal`, `Enclosure`, and `Ticket`, that fit your problem.

> **TRY IT**
>
> In the Python interactive shell, type `dir("hello")`. You will see a long list of names. Most of them, such as `upper`, `split`, and `replace`, are methods that every string object carries. Try calling three you have never used before.

A Short History, Without the Dates Quiz

Object-oriented ideas appeared in the 1960s in a language called Simula, which was designed for simulations: programs that model real-world things such as ships, customers, or machines. Modeling each real-world thing as a software "object" turned out to be a natural fit. In the 1970s the language Smalltalk pushed the idea further and popularized the picture of objects sending messages to each other.

Later, object-oriented features spread into mainstream languages. C++ added classes to C, and Java and C# were designed around classes from the start. Python, JavaScript, Ruby, Kotlin, Swift, and many others all support objects, each in its own style. Learning OOP well in Python gives you ideas you can carry to almost any modern language.

The Four Pillars at a Glance

Books and courses often summarize OOP with four core ideas, sometimes called the "four pillars". Each has its own chapter in Part II. Here is a one-line preview so that the names will feel familiar when you meet them.

Pillar	Plain meaning	Chapter
Encapsulation	Protect an object's data behind its methods	5
Abstraction	Show a simple interface, hide the details	6
Inheritance	Build a new class from an existing one	7
Polymorphism	Different objects answer the same message	8

You have already seen tiny previews of three of them. The `Animal` class keeps each animal's data together with its methods (the start of encapsulation). The kitchen loop asked each animal for `daily_food_kg()` without caring how the answer was calculated (abstraction and polymorphism). And `Lion(Animal)` built a new class from an existing one (inheritance).

What OOP Gives You (and What It Does Not)

It is easy to oversell OOP, so let us be precise about what it actually offers.

Real benefits

- **Organization.** Code about animals lives in the `Animal` class; code about tickets lives in the `Ticket` class. You know where to look.
- **Protection.** An object can check changes to its own data. An `Animal` can refuse a negative age. Chapter 5 shows how.
- **Change in one place.** When knowledge lives in one class, a change to that knowledge happens in one class.

- **Extension without editing.** New kinds of objects can often be added without touching working code, as the `RedPanda` example suggested.
- **A shared vocabulary.** Classes named after real concepts (Keeper, Enclosure, Ticket) make code easier to discuss with non-programmers, such as the zoo's staff.

Honest limits

- **OOP does not make bad design good.** You can write tangled code with classes too. Part IV of this book is about designing classes well.
- **Small scripts may not need it.** A 30-line script that renames files is often clearer as a few functions.
- **It is not the only paradigm.** Functional programming, for example, organizes code around functions that avoid changing data. Python lets you mix styles, and good Python code often does.

> **WARNING**
>
> A common beginner mistake after learning OOP is to put *everything* into classes, even a function that just adds two numbers. A class should earn its place by grouping data with behavior. If there is no data to protect or organize, a plain function is usually the better choice.

When Should You Reach for Objects?

A few questions help you decide whether a piece of your program should become a class:

- Do you have several values that always travel together (a name, species, age, and diet)?
- Are there rules about those values (age cannot be negative, an enclosure cannot exceed its capacity)?
- Are there actions that naturally belong to that group of values (have a birthday, add an animal)?
- Will there be many of these things (dozens of animals, thousands of tickets)?
- Are there different kinds of the thing that behave slightly differently (lions and penguins eat differently)?

If you answer "yes" to two or more, a class is probably a good fit. If you answer "no" to all of them, a function or a simple data structure is likely enough.

A Glimpse at Other Languages

Python lets you choose between procedural and object-oriented styles freely. Some languages are stricter. In Java, every function must live inside a class, even the program's starting point. Here is Leo's class in Java, just to show that the same idea looks similar across languages:

Animal.java

```
public class Animal {
    private String name;
    private int age;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String describe() {
        return name + ", age " + age;
    }

    public void haveBirthday() {
        age = age + 1;
    }
}
```

The Java version declares the type of every value (`String`, `int`) and uses the keyword `private` to hide the data from other classes. The part named `Animal(...)` inside the class is a **constructor**, which plays the role of Python's `__init__`. The word `this` plays the role of Python's `self`. C# looks almost the same. Chapter 23 compares the languages side by side; for most of this book we stay in Python, where the ideas are easiest to see.

Common Misunderstandings

"OOP means using the class keyword"

Writing `class` does not by itself make code object-oriented. A class that holds nothing but a dozen unrelated functions is just procedural code in a box. OOP is about *responsibility*: each object looks after its own data and offers useful behavior. Chapter 4 teaches how to decide what each class is responsible for.

"Objects must model physical things"

Lions and tickets are easy examples because you can picture them. But many useful objects are abstract: a `FeedingSchedule`, a report generator, a connection to a database, or a rule for calculating discounts. If it has state and behavior that belong together, it can be an object.

"OOP is slower, so avoid it"

For the kinds of programs beginners and most professionals write, the organizational benefits matter far more than tiny differences in speed. Write clear code first. If a program is ever too slow, measure it to find out where the time actually goes before changing anything.

"Inheritance is the main point of OOP"

Inheritance is famous, but it is only one tool, and it is easy to overuse. Many experienced developers prefer to build objects out of other objects, an approach called **composition**. Chapter 9 explains why the principle "composition over inheritance" is so widely recommended.

From Procedures to Objects: A Side-by-Side Summary

Question	Procedural	Object-oriented
Where is data?	In dicts, lists, globals	Inside objects
Where is behavior?	In separate functions	In methods on objects
How is work done?	Call a function, pass data	Ask an object
Who guards data?	Nobody	The object itself
Adding a species	Edit many if-chains	Add one class
Best for	Small, linear tasks	Growing, rich domains

Remember that this is a comparison of tendencies, not a law. Well-organized procedural code can be very maintainable, and poorly designed classes can be a mess. The goal of this book is to help you design objects that really do deliver the benefits in the right-hand column.

> **TRY IT**
> Take the procedural program from the start of this chapter and add Ruby the red panda (4 years old, eats plants, 1.2 kg of food per day). Count how many places you had to change. Then add a `RedPanda` class to `zoo_food_preview.py` and count again.

Practice Exercises

1. **[Easy]** In your own words, write two sentences explaining the difference between procedural and object-oriented programming. Use the words "data" and "behavior".
2. **[Easy]** For each of these built-in values, name its class using `type()` and call one method on it: `"Leo"`, `[8, 3, 62]`, `{"adult": 18.0}`, `3.5`.
3. **[Easy]** List the state and the behavior of a zoo ticket. Name at least three pieces of state and two behaviors.
4. **[Easy]** In `zoo_objects.py`, create Ruby the red panda and print her description. Then call `have_birthday()` on her and print it again.
5. **[Medium]** Add a function `is_senior(animal)` to the procedural program that returns `True` when the animal's age is 15 or more. Then add the same behavior as a method `is_senior(self)` in the `Animal` class of `zoo_objects.py`. Compare how each is called.