Basics of Python Programming for Pharmaceutical Sciences

B.Pharm I Semester
BP101T

As per New PCI Syllabus 2026 (NEP 2020)

Authors

Komal Deokar Rajgude
Vaibhavi Umesh Avachat
Mahadevan R Swamy

Copyright © 2026 Wiseleaf Scientific Ventures Pvt. Ltd.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher, except for brief quotations in reviews or scholarly works permitted by copyright law.

ISBN: 978-81-686946-9-9
e-ISBN: 978-81-686946-2-0
May, 2026.

Published by: Wiseleaf Scientific Ventures Pvt. Ltd.
Address: Wework - Berger Delhi One Floor 19, C-001/A2, Sector 16b, Noida, Uttar Pradesh, 201301, India

Preface

Technology has revolutionized every corner of science, and pharmacy is no different. Today's pharmaceutical professionals need more than just chemistry knowledge they need to understand data, manage information, and leverage computational tools. That's where Python comes in.

Python has become the go-to language for researchers, data analysts, and professionals across healthcare and pharmaceutical sciences. Whether you're analyzing drug data, exploring bioinformatics, studying formulations, or diving into AI-powered drug discovery, Python skills will set you apart. And here's the good news: you don't need a computer science background to learn it.

This book is written specifically for B.Pharm students in their first semester including those who've never coded before. We've designed it to make Python accessible, practical, and relevant to pharmacy. Rather than overwhelming you with complex theory, we focus on the fundamentals you actually need: variables, data types, operators, loops, functions, and more. Each concept is explained simply and supported with real-world examples from pharmaceutical sciences.

What You'll Learn

You'll work through the core building blocks of Python programming from basic syntax to lists, dictionaries, and file handling. But more importantly, you'll see *why* these skills matter for pharmacy. We've woven in practical applications that show how programmers and researchers use these tools to solve real problems in the field.

Our Promise

Our goal is simple: build your confidence as a programmer and help you see that computational thinking is just as important as lab skills in modern pharmacy. We've worked hard to keep explanations clear and examples straightforward, but we know there's always room to improve.

If you find areas that could be explained better, or if you have suggestions for future editions, please share them. Feedback from students and instructors makes this book stronger.

Thank You

We're grateful to the many colleagues, mentors, and students who supported us in creating this resource. Their encouragement and insights shaped every chapter. We hope this book inspires you to explore the exciting space where programming meets pharmacy and maybe even sparks your interest in innovation and research in healthcare. Let's get started!

Komal Deokar Rajgude
Vaibhavi Umesh Avachat
Mahadevan R Swamy

Course Objectives:

The objectives of this course are to:

1. Introduce the fundamentals of Python programming for pharmaceutical sciences.
2. Develop basic programming skills using control structures, functions, and data structures.
3. Provide knowledge of data handling techniques for structured dataset management.
4. Familiarize students with data analysis tools such as NumPy and Pandas for healthcare datasets.
5. Enable students to visualize and interpret pharmaceutical data.

Course Outcomes (CO):

Upon successful completion of this course, the students will be able to:

1. Explain the fundamentals of Python programming, including variables, data types, operators, and libraries.
2. Analyze program logic using control structures and functions.
3. Organize, manipulate, and retrieve data using data structures and file handling techniques.
4. Analyze pharmaceutical datasets using Python libraries.
5. Visualize and interpret pharmaceutical data using graphical tools.

Syllabus

Unit	Topics	No. of Lectures
I	**Introduction to Python programming** • Installing Python and an Integrated Development Environment (IDE) [Jupyter Notebook, PyCharm, VS Code etc.], Advantages of IDEs over text editors. • Python variables and data types (integers, floats, strings, booleans), Type casting and basic operators (arithmetic, comparison, logical), Input and output operations. • Basic string operations and manipulation techniques. Introduction to standard libraries and third-party libraries, installing and uninstalling libraries.	6 hours
II	**Control Structures & Functions** • Conditional statements (if, if-else, if-elif-else), nested conditions • Loops (for loop, while loop). • Break and continue statements. • Defining and calling functions, passing arguments and returning values. • Writing modular programs for simple pharmaceutical applications- dosage calculation and BMI calculation.	6 hours
III	**Data Structures & File Handling** • Lists, tuples, and dictionaries. • Indexing and slicing lists, basic operations on lists and dictionaries, string manipulation techniques. • Introduction to NumPy arrays, basic operations using NumPy (array creation, arithmetic operations). • Reading and writing CSV files. • Understanding structured healthcare datasets. • Importing small pharmaceutical datasets and performing basic data access and manipulation tasks.	6 hours
IV	**Data Handling with Pandas** • Introduction to Pandas library. • Pandas Series and DataFrame structures. • Reading CSV and Excel files-PK study datasets and ADR reports • Inspecting datasets using functions such as head(), tail(), info(), and describe(). • Data cleaning techniques and handling missing values. • Filtering and selecting data based on conditions. • Grouping data and performing aggregation functions.	6 hours
V	**Data Visualization with Matplotlib** • Introduction to Matplotlib. • Creating line plots, histograms, scatter plots, and box plots. • Labeling axes, titles, and legends. • Create plots and visualize pharmaceutical datasets - concentration-time curves for oral and IV administration, ADR reporting rates across drugs, dissolution profiles. • Scientific interpretation of plots.	6 hours

How to use this book

The book is shaped, more or less, like the BP101T syllabus itself: five units, six lecture-hours each. There is no clever new ordering here. The aim is to make it possible for a lecturer to teach from this book directly, page by page, with very little adaptation.

Reading from front to back (recommended for beginners)

If this is your first encounter with programming, please read the book sequentially. Each unit leans on the one before it, and Unit 1 quietly sets up the vocabulary that the rest of the book uses without further explanation. Do one unit a week. Do not rush. And (this is important) type out every code listing yourself rather than only reading it. Watching someone tie a shoelace is no substitute for trying it yourself.

Reading selectively (for revision or quick reference)

If you have done some Python already and are reaching for the book the night before a paper, go straight to the chapter you need. Each chapter opens with a short list of learning outcomes and ends with a chapter-summary box, MCQs, short-answer questions and a couple of lab exercises. The summary boxes are deliberately compact. You should be able to revise an entire chapter from one in roughly fifteen minutes.

Icons used inside the chapters

> **PHARMA EXAMPLE**
> These boxes anchor a Python idea to a real pharmaceutical situation. Read them carefully, they are not decoration. They are where the syllabus comes alive.

> **QUICK CHECK**
> A short conceptual question. Cover the next paragraph with a sheet of paper, think for a moment, and only then read on. The pause matters.

> **COMMON PITFALL**
> These are the mistakes I have watched students make, year after year. Reading them once saves several hours of confused debugging later.

FROM THE BENCH

Short asides connecting the chapter material to a real laboratory or clinical workflow, the kind of thing your seniors at the hospital pharmacy will remark on without explaining.

CODE LISTING

Code blocks have a coloured left bar and a grey background. The blue bar means it is code for you to type. A separate dotted box marked ‘output’ shows what Python prints back.

Table of Contents

What You'll Learn iii
Our Promise iii
Thank You iii
How to use this book vi
Reading from front to back (recommended for beginners) vi
Reading selectively (for revision or quick reference) vi
Icons used inside the chapters vi
Table of Contents viii
1.0 Learning outcomes for this unit 2
1.1 Why Python in pharmacy? 2
1.2 Installing Python and choosing an IDE 4
1.2.1 Installing Python 4
1.2.2 Why a text editor is not enough 4
1.2.3 Three IDEs you should know about 5
Jupyter Notebook 5
PyCharm 6
Visual Studio Code 6
1.3 Your first Python program 6
1.4 Variables and data types 7
1.4.1 Naming rules and good style 7
1.4.2 The four primitive data types 8
1.5 Type casting 9
When does casting fail? 10
1.6 Operators 10
1.6.1 Arithmetic operators 10
1.6.2 Comparison operators 11
1.6.3 Logical operators 11
1.6.4 Operator precedence 12
1.7 Input and output 12
1.7.1 print() in detail 12
1.7.2 f-strings: formatted string literals 12
1.7.3 input(): reading from the keyboard 13
1.8 Basic string operations 13
1.8.1 Concatenation, repetition and length 14
1.8.2 Indexing and slicing 14

1.8.3 Useful string methods 14
1.8.4 Strings are immutable 15
1.9 Standard libraries and third-party libraries 15
1.9.1 The standard library 15
1.9.2 Third-party libraries and pip 17
1.9.3 A short note on virtual environments 17
1.10 Worked examples 18
Example 1.1: Tablets per day 18
Example 1.2: Body-mass index 18
Example 1.3: Cleaning a drug name 19
Example 1.4: Half-life calculator 19
1.12 Self-assessment: multiple-choice questions 20
1.13 Short-answer questions 23
1.14 Practical exercises 23
2.0 Learning outcomes for this unit 26
2.1 Conditional statements 26
2.1.1 The plain if statement 26
2.1.2 if-else: choosing one of two paths 27
2.1.3 if-elif-else: choosing one of many paths 28
2.1.4 Nested conditions 30
2.1.5 Truthy and falsy values 31
2.2 Loops: repeating an action 31
2.2.1 The for loop 31
range(): a built-in counter 32
Enumerating items with their index 33
2.2.2 The while loop 33
2.2.3 break and continue 34
2.3 Functions 35
2.3.1 Defining and calling a function 36
2.3.2 Default values and keyword arguments 37
2.3.3 Multiple return values 38
2.3.4 Local versus global names 38
2.4 Putting it together: pharmaceutical calculators 38
2.4.1 A weight-based dose calculator 38
2.4.2 A BMI calculator with WHO categories 40
2.4.3 Asian-specific BMI cut-offs 41
2.5 Worked examples 41
Example 2.1: Counting febrile patients 41

Example 2.2: Renal-dose adjustment (illustrative; drug-specific in practice) 42
Example 2.3: Generating a dose-response table 42
Example 2.4: Eligibility checker as a function 43
2.6 Chapter summary 43
2.7 Self-assessment: multiple-choice questions 44
2.8 Short-answer questions 46
2.9 Practical exercises 47
3.0 Learning outcomes for this unit 49
3.1 Lists 49
3.1.1 Indexing and slicing 49
3.1.2 Common list operations 50
3.1.3 Iterating over a list 51
3.1.4 List comprehensions 51
3.2 Tuples 51
3.3 Dictionaries 52
3.3.1 Useful dictionary operations 53
3.3.2 A list of dictionaries 54
3.4 String manipulation revisited 54
3.4.1 split() and join() 55
3.4.2 Searching with in and find() 55
3.4.3 startswith, endswith, isnumeric 55
3.5 Introduction to NumPy arrays 56
3.5.1 Creating arrays 57
3.5.2 Element-wise arithmetic 57
3.5.3 Indexing and slicing 57
3.5.4 Two-dimensional arrays 58
3.6 Reading and writing CSV files 58
3.6.1 Reading with the standard csv module 58
3.6.2 Writing a CSV 59
3.6.3 Reading a CSV directly into a NumPy array 60
3.7 Working with healthcare datasets 60
3.7.1 Where to find practice datasets 60
3.7.2 A sample workflow 61
3.7.3 Putting steps 1 and 2 together 61
3.8 Worked examples 62
Example 3.1: Average dose per drug 62
Example 3.2: Filtering with a list comprehension 62
Example 3.3: NumPy summary 63

3.9 Chapter summary 63
3.10 Self-assessment: multiple-choice questions 63
3.11 Short-answer questions 66
3.12 Practical exercises 66
4.0 Learning outcomes for this unit 69
4.1 Why Pandas? 69
4.2 Series and DataFrame 69
4.2.1 Series: a single column with a name 70
4.2.2 DataFrame: many columns sharing one index 70
Index, columns and values 71
4.3 Reading CSV and Excel files 71
4.3.1 Writing back out 72
4.4 Inspecting a new dataset 73
4.5 Cleaning and missing values 74
4.5.1 Detecting missing values 75
4.5.2 Strategies for missing values 75
Drop the rows or columns 75
Fill (impute) the missing values 75
Leave them in and let aggregations skip them 76
4.5.3 Cleaning text columns 76
4.6 Filtering and selecting data 76
4.6.1 Selecting columns 76
4.6.2 Selecting rows by position or label 76
4.6.3 Filtering with boolean masks 77
4.6.4 Adding and replacing columns 77
4.6.5 Sorting 78
4.7 Group-by and aggregation 78
4.7.1 Multiple grouping columns 79
4.7.2 Custom aggregations 79
4.7.3 Pivot tables 79
4.8 Worked examples 80
Example 4.1: Quarterly ADR summary 80
Example 4.2: Cleaning a messy weight column 80
Example 4.3: Top-five prescribed antibiotics 80
Example 4.4: Joining two tables 81
4.9 Chapter summary 81
4.10 Self-assessment: multiple-choice questions 82
4.11 Short-answer questions 84

4.12 Practical exercises 85
5.0 Learning outcomes for this unit 87
5.1 The Matplotlib mental model 87
5.2 Line plots: concentration-time curves 88
5.2.1 Your first PK plot 88
5.2.2 Comparing two routes of administration 88
5.2.3 Useful keyword arguments to plot() 89
5.2.4 Log-scale axes 90
5.3 Histograms and box plots 90
5.3.1 Histograms 90
5.3.2 Box plots 91
5.4 Scatter plots: dose, AUC, and other relationships 92
5.5 Bar plots: ADR reporting rates 93
5.6 Dissolution profiles 94
5.7 Subplots: many panels at once 95
5.8 Reading a plot scientifically 96
5.8.1 Always look at the axes first 96
5.8.2 Translate the geometry into chemistry 96
5.8.3 Beware of small samples 96
5.9 Worked examples 97
Example 5.1: A Cmax annotation 97
Example 5.2: A two-panel ADR figure 97
Example 5.3: Plotting a DataFrame directly 97
5.10 Chapter summary 98
5.11 Self-assessment: multiple-choice questions 98
5.12 Short-answer questions 101
5.13 Practical exercises 101
Sample sessional examination paper 103
Section A · Multiple-choice questions (5 × 1 mark) 103
Section B · Short-answer questions (3 × 2 marks) 104
Section C · Long-answer questions (3 × 3 marks) 104
Sample end-semester examination paper 105
Section A · Multiple-choice questions (10 × 1 mark) 105
Section B · Short-answer questions (4 × 2 marks) 106
Section C · Long-answer questions (3 × 4 marks) 106
Glossary 108
References and further reading 111
Books 111

Online tutorials 111
Healthcare datasets for practice 111
Pharmacopoeias and clinical references 111

UNIT 1

Introduction to Python Programming

1.0 Learning outcomes for this unit

By the end of this unit you should be able to:

1. **Install Python** on your laptop and pick an Integrated Development Environment (IDE) you actually enjoy working in.
2. **Tell apart** the four primitive Python data types (integers, floats, strings and booleans) and convert between them sensibly.
3. **Write and read short Python programs** that use arithmetic, comparison and logical operators to compute simple pharmaceutical quantities.
4. **Use print() and input()** to talk to whoever is sitting at the keyboard.
5. **Tidy up strings** (drug names, batch numbers, OPD identifiers) using Python's built-in string methods.
6. **Import standard and third-party libraries** such as math and NumPy, and add or remove libraries with pip.

1.1 Why Python in pharmacy?

Open any pharmacy journal published in the last five years. The Indian Journal of Pharmaceutical Sciences. AAPS PharmSciTech. The European Journal of Pharmaceutical Sciences. The Journal of Pharmaceutical Analysis. Look at the methods sections. Python turns up everywhere, fitting dissolution profiles, mining ADR registers, simulating clinical trials, estimating pharmacokinetic parameters from sparse plasma sampling. A field that, even ten years ago, you could navigate with Excel alone has quietly turned digital, and Python has become its lingua franca.

Why Python and not, say, MATLAB or R? Three reasons, each worth pausing over. First, Python is free. You download it in twenty minutes, install it on your own laptop, and pay nothing. Python is unusually readable. A program in Python looks much like a careful set of instructions written in English. There are no curly braces to keep track of, no semicolons at the end of every line, and the indentation that organises the code reads almost like a paragraph. Third, the people who solved the everyday problems of scientific computing (fast numerical work, plotting, machine learning, statistics) chose Python first. NumPy, Pandas, Matplotlib, scikit-learn, RDKit, PyTorch: all of these were written for Python before any other language. Whichever direction you grow in later, the toolkit is already waiting for you.

The figure below is not a definitive map. It is a hint at how broad the territory is. Read it as an invitation rather than as a checklist.

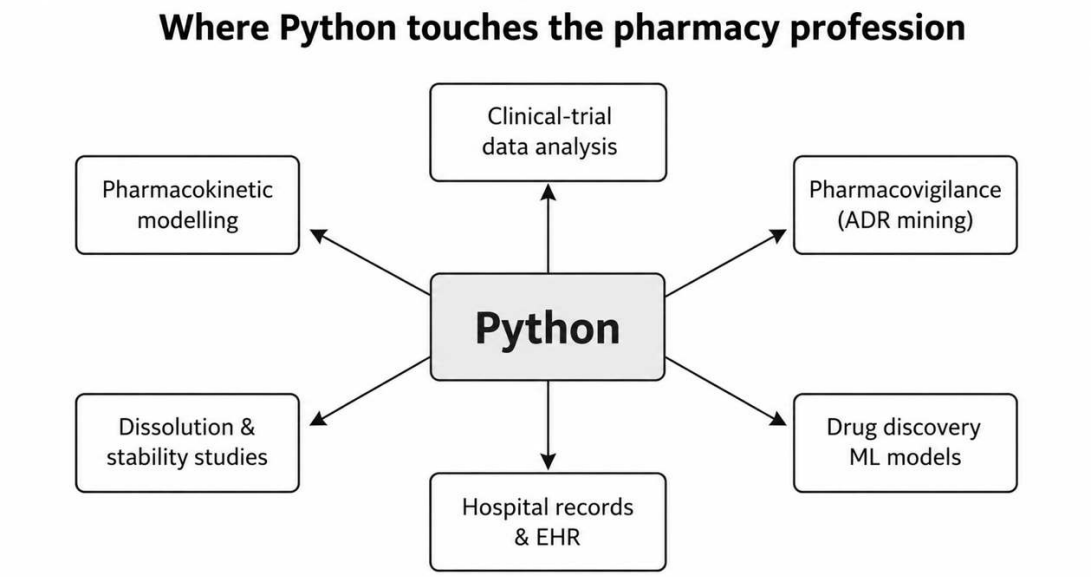

Figure 1.1 Python is used across the entire pharmaceutical pipeline, from molecule design to post-marketing surveillance.

PHARMA EXAMPLE

A small story from the pandemic. When the second wave of COVID-19 hit India in 2021, pharmacovigilance cells inside many tertiary-care hospitals were drowning in spreadsheets of suspected adverse drug reactions, each arriving from a different ward in a different format. Some had a column called 'Drug'; others called the same column 'Suspect medicine'. Some used DD/MM/YYYY for dates; others used MM/DD/YYYY. Reformatting each file by hand would have taken weeks. A junior resident I knew of solved the whole thing with a forty-line Python script, written across one weekend. That script merged the files, normalised the column names, flagged duplicates, and produced a weekly summary chart, and the skills she used were exactly the ones in this book.

If you are sitting there worrying that programming is something only 'computer-people' do, take a breath. Programming is, in the end, a precise way of writing down what you would anyway explain on a whiteboard. The first time will feel awkward (like the first time you handled a pipette without spilling) but the awkwardness fades quickly. Within three weekends, most students start writing small Python programs without thinking about it.

QUICK CHECK

Before moving on: list three pharmaceutical situations where you would want a computer to look at data instead of doing it yourself. Note them down. Only then read on.

1.2 Installing Python and choosing an IDE

Two pieces of software need to live on your laptop:

1. **The Python interpreter itself.** This is the program that actually reads and runs your code. You install it once. After that it sits quietly in the background until you call on it.
2. **An Integrated Development Environment (IDE).** This is the friendly editor in which you will type your code. Strictly optional (Python will run a file written in Notepad) but a real IDE makes a real difference.

1.2.1 Installing Python

Go to **python.org/downloads** and click the big yellow button offering the latest stable release. Anything Python 3.10 or newer is fine for everything in this book; the language is now in a settled, backward-compatible phase, so do not worry too much about the exact minor version.

On Windows: run the installer, and tick the box that says 'Add Python to PATH' on the very first screen. If you miss this box, the rest of the installation will complete just fine, but Python will not be found from the command line and you will spend a confused afternoon wondering why nothing works. I have seen it happen too many times.

On macOS: the installer is a regular .pkg file. Double-click and follow the prompts. On Ubuntu and most modern Linux distributions: Python is already installed; you can skip ahead.

To check that the installation worked, open a terminal (Command Prompt on Windows, Terminal on macOS, or your favourite shell on Linux) and type:

```
python --version
```

If something like 'Python 3.11.4' or 'Python 3.13.1' comes back, you are done, any 3.10-or-later version printed here is fine. If you see 'command not found', kindly go back and re-tick the PATH box.

1.2.2 Why a text editor is not enough

You can, in principle, write Python in Notepad and run it from the terminal. Generations of programmers did exactly that. It is not pleasant. Notepad cannot tell you that you have misspelt a function name. It cannot warn you when you forget a colon. It cannot show you the value of a variable while the program is paused mid-execution. It cannot autocomplete the long method names of the Pandas library.

An IDE (Integrated Development Environment) wraps a text editor together with a Python interpreter, a debugger, an interactive console, and a small army of helpful little features. The difference is much like the difference between a glass beaker and a graduated burette: both will hold liquid, but only one helps you see what you are actually doing.

TIP

Cannot decide which IDE to install? Start with Jupyter Notebook. Most pharmacy data work is exploratory in nature, and Jupyter is the gentlest place to learn. PyCharm and VS Code can wait. You can graduate to them later, when your projects outgrow a single notebook.

1.2.3 Three IDEs you should know about

The PCI syllabus singles out three IDEs by name: Jupyter Notebook, PyCharm and Visual Studio Code. Each has its own personality.

A quick comparison of the three IDEs in the syllabus

IDE	Style	For data work	Resource use	Best for
Jupyter Notebook	Browser-based, cell-by-cell	Excellent	Light	Beginner
VS Code	General editor, many languages	Good	Medium	Beginner+
PyCharm	Full IDE, debugger, refactor	Limited	Heavy	Intermediate

Figure 1.2 A quick side-by-side comparison of the three IDEs in the syllabus.

Jupyter Notebook

Jupyter is the favourite of data-science folk everywhere, and is therefore the natural home for pharmacy students who care about data. It runs in your web browser. Your work is broken into ‘cells’, each cell being a small piece of code. Run a cell, the result appears immediately below it, a number, a table, a plot, whatever. The whole experience feels much more like a lab notebook than like writing software.

The simplest way to get Jupyter is the **Anaconda** distribution from anaconda.com. Anaconda installs Python, Jupyter, NumPy, Pandas, Matplotlib and almost everything else you will need in one go. To start it, open the Anaconda Navigator and click ‘Launch’ next to Jupyter Notebook. A new browser tab opens at the address http://localhost:8888.

NOTE ON ANACONDA'S LICENCE

Anaconda's commercial terms have tightened in recent years. Organisations with more than 200 employees now need a paid licence to use the default channel. If you are installing it for personal study or for a small college lab, this does not affect you. If you are installing it inside a hospital, a

pharmaceutical company or a large institution, please check the licence first. The community-maintained Miniforge distribution (github.com/conda-forge/miniforge) is a free alternative that behaves identically and defaults to the conda-forge channel.

PyCharm

PyCharm is a heavyweight IDE made by JetBrains. PyCharm shines once your project grows beyond a single file: it understands which functions live where, lets you jump between them with one click, and has a debugger that is genuinely a pleasure to use. The trade-off is that PyCharm is large (about 1.2 GB on disk) and slower to start up. If your laptop is more than five years old, consider VS Code instead.

Visual Studio Code

VS Code is made by Microsoft. It is much lighter than PyCharm and, with a couple of extensions, almost as capable. To turn VS Code into a Python IDE: install it from code.visualstudio.com, open it up, click the four-square 'Extensions' icon on the left side, and install the 'Python' extension by Microsoft. From that moment on, VS Code understands Python. Many of my own colleagues have settled on VS Code in recent years.

COMMON PITFALL

Some students end up with multiple Python installations on the same laptop (one from python.org and another from Anaconda) and then become confused about which one their IDE is actually using. If a library you 'installed' refuses to import, this is almost always the reason. The cleanest cure is to commit to one Python installation and use only it. Pick Anaconda or pick plain Python; do not mix without good reason.

1.3 Your first Python program

Tradition demands that the first program in any new language print the words 'Hello, world!' to the screen. We will obey tradition, but with a pharmaceutical twist that fits this book.

Listing 1.1 Open Jupyter Notebook (or whichever IDE you have settled on), create a fresh notebook, and type:

```
# A friendly first program
drug = "Paracetamol"
dose = 500   # milligrams
print("Dispensing", drug, "-", dose, "mg tablet.")
```

Run it the green 'play' button in Jupyter, or by pressing Shift+Enter inside the cell. The interpreter will print:

Output
Dispensing Paracetamol - 500 mg tablet.

Three small things in those four lines deserve a pause.

First, the line that begins with # is a **comment**. Python ignores comments completely. They exist only for the human reader. The same is true in any programming language; you should learn early on to leave a comment whenever the next line is doing something .

Second, the lines starting with ‘drug =’ and ‘dose =’ create variables, labelled boxes that hold values. We will spend more time on variables in §1.4.

Third, print() is a function: a ready-made block of behaviour that takes some inputs (here, four pieces of information separated by commas) and produces a side-effect (it writes to the screen). Notice that print drops a single space between each of its inputs by default. We will see in §1.7 how to change that.

FROM THE BENCH
Programs that look this small are deceptive. The same three-line pattern (a couple of variables, then a print) sits at the heart of the quarterly reports that pharmacovigilance cells send to the CDSCO. The variables grow, the formatting becomes more elaborate, but the shape stays exactly the same. You are, in a real sense, already writing the same shape of code as a senior data analyst.

1.4 Variables and data types

A variable is, simply, a name you give to a value so that your program can refer to it later. In Python you create a variable by writing its name, an equals sign, and the value you want to store. There is no need to declare the variable's type ahead of time, the way you would in C or Java. Python figures the type out from the value itself.

```
patient_id = "P-1042"
age = 56              # years
weight = 64.5         # kilograms
is_diabetic = True
```

Once those four lines have run, four labelled boxes exist in Python's memory. The label patient_id refers to the text ‘P-1042’; the label age refers to the whole number 56; weight refers to the decimal 64.5; and is_diabetic refers to the truth value True. That is all there is to it.

1.4.1 Naming rules and good style

Python is fairly relaxed about variable names, but it does have rules:

- A name must start with a letter or underscore, never with a digit. (weight2 is fine; 2weight is not.)

- After the first character, you may use letters, digits and underscores. No spaces, no hyphens, no fancy punctuation.
- Names are case-sensitive. dose, Dose and DOSE are three different variables.
- You may not use a Python reserved word (for, while, if, return, class, …) as a variable name.

Beyond rules, there is style. Python programmers strongly prefer *snake_case* (lowercase letters with underscores between words) for variable names: body_weight_kg rather than BodyWeightKg. A clear name is worth ten lines of comments. Prefer blood_glucose_mg_dl over bg or x1. You will read this code yourself two months from now, and you will thank yourself.

1.4.2 The four primitive data types

Every value in Python belongs to a **type**. The four you will meet most often, especially early on, are int, float, str and bool.

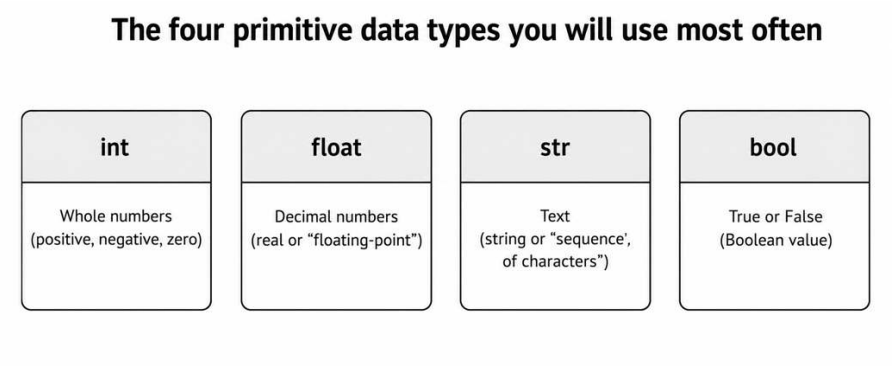

Figure 1.3 The four primitive data types. Every value belongs to exactly one type.

Type	What it is	Example	Typical pharmacy use
int	Whole numbers (positive, zero or negative) without a decimal point.	tablet_count = 30	Counting things; ages; days of therapy.
float	Numbers with a decimal point. Stored internally in IEEE 754 binary, so very small rounding errors can creep in.	dose = 2.5	Doses; concentrations; pH; AUC values.
str	A sequence of characters wrapped in quotation marks (single or double; both work equally well).	drug = "Paracetamol"	Drug names; patient IDs; report text.
bool	Truth values: True or False (with capital T and F).	is_pregnant = False	Eligibility flags; yes/no fields.

To find out the type of any value at any time, ask the built-in type() function:

```
print(type(patient_id))   # <class 'str'>
print(type(age))          # <class 'int'>
print(type(weight))       # <class 'float'>
print(type(is_diabetic))  # <class 'bool'>
```

This is something you will reach for surprisingly often when debugging.

> **COMMON PITFALL**
> If you write `dose = '2.5'` (note the quotation marks) Python stores the string "2.5", not the floating-point number 2.5. They look identical when printed, but they behave differently in arithmetic. When something is not adding up, the first thing to check is `type(your_variable)`. You will be surprised how often this is the answer.

1.5 Type casting

Type casting (sometimes called type conversion) means deliberately changing a value from one type to another. Python provides a built-in function for each of the primitive types:

Function	What it does
int(value)	Convert to an integer. Truncates towards zero: int(2.9) is 2, and int(-2.9) is -2 (also truncated towards zero, not downwards).
float(value)	Convert to a floating-point number. float("2.5") gives 2.5.
str(value)	Convert to a string. str(42) gives "42".
bool(value)	Convert to a truth value. The empty string and the number 0 are False; everything else is True.

Listing 1.2 A typical pharmacy use of casting: a weight read from a hospital record arrives as a string, but you need it as a float to compute a dose.

```
raw_weight = "64.5"          # value coming from a CSV file
weight_kg  = float(raw_weight) # convert to a number
dose_mg    = 10 * weight_kg    # 10 mg/kg, mg-per-kilogram dosing
print("Dose:", dose_mg, "mg")
```

Output

```
Dose: 645.0 mg
```

Two subtle things have happened. The multiplication `10 * weight_kg` produced a float because one of the operands was a float, Python promotes int to float automatically when they meet. And the `print()` statement implicitly converted the float to a string for display. Python casts on its own when it knows what you want, but you should still get into the habit of casting explicitly. It avoids surprises.

When does casting fail?

If you ask Python to do an impossible cast, it raises an error. `int("hello")` makes no sense (there is no integer hidden inside the word ‘hello’) and Python responds with a `ValueError`. Likewise `float("")` on an empty string fails. Always validate user input before casting it; otherwise a single stray space will crash your program.

1.6 Operators

Operators are the punctuation of arithmetic and logic. Python has three families of operators that you will use almost every day: arithmetic, comparison and logical.

1.6.1 Arithmetic operators

Symbol	Meaning	Example	Result
+	Addition	5 + 3	8
−	Subtraction	5 − 3	2
*	Multiplication	5 * 3	15
/	True division	5 / 2	2.5
//	Integer (floor) division	5 // 2	2
%	Modulus (remainder)	5 % 2	1
**	Exponentiation	5 ** 2	25

Two of these deserve special mention. The single-slash operator `/` always produces a float, even if both inputs are integers, so `4 / 2` is `2.0`, not `2`. The double-slash operator `//` keeps the result the same type as its inputs and rounds towards minus infinity. The designers of Python made this distinction long ago, and once you have written a few hundred lines of code you will appreciate why.

PHARMA EXAMPLE

Tablets per day. Suppose a patient needs 250 mg of cefuroxime twice a day, and the available tablet strength is 125 mg. The number of tablets per dose is `250 // 125 = 2` (integer division: you cannot give a fraction of a coated tablet). The total tablets for a seven-day course is `2 * 2 * 7 = 28`. This is the kind of calculation that hospital pharmacists do quietly, hundreds of times a day.

1.6.2 Comparison operators

Comparison operators ask a yes/no question and return a boolean.

Symbol	Meaning	Example
==	is equal to	dose == 500
!=	is not equal to	drug != "Aspirin"
<	is less than	ph < 7
>	is greater than	temperature > 38
<=	is less than or equal to	creatinine <= 1.2
>=	is greater than or equal to	age >= 18

COMMON PITFALL

A single equals sign = is assignment (it sets a variable). A double equals sign == is a comparison (it asks whether two things are equal). Mixing them up is the single most common beginner error in Python, partly because mathematicians never face this distinction. Read every = out loud as 'gets' and every == as 'is equal to'.

1.6.3 Logical operators

Logical operators combine multiple boolean values into one.

Operator	Meaning	Example
and	True only if both sides are True	(age >= 18) and (weight > 50)
or	True if either side is True	(temp > 38) or (heart_rate > 100)
not	Flips True to False and vice versa	not is_pregnant

Listing 1.3 An eligibility check for a study that requires the patient to be over 18 and not pregnant:

```
age = 32
is_pregnant = False
eligible = (age >= 18) and (not is_pregnant)
print("Eligible to enrol:", eligible)
```

Output

```
Eligible to enrol: True
```

Notice how the parentheses make the intent clear. Strictly speaking, Python does not require them, because comparison operators bind more tightly than logical operators. But parentheses cost nothing, and the code reads more like an English sentence with them. Use them.

1.6.4 Operator precedence

When Python sees a long expression like `2 + 3 * 4`, it follows the same precedence rules you learnt in school: multiplication and division before addition and subtraction (the BODMAS that we all grew up with). Exponentiation has even higher precedence than multiplication. If you are ever uncertain, place parentheses around the bit you want evaluated first. Parentheses make the order explicit and never cost anything.

1.7 Input and output

Programs that only print fixed text get tiresome very quickly. The two functions `input()` and `print()` let your program have a real conversation with whoever is sitting at the keyboard.

1.7.1 print() in detail

You have already met `print()`. It accepts any number of values, separated by commas, writes them to the screen with a single space between them and a newline at the end. Two keyword arguments give you fine control:

- `sep=` sets the separator between items. Default: a single space.
- `end=` sets what is printed at the end of the line. Default: the newline character.

```
print("Drug:", "Paracetamol", sep=" ", end="\n")  # default style
print("P-1042", "P-1043", "P-1044", sep=", ")       # comma-separated
print("Saving...", end="")                          # no newline
print("done.")
```

Output

```
Drug: Paracetamol
P-1042, P-1043, P-1044
Saving...done.
```

1.7.2 f-strings: formatted string literals

Modern Python (since version 3.6) introduces **f-strings**, which are by far the cleanest way to weave variable values into a piece of text. Place an ‘f’ in front of the opening quotation mark, and any expression placed inside curly braces is evaluated and substituted in.

```
drug = "Atorvastatin"
```

```
dose_mg = 20
auc = 612.78
print(f"{drug} {dose_mg} mg gave an AUC of {auc:.1f} ng·h/mL.")
```

Output

```
Atorvastatin 20 mg gave an AUC of 612.8 ng·h/mL.
```

The bit :.1f is a format specifier; it tells Python to print the value as a fixed-point number rounded to one decimal place. There is a long list of such codes, :.2f for two decimals, :,.0f for thousands separators, :>10 for right-justified in a width of 10, and so on. The four or five you actually need will become familiar within a fortnight. The rest you will look up when you need them.

1.7.3 input(): reading from the keyboard

The input() function pauses the program and waits for the user to type something and press Enter. Whatever the user types is returned as a string. Even if the user types '64.5', the input function gives you the four-character string '64.5', not the float 64.5. Cast it before doing arithmetic on it.

Listing 1.4 A tiny dose calculator the user can run from the keyboard:

```
name = input("Patient name: ")
weight_kg = float(input("Weight (kg): "))
mg_per_kg = float(input("mg per kg dose: "))
dose = weight_kg * mg_per_kg
print(f"For {name}: dose = {dose:.0f} mg")
```

Output

```
Patient name: Mrs Iyer
Weight (kg): 58
mg per kg dose: 12.5
For Mrs Iyer: dose = 725 mg
```

1.8 Basic string operations

Strings are how Python represents text, and pharmacy is full of text, drug names, batch numbers, OPD identifiers, ward names, free-text notes from the consulting physician. Python gives you a generous toolkit for cleaning and reshaping such text.

1.8.1 Concatenation, repetition and length

```
first = "Atorva"
second = "statin"
drug = first + second        # concatenation
banner = "-" * 30           # repetition
print(banner)
print(drug, "length =", len(drug))
print(banner)
```

Output

```
------------------------------
Atorvastatin length = 12
------------------------------
```

The + operator joins two strings end to end. The * operator repeats a string a given number of times, handy for separator lines in printed reports. The built-in `len()` function returns the number of characters in a string (or, more generally, the number of items in any sequence).

1.8.2 Indexing and slicing

Each character of a string sits at a position called an *index*. Python indexes from zero, not one. Negative indices count from the end.

```
drug = "Paracetamol"
print(drug[0])      # 'P'
print(drug[-1])     # 'l'
print(drug[0:4])    # 'Para'  -- a slice
print(drug[5:])     # 'etamol' -- from index 5 to the end
```

A slice `drug[a:b]` returns the characters from index `a` up to but **not including** index `b`. The half-open convention may feel arbitrary at first; with practice it becomes second nature.

1.8.3 Useful string methods

A method is a function attached to a value. Call it by writing the value, a dot, and the method name. Strings come with a long list of methods; here are the ones you will use :

Method	What it does	Example	Result
.upper()	Uppercase copy	"para".upper()	PARA
.lower()	Lowercase copy	"PARA".lower()	para
.strip()	Removes leading/trailing spaces	" para ".strip()	para
.replace(a,b)	Replaces all 'a' with 'b'	"para".replace('a','o')	poro
.split(sep)	Splits on separator → list	"a,b,c".split(',')	['a','b','c']
.find(sub)	Index of substring (or −1)	"para".find('ra')	2
.startswith(s)	Boolean test of prefix	"P-1042".startswith('P')	True

PHARMA EXAMPLE

Cleaning a clinical-trial CSV. Suppose a junior researcher receives a spreadsheet of patient identifiers from three centres: 'P-1042', ' p1042 ', 'P 1042'. To make the IDs uniform, she trimmed whitespace, upper-cased the letters, and removed embedded spaces:

```
raw = " p 1042 "
clean = raw.strip().upper().replace(" ", "")
print(clean)   # 'P1042'
```

1.8.4 Strings are immutable

An important detail: strings cannot be changed in place. The expression drug.upper() does not modify drug it returns a new string. If you want to keep the upper-cased version, assign it back: drug = drug.upper().

1.9 Standard libraries and third-party libraries

Most useful Python code is not written by you. It is written, freely shared, and continuously improved by other people, packaged into *libraries* (or *modules* or, when grouped together, *packages*). The art of a working pharmacy programmer is, in part, the art of finding the right library for the job before reinventing the wheel.

1.9.1 The standard library

Every installation of Python comes with a generous standard library. You do not have to install it. You only have to import the parts you need. Some of the most useful modules for pharmacy work:

Module	What it offers
math	Mathematical functions: sqrt, log, exp, sin, pi, e.
statistics	Mean, median, mode, standard deviation, variance.
random	Random numbers: useful when simulating clinical trials or sampling distributions.
datetime	Dates and times: useful for trial timelines, stability schedules, expiry tracking.
csv	Reading and writing CSV files. (We will see Pandas in Unit 4, but csv is the lighter, no-install option.)
os, pathlib	Working with files and folders on disk.

Listing 1.5 Using the math module to compute a half-life.

```
import math

k_e = 0.231                    # elimination rate constant per hour
t_half = math.log(2) / k_e     # natural log
print(f"Half-life = {t_half:.2f} h")
```

Output

```
Half-life = 3.00 h
```

Three styles of import are common, and you should recognise all three:

```
import math              # use the whole module:  math.sqrt(2)
from math import log, exp  # bring in just the names you need
import math as m         # short alias:  m.sqrt(2)
```

TIP

Conventional aliases that you will see in almost every notebook on the planet: ‘import numpy as np’, ‘import pandas as pd’, ‘import matplotlib.pyplot as plt’. Stick to these conventions. It will make your code intelligible to anyone else who comes across it.

1.9.2 Third-party libraries and pip

Anything beyond the standard library has to be installed separately. The standard installer is pip, which comes with Python. To install a library, open a terminal and type:

```
pip install numpy
pip install pandas matplotlib
```

(On some systems you may need to type ‘pip3’ instead of ‘pip’ to make sure you are installing for Python 3 and not Python 2, the latter is now retired but still lingers on a few older laptops.)

To uninstall a library that you no longer need:

```
pip uninstall numpy
```

To list what is currently installed:

```
pip list
```

If you went the Anaconda route, the equivalent command is `conda install`. Conda manages whole environments (folders that hold a copy of Python and a particular set of libraries) and is therefore especially useful when you want different projects to use different versions of the same library. For your first months you will not need that level of complexity. Keep things simple.

COMMON PITFALL

Beginners often install a library (Pandas, say) and find that their script still complains ‘ModuleNotFoundError: No module named pandas’. The cause is almost always that pip installed the library into a different Python from the one their IDE is using. To check, run `import sys; print(sys.executable)` inside Jupyter and compare the path to the one used by your terminal pip. The mismatch tells you everything.

1.9.3 A short note on virtual environments

As your projects grow you will start to want different libraries, sometimes different *versions* of the same library: for different projects. The standard solution is a *virtual environment*: a self-contained folder that holds a private copy of Python and its libraries, isolated from the rest of your system. The four lines below create a venv called `.venv`, activate it, and install Pandas inside it:

```
python -m venv .venv             # create the env
source .venv/bin/activate        # macOS / Linux
.venv\Scripts\activate           # Windows (in PowerShell)
pip install pandas matplotlib    # libraries go inside the env, not your global Python
```

After deactivate, you are back to your system Python. The venv folder can be deleted at any time without affecting your global installation. If you went the Anaconda route, the equivalent is conda create -n bp101 python=3.11 followed by conda activate bp101. Either way, the habit is worth picking up early, it solves many of the problems described in the pitfall above.

On uninstalling. pip uninstall pandas removes Pandas but does not remove the libraries that came with it (NumPy, dateutil, six, and so on). Over many install/uninstall cycles your environment ends up cluttered. The cleanest reset is to delete the venv folder and start a new one, which is exactly why venvs are a good habit.

1.10 Worked examples

In the closing section of this chapter we put the pieces together. Three short examples. Try each one yourself before reading the solution.

Example 1.1: Tablets per day

A patient is prescribed a 250 mg dose of azithromycin once daily for 5 days. The available tablet strength is 500 mg. Write a program that prints the number of tablets the pharmacist should dispense.

```
dose_mg = 250         # prescribed dose per day
strength_mg = 500     # available tablet strength
days = 5

tablets_per_day = dose_mg / strength_mg     # 0.5 tablet -- half-tablet allowed
total_tablets = tablets_per_day * days      # 2.5 tablets total

print(f"Dispense {total_tablets} tablets of {strength_mg} mg.")
```

Output

```
Dispense 2.5 tablets of 500 mg.
```

The output of 2.5 is correct mathematically, but in practice you would either round up to whole tablets (each tablet being scored to allow halving) or hand the patient a different strength. We will revisit this in Unit 2 once we have learnt the if-statement.

Example 1.2: Body-mass index

Read the user's height (in metres) and weight (in kilograms) from the keyboard, compute the BMI, and print it to two decimal places.

```
h = float(input("Height (m): "))
w = float(input("Weight (kg): "))

bmi = w / h ** 2
print(f"BMI = {bmi:.2f}")
```

Output

```
Height (m): 1.62
Weight (kg): 58
BMI = 22.10
```

Notice how `h ** 2` is evaluated before the division because of operator precedence. If you are ever in doubt, write `(h ** 2)` parentheses always make the order explicit, and they cost nothing.

Example 1.3: Cleaning a drug name

A user types the name of a drug with extra spaces and inconsistent casing, for example, ‘ paraceTAMol ’. Print a clean, capitalised version: ‘Paracetamol’.

```
raw = input("Drug name: ")
clean = raw.strip().lower().capitalize()
print("Cleaned:", clean)
```

Output

```
Drug name:   paraceTAMol
Cleaned: Paracetamol
```

Example 1.4: Half-life calculator

Given an elimination-rate constant k_e (per hour), compute the half-life in hours and minutes.

A pharmacological caveat. The formula $t_{½} = \ln(2) / k_e$ holds only for *first-order* elimination: true for most drugs at therapeutic doses, but explicitly wrong for drugs that follow zero-order or saturable kinetics (phenytoin within its therapeutic range, alcohol, salicylates at high doses, paracetamol overdose). For now we will assume first-order behaviour; we revisit non-linear kinetics in Unit 5.

```
import math

k_e = float(input("k_e (per hour): "))
t_half_hours = math.log(2) / k_e
```

```
hours = int(t_half_hours)
minutes = int((t_half_hours - hours) * 60)

print(f"t½ = {t_half_hours:.2f} h  =  {hours} h {minutes} min")
```

Output

```
k_e (per hour): 0.116
t½ = 5.97 h  =  5 h 58 min
```

1.12 Self-assessment: multiple-choice questions

Each question has exactly one correct answer. Answers and brief explanations follow.

Q1. Which of the following is NOT a primitive Python data type?

(a) int
(b) float
(c) tablet
(d) bool

Q2. What is the value of `type(3 / 1)` ?

(a) <class 'int'>
(b) <class 'float'>
(c) <class 'str'>
(d) <class 'bool'>

Q3. Which symbol is used for the modulus (remainder) operator?

(a) /
(b) %
(c) **
(d) //

Q4. Which of the following names is NOT a legal Python variable name?

(a) dose_mg
(b) _age
(c) 2nd_dose
(d) patientID

Q5. Which function returns the type of any value?

(a) typeof()
(b) kind()
(c) type()
(d) datatype()

Q6. What does the expression `10 // 3` evaluate to?

(a) 3
(b) 3.33
(c) 3.0
(d) 1

Q7. Which of the following correctly converts the string "4.5" to a floating-point number?

(a) int("4.5")
(b) float("4.5")
(c) str("4.5")
(d) bool("4.5")

Q8. An f-string in Python begins with which character before the opening quote?

(a) s
(b) r
(c) f
(d) b

Q9. What is the output of print("Para" * 3) ?

(a) Para Para Para
(b) ParaParaPara
(c) TypeError
(d) Para3

Q10. Which logical operator returns True only if both sides are True?

(a) or
(b) not
(c) and
(d) xor

Q11. input() always returns a value of type:

(a) int
(b) float
(c) str
(d) bool

Q12. Which of the following will print Para?

(a) print('Paracetamol'[0:3])
(b) print('Paracetamol'[0:4])
(c) print('Paracetamol'[1:4])
(d) print('Paracetamol'[0:5])

Q13. Which command installs a third-party library?

(a) python install pandas
(b) pip install pandas
(c) import pandas
(d) load pandas

Q14. The expression bool(0) evaluates to:

(a) True
(b) False
(c) 0
(d) Error

Q15. Which symbol is used for exponentiation in Python?

(a) ^
(b) **
(c) //
(d) exp

ANSWERS AND EXPLANATIONS

1. (c) tablet 'tablet' is not a Python type.

2. (b) <class 'float'> a single slash always returns a float.

3. (b) % the percent sign is the modulus operator.

4. (c) 2nd_dose variable names cannot start with a digit.

5. (c) type() a built-in function.

6. (a) 3 floor division of integers gives an integer result.

7. (b) float("4.5") the float constructor handles a numeric string.

8. (c) f 'f' marks the literal as an f-string.

9. (b) ParaParaPara the * operator on a string repeats it without inserting spaces.

10. (c) and the 'and' operator requires both operands to be True.

11. (c) str input always returns a string. Cast it before doing arithmetic.

12. (b) print('Paracetamol'[0:4]) slicing is up to but not including the end index.

13. (b) pip install pandas pip is the standard installer.

14. (b) False zero is the only numeric value that is falsy.

15. (b) ** Python uses the double-asterisk for exponentiation.

1.13 Short-answer questions

1. Define an Integrated Development Environment, and explain why an IDE is preferable to a plain text editor.
2. List the four primitive data types in Python and give one pharmacy-flavoured example of each.
3. Distinguish between the / and // arithmetic operators with an example.
4. What is the difference between assignment (=) and equality testing (==)? Give an example of a bug that arises when these are confused.
5. Explain, with a code example, why we cast the value returned by input() before using it in arithmetic.
6. Briefly describe three string methods you would use while cleaning a list of patient identifiers, and explain what each one does.
7. State the difference between import math and from math import log.

1.14 Practical exercises

Complete each of these in Jupyter Notebook or your chosen IDE, and save the resulting .ipynb (or .py) file with your roll number as the filename.

LAB 1.1: ADVERSE-DRUG-REACTION CALCULATOR

Write a program that asks the user for the total number of prescriptions issued at a hospital in a month, and the number of those that resulted in a reported ADR. Compute and print the ADR rate per 1000 prescriptions, to two decimal places.

Hint: rate = (adr / total) * 1000.

LAB 1.2: HALF-LIFE VS ELIMINATION CONSTANT

Write a program that asks the user for the elimination rate constant k_e (per hour) and prints the half-life in hours and minutes. Use the math module.

LAB 1.3: DRUG-NAME CLEANER

Read a drug name from the user (which may have leading/trailing spaces and inconsistent case). Print the cleaned name in title case. Then print the number of letters in the name. Test your program with at least three different inputs.

LAB 1.4: ELIGIBILITY FLAG

A clinical trial requires that participants be aged 18 to 65 (inclusive), weigh more than 45 kg, and be non-pregnant. Read the three quantities from the user and print whether the participant is eligible to enrol.

Note: you may use a single boolean expression here; we will refine the program in Unit 2 using if-statements.

UNIT 2

Control Structures & Functions

2.0 Learning outcomes for this unit

After working through this unit you should be able to:

1. **Write conditional statements** (if, if-else, if-elif-else, and nested) so that a Python program takes different paths depending on the data it sees.
2. **Construct for and while loops** to repeat a block of code, and use break and continue to alter the flow of those loops.
3. **Define and call functions** with named parameters, optional default values, and a return value.
4. **Build small pharmaceutical calculators** such as a body-mass-index classifier, a paediatric dose calculator, and a study-eligibility checker.
5. **Read another person's program** and predict what it will print, by tracing the control flow on paper.

2.1 Conditional statements

Up to this point, every program we have written has been a single straight road. Each line executed in turn, top to bottom.

A pharmacist looking at a prescription has to ask questions: is the patient pregnant, is the renal function impaired, is the dose within the licensed range? And depending on the answers, very different things must happen. Conditional statements are how Python expresses exactly this kind of decision. Once you can write an if-statement, you can write a small clinical decision-support tool. The step from one to the other is shorter than you might think.

2.1.1 The plain if statement

The simplest form of conditional runs a block of code only when a condition is true. The keyword is if, followed by the condition, a colon, and an indented block:

```
temperature_c = 38.6

if temperature_c > 38:
    print("Patient is febrile.")
```

Output

```
Patient is febrile.
```

Three small details. The colon at the end of the if line is mandatory; the body is indented by four spaces; and only the indented block runs when the condition is true. Python uses indentation (not curly braces) to mark the body of an if-statement. This feels strange at first and very quickly feels right. Poorly indented Python is not even legal Python. The language quietly forces your code to look readable.

COMMON PITFALL

Tabs versus spaces. Indent your code with four spaces. Always. Never with tabs, and never mix the two. Some editors will quietly insert tab characters when you press the Tab key, and Python will refuse to run a file that mixes both. Set your IDE to ‘insert spaces for tab’ and you will avoid a lifetime of strange ‘inconsistent indentation’ errors.

2.1.2 if-else: choosing one of two paths

When you want one thing to happen if a condition is true and a different thing if it is false, pair if with else:

```
temperature_c = 36.9

if temperature_c > 38:
    print("Give antipyretic")
else:
    print("Continue monitoring")
```

Output

```
Continue monitoring
```

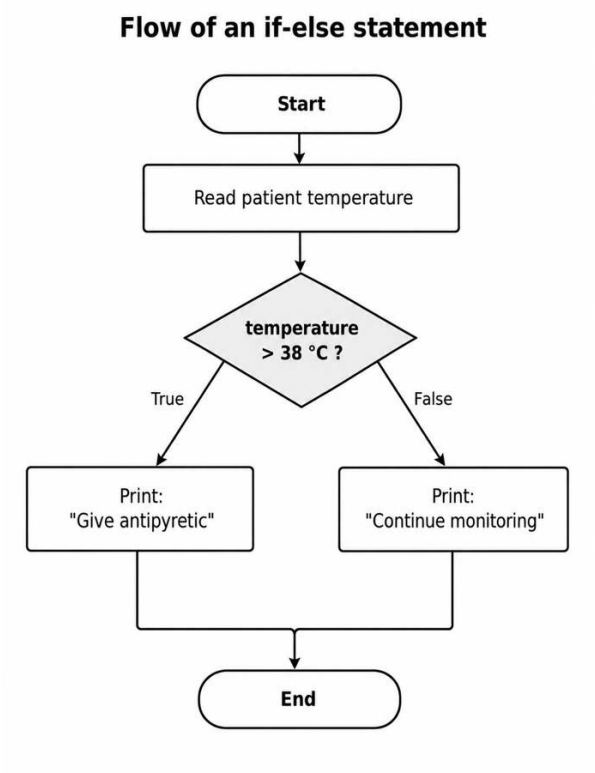

Figure 2.1 The two-branch flow of an if-else statement.

The else keyword sits at the same indentation as the if, with its own colon and its own indented block. Always exactly one of the two blocks runs. Never both. Never neither. That guarantee is what makes if-else easier to reason about than a sequence of unrelated ifs.

2.1.3 if-elif-else: choosing one of many paths

Sometimes a quantity has to be classified into more than two categories. Body-mass index, for instance, is conventionally read into four categories: under-weight, normal, over-weight and obese. The keyword elif (short for 'else if') chains conditions together. Python checks them top to bottom and runs the first one whose condition is true. The rest are skipped.

```
bmi = 27.4

if bmi < 18.5:
    category = "Under-weight"
elif bmi < 25:
    category = "Normal"
elif bmi < 30:
    category = "Over-weight"
else:
    category = "Obese"

print(f"BMI {bmi} is {category}.")
```

Output
BMI 27.4 is Over-weight.

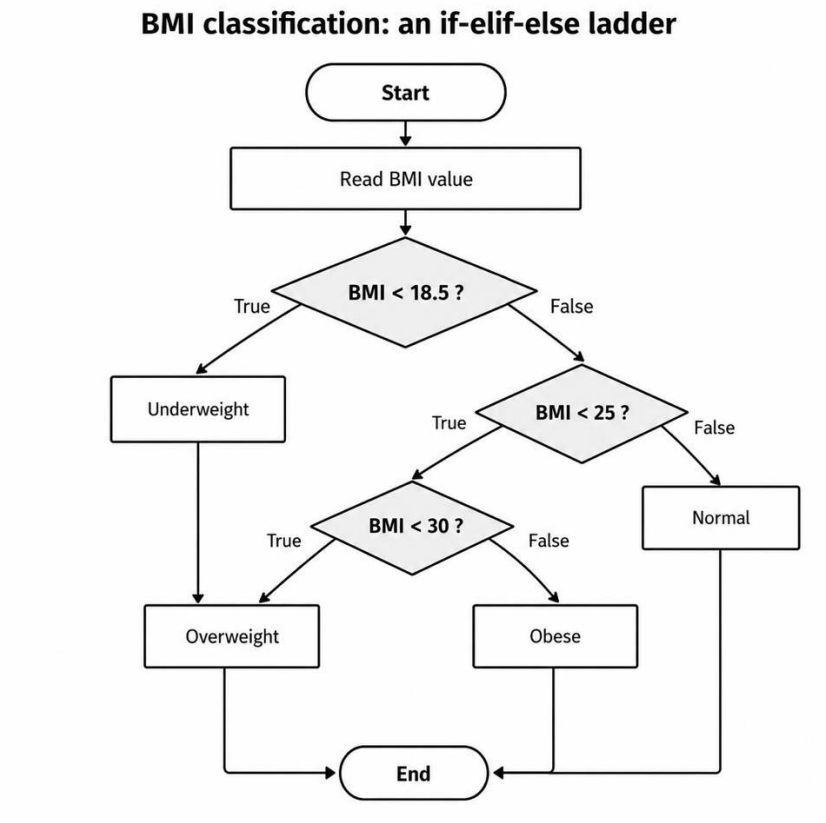

Figure 2.2 Decision flow for the four BMI categories using if-elif-else.

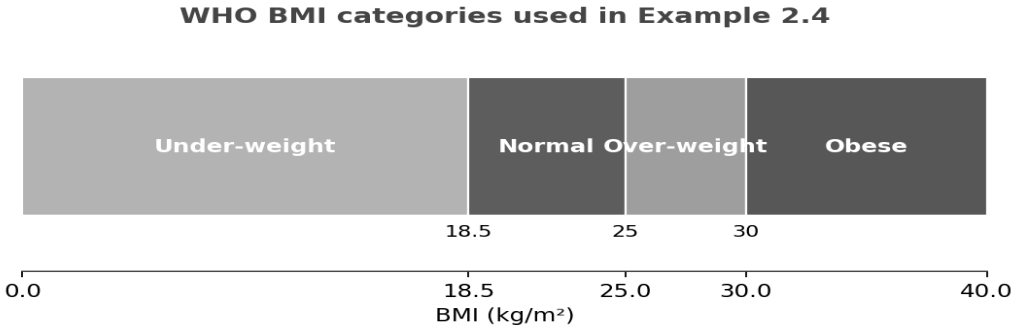

Figure 2.3 The numerical breakpoints. The Asian/South-Asian cut-offs differ slightly; we discuss them in §2.4.3.

Three small details are worth noting. First, the conditions are checked in order, so by the time Python reaches 'BMI < 25' it already knows that 'BMI ≥ 18.5' is true. You can rely on that. Second, the chain may have as many elif branches as you like. The trailing else is optional, but it is often a good idea, it forces you to think about what happens when none of the conditions match. Third: only one branch runs.

PHARMA EXAMPLE

Triage by NEWS-2 score. The National Early Warning Score (NEWS-2) is the standard early-warning score in the UK NHS and is being adopted by a growing number of Indian tertiary-care hospitals; many Indian institutions also use MEWS (Modified Early Warning Score) or local variants. Whichever score is in use, the structure is the same: a numerical aggregate is mapped to low, medium or high risk by a small set of cut-offs. For NEWS-2: total < 5 is low risk, 5–6 is medium, ≥ 7 is high. The decision rule is exactly an if-elif-else, and the same shape of code crops up everywhere clinical decision support exists.

2.1.4 Nested conditions

A conditional statement can itself contain another conditional statement. We say the inner one is *nested* inside the outer one. This lets you express compound decisions:

```
is_pregnant = True
trimester = 1

if is_pregnant:
    if trimester == 1:
        print("Avoid known teratogens; review prescription for risk.")
    else:
        print("Use higher-risk drugs only when benefit clearly outweighs risk.")
else:
    print("No pregnancy-related restrictions for this patient.")
```

A NOTE ON PREGNANCY CATEGORIES

Older textbooks (and many real prescriptions) refer to drugs as 'Category A/B/C/D/X', the FDA letter system. This system was officially retired in 2015 and replaced by the Pregnancy and Lactation Labeling Rule (PLLR), which uses narrative risk text rather than letter grades. Newer drug labels do not carry letter categories at all. We use risk-based phrasing in this book; if you encounter the old letters in your clinical placements, treat them as deprecated shorthand and read the actual product information.

Two levels of nesting is, in practice, the limit beyond which a program becomes hard to follow. Beyond that, prefer to combine conditions with logical operators (and, or, not) or extract the inner block into a function (we will get to functions in §2.3).

2.1.5 Truthy and falsy values

Any value placed in the slot of a condition is implicitly converted to a boolean. The values that count as False, the *falsy* values: are surprisingly few:

- The boolean False itself.
- The integer 0 and the float 0.0.
- The empty string "".
- The empty list [], the empty tuple (), and the empty dictionary {}.
- The special value None.

Everything else is truthy. This is why you can write:

```
name = input("Patient name: ")
if name:
    print("Hello,", name)
else:
    print("No name entered.")
```

and the empty-string case is caught by the falsy rule. The idiom (‘if a string is not empty’) turns up everywhere in real Python.

2.2 Loops: repeating an action

If conditional statements let your program take different paths, loops let your program walk the same path many times. Pharmacy is full of loop-shaped tasks: read a list of fifty patient weights and compute a dose for each one; iterate over twenty wells of an HPLC plate and average the peak areas; flip through the rows of an ADR spreadsheet and tally the reports per drug. Python offers two kinds of loops, the for loop and the while loop, with subtly different temperaments.

2.2.1 The for loop

A for loop walks through every item of a sequence (a list, a string, a range of numbers, the lines of a file) and runs its body once for each item.

```
weights = [55, 62, 70, 80, 95]   # patient weights in kg

for w in weights:
    dose = 10 * w              # 10 mg/kg dosing
    print(f"Weight {w} kg → dose {dose} mg")
```

Output

```
Weight 55 kg → dose 550 mg
Weight 62 kg → dose 620 mg
Weight 70 kg → dose 700 mg
Weight 80 kg → dose 800 mg
Weight 95 kg → dose 950 mg
```

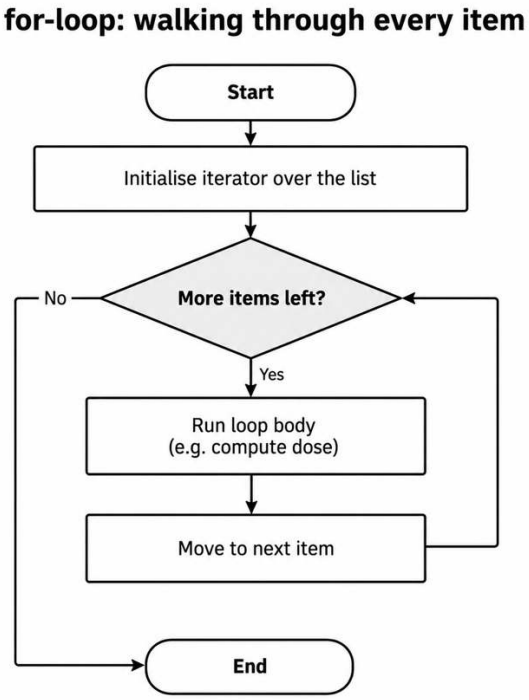

Figure 2.4 The for loop visits every item of a sequence in turn.

The variable w is the *loop variable*. On each iteration Python automatically assigns it the next item from the list. After the loop ends, w keeps the value of the very last item, but you should not rely on this. The cleanest code uses the loop variable only inside the loop.

range(): a built-in counter

Sometimes you want to repeat an action a fixed number of times rather than walk through a list. The range() function generates a sequence of integers cheaply. There are three forms:

Call	Yields	Notes
range(5)	0, 1, 2, 3, 4	stop is excluded
range(2, 8)	2, 3, 4, 5, 6, 7	start, stop
range(0, 100, 10)	0, 10, 20, …, 90	start, stop, step

Used inside a for loop:

```
for i in range(5):
```

```
    print("Tablet number", i + 1)
```

Output

```
Tablet number 1
Tablet number 2
Tablet number 3
Tablet number 4
Tablet number 5
```

Enumerating items with their index

When you need both the item and its position, enumerate() is the idiom of choice:

```
drugs = ["Paracetamol", "Ibuprofen", "Aspirin"]

for i, name in enumerate(drugs, start=1):
    print(f"{i}. {name}")
```

Output

```
1. Paracetamol
2. Ibuprofen
3. Aspirin
```

2.2.2 The while loop

A while loop repeats its body as long as a condition stays true. We do not have to know in advance how many iterations will be needed; we only have to know when to stop.

```
dose_mg = 100      # starting dose
step = 0

while dose_mg > 12.5:
    dose_mg = dose_mg / 2
    step += 1
    print(f"Step {step}: dose now {dose_mg} mg")

print("Tapering complete.")
```

Output

```
Step 1: dose now 50.0 mg
Step 2: dose now 25.0 mg
Step 3: dose now 12.5 mg
Tapering complete.
```

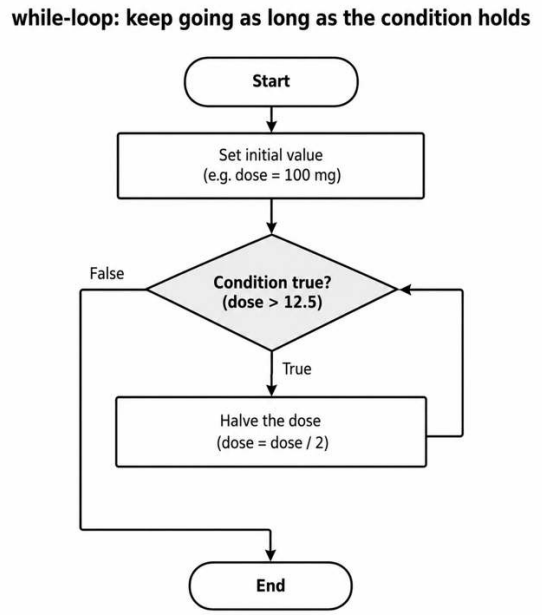

Figure 2.5 The while loop checks its condition before every iteration.

COMMON PITFALL

Infinite loops. If the condition of a `while` loop never becomes false, the loop runs forever and the program appears to hang. Always make sure the body of the loop changes some variable that is part of the condition. If you ever find yourself stuck in a runaway loop in Jupyter, click the 'stop' button (the black square on the toolbar). In a terminal, press Ctrl-C.

2.2.3 break and continue

Two small keywords give you fine control over a loop's flow. `break` jumps out of the loop entirely. `continue` skips the rest of the current iteration and starts the next one. Both feel a little bit magical the first time you use them, and they remain among the most reached-for tools in any working Python programmer's vocabulary.

Listing 2.1 Stop scanning a list as soon as we find a patient with an unsafe creatinine.

```
creatinines = [0.9, 1.0, 1.1, 2.4, 1.0, 1.2]   # mg/dL, sample of patients

for i, c in enumerate(creatinines, start=1):
    if c > 2.0:
```

```
        print(f"Stopping audit at patient {i}, creatinine = {c}")
        break
    print(f"Patient {i} OK")
```

Output

```
Patient 1 OK
Patient 2 OK
Patient 3 OK
Stopping audit at patient 4, creatinine = 2.4
```

Listing 2.2 Skip the empty (None) rows in a list of patient ages.

```
ages = [54, None, 32, None, 47, 61]   # None means missing
total = 0
valid = 0

for a in ages:
    if a is None:
        continue          # skip the missing rows
    total += a
    valid += 1

print("Mean age =", total / valid)
```

Output

```
Mean age = 48.5
```

A small detail. Notice we wrote `if a is None:` rather than `if a == None:`. The two operators are not synonyms. `==` checks whether two values are equal; `is` checks whether two names refer to the very same object in memory. For singletons such as `None`, `True` and `False`, PEP 8 (the official Python style guide) recommends `is`.

2.3 Functions

Functions are reusable bundles of code with a name. Once you have defined a function, you can call it as many times as you like, with different inputs each time, and treat the whole thing as a single operation. Functions are the most important tool we have for keeping a program understandable as it grows from a five-line script to a five-thousand-line application.

2.3.1 Defining and calling a function

A function is defined with the keyword def, followed by the function name, the list of parameters in parentheses, a colon, and an indented body. The body usually ends with a return statement that says what the function produces back to its caller.

```
def calculate_dose(weight_kg, mg_per_kg):
    """Return the dose in milligrams using mg-per-kilogram dosing."""
    dose = weight_kg * mg_per_kg
    return dose

# Calling the function:
d1 = calculate_dose(60, 10)
d2 = calculate_dose(75, 12.5)

print(d1, d2)
```

Output

```
600 937.5
```

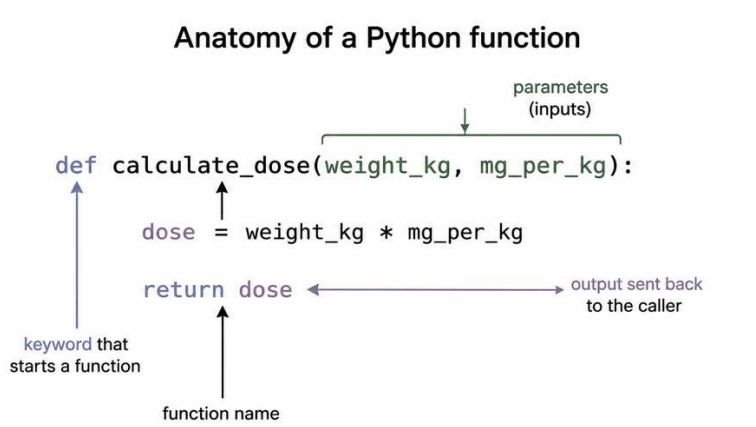

Figure 2.6 Anatomy of a Python function definition.

Three small things deserve attention. The triple-quoted string at the top is a *docstring* optional, but strongly encouraged because it documents what the function does and is shown by Python's built-in help() function. The parameters weight_kg and mg_per_kg are local variables that exist only inside the function. And the value returned by return is the function's output: the calling code captures it with an assignment.

2.3.2 Default values and keyword arguments

Function parameters can carry *default values*. When a default is provided, the caller may omit that argument entirely; the function uses the default if the caller did not specify one.

```
def calculate_dose(weight_kg, mg_per_kg=10):
    return weight_kg * mg_per_kg

calculate_dose(60)           # uses default → 600
calculate_dose(60, 12.5)     # overrides    → 750
calculate_dose(weight_kg=60, mg_per_kg=12.5)   # named arguments, same answer
```

Calling a function with named arguments, also known as *keyword arguments* is a habit worth picking up early. calculate_dose(weight_kg=60, mg_per_kg=12.5) reads almost like English; calculate_dose(60, 12.5) forces the reader to remember which is which.

COMMON PITFALL: MUTABLE DEFAULTS

This one catches every Python beginner eventually. Default values are evaluated once, at the moment the function is defined, not each time the function is called. So if the default is a mutable object (a list, a dictionary, a set), all calls share it:

```
def add_drug(name, drugs=[]):     # do NOT do this
    drugs.append(name)
    return drugs

add_drug("Crocin")    # ['Crocin']
add_drug("Combiflam") # ['Crocin', 'Combiflam']  -- the list is shared!
```

Correct pattern: default to None and create the empty container inside the function body.

```
def add_drug(name, drugs=None):
    if drugs is None:
        drugs = []
    drugs.append(name)
    return drugs
```

2.3.3 Multiple return values

A function may return more than one value. Strictly speaking, Python returns a single tuple, but the syntax lets you ‘unpack’ it instantly:

```
import math

def half_life(k_e):
    """Return half-life in (hours, minutes) given the elimination constant."""
    t = math.log(2) / k_e
    h = int(t)
    m = round((t - h) * 60)
    return h, m             # returns the tuple (h, m)

hours, minutes = half_life(0.231)
print(f"t½ = {hours} h {minutes} min")
```

Output

```
t½ = 3 h 0 min
```

2.3.4 Local versus global names

A variable defined inside a function is *local* to that function. It does not exist outside. A variable defined at the top level of a script is *global* and can be read (but not modified) from inside a function. Strong convention in Python: do not write to global variables from inside a function. Pass values in as parameters and send results out as return values. Functions whose only effect is on their inputs and outputs are far easier to test and reason about than functions that quietly mutate global state.

2.4 Putting it together: pharmaceutical calculators

In this section we build two complete programs that the syllabus singles out as applications: a dosage calculator and a body-mass-index classifier. Each combines input/output, conditionals, loops and functions.

2.4.1 A weight-based dose calculator

Specification: ask the user for a list of patient weights, the milligram-per-kilogram dose, and a maximum single-dose ceiling. Print, for each patient, the calculated dose (capped at the ceiling) together with a flag if the calculated dose exceeded the ceiling.

```
def cap_dose(weight_kg, mg_per_kg, ceiling):
    """Return (dose_mg, was_capped) for a single patient."""
    raw = weight_kg * mg_per_kg
    if raw > ceiling:
        return ceiling, True
    else:
        return raw, False

weights = [25, 40, 60, 75, 95]      # kg
mg_per_kg = 15
ceiling = 1000                      # mg

print(f"{'Wt':>4} | {'Dose':>6} | Capped?")
print("-" * 26)
for w in weights:
    dose, capped = cap_dose(w, mg_per_kg, ceiling)
    flag = "YES" if capped else ""
    print(f"{w:>4} | {dose:>6} | {flag}")
```

Output

```
  Wt |   Dose | Capped?
--------------------------
  25 |    375 |
  40 |    600 |
  60 |    900 |
  75 |   1000 | YES
  95 |   1000 | YES
```

Several ideas come together here. The function `cap_dose` keeps the per-patient logic in one place. The for loop walks the list. The expression `"YES" if capped else ""` is the so-called conditional expression, a one-line shorthand for an if-else when both branches yield a value. (Pythonistas call it the ternary operator. Other languages spell it with a question mark.)

FROM THE BENCH

This pattern (‘compute, then cap’) is exactly how the British National Formulary (BNF) and the National Formulary of India (NFI) express most paediatric doses: weight-based for small patients, but with a hard adult-size ceiling. Reading those formularies after writing this program feels different. The rules are no longer abstract instructions. They are recipes you have already coded.

2.4.2 A BMI calculator with WHO categories

Specification: ask the user for height (m) and weight (kg). Compute the BMI to two decimals, and print the WHO category. As a bonus, warn the user if the inputs are obviously wrong (a height of 30 m or a weight of 5 kg suggests a typo).

```
def categorise_bmi(bmi):
    """Return the WHO weight category for the given BMI value."""
    if bmi < 18.5:
        return "Under-weight"
    elif bmi < 25:
        return "Normal"
    elif bmi < 30:
        return "Over-weight"
    else:
        return "Obese"

def main():
    h = float(input("Height (m): "))
    w = float(input("Weight (kg): "))

    if h < 0.5 or h > 2.5:
        print("Suspicious height, please re-enter in metres, not centimetres.")
        return
    if w < 10 or w > 250:
        print("Suspicious weight, please check the value.")
        return

    bmi = w / (h ** 2)
    print(f"BMI = {bmi:.2f}  →  {categorise_bmi(bmi)}")

if __name__ == "__main__":
    main()
```

The bottom-line idiom if __name__ == "__main__": main() is a small piece of Python convention worth knowing. When a Python file is run directly, Python sets the special variable __name__ to "__main__"; when the same file is *imported* from another script (so you can reuse one of its functions), Python sets it to the module's own name. The guard means the interactive prompts only fire when the file is run directly, importing it stays silent. You will see this idiom at the bottom of almost every Python file in the wild.

Output

```
Height (m): 1.62
Weight (kg): 78
BMI = 29.72  →  Over-weight
```

Two things are worth pointing out. The function categorise_bmi has been pulled out of the main flow because it has its own well-defined job, given a number, return a label. This is a small example of *decomposition*: each function carries one responsibility, no more. And the early return statements after a suspicious input mean that the rest of the function is skipped without needing a deeply nested if-else. Code that bails out early at the start of the function is always easier to read than code that wraps the whole logic in another conditional.

2.4.3 Asian-specific BMI cut-offs

WHO recognises that Asian and South Asian populations develop diabetes-related risk at lower BMI values. The Indian Council of Medical Research therefore recommends 23 kg/m² as the boundary between normal and over-weight, and 25 kg/m² as the boundary between over-weight and obese. Adapt the function above:

```
def categorise_bmi_asian(bmi):
    if bmi < 18.5:   return "Under-weight"
    elif bmi < 23:   return "Normal"
    elif bmi < 25:   return "Over-weight"
    else:            return "Obese (Asian cut-off)"
```

The same control structure, the same shape of code, only the breakpoints have changed. A small reminder that programs encode policy, not eternal truth, and that the policy must be reviewed.

A NOTE ON THE CONTROVERSY

The cut-offs above (23 / 25) follow the WHO Asian-Pacific 2004 guidelines, which are the values most often quoted in Indian public-health work. The ICMR-INDIAB programme has at various points proposed slightly different thresholds (23 / 27.5 in some publications) on the grounds that 25 is too aggressive a definition of obesity for the Indian population. Both positions are defensible; the choice depends on the public-health question. Cite the source you actually use.

2.5 Worked examples

Example 2.1: Counting febrile patients

Given a list of body temperatures (°C), print how many of them are above 38 °C.

```
temps = [36.7, 38.4, 37.5, 39.1, 38.0, 37.0, 38.6]

febrile = 0
for t in temps:
    if t > 38:
        febrile += 1
```

```
print(f"{febrile} patients out of {len(temps)} are febrile.")
```

Output

```
3 patients out of 7 are febrile.
```

Example 2.2: Renal-dose adjustment (illustrative; drug-specific in practice)

Important caveat. Renal dose adjustments are exquisitely drug-specific. Vancomycin uses different cut-offs from metformin, and both differ from gentamicin. The function below is shown only to teach the if-elif-else pattern; do not apply this rule to a real drug without checking its prescribing information. We pick ranitidine here because its renal adjustment is well documented and reasonably representative.

Specification: for ranitidine, halve the dose if CrCl < 50 mL/min, and avoid the drug entirely if CrCl < 10 mL/min. Write a function adjust_ranitidine_dose(crcl, normal_dose) that returns the appropriate dose, or the string 'AVOID, substitute with alternative agent'.

```
def adjust_ranitidine_dose(crcl, normal_dose):
    """Illustrative renal adjustment for ranitidine. Always cross-check the BNF/NFI."""
    if crcl < 10:
        return "AVOID: substitute with alternative agent"
    elif crcl < 50:
        return normal_dose / 2
    else:
        return normal_dose

print(adjust_ranitidine_dose(80, 150))   # full dose
print(adjust_ranitidine_dose(30, 150))   # half dose (renal impairment)
print(adjust_ranitidine_dose(5, 150))    # avoid
```

Output

```
150
75.0
AVOID: substitute with alternative agent
```

Example 2.3: Generating a dose-response table

Print a table of dilution factors and the resulting concentrations starting from a stock of 1000 µg/mL, halving each step, until the concentration falls below 1 µg/mL.

```
stock = 1000.0    # µg/mL
factor = 1

while stock >= 1.0:
    print(f"1:{factor:>5d}  →  {stock:>8.2f} µg/mL")
    stock = stock / 2
    factor *= 2
```

Output

```
1:    1  →   1000.00 µg/mL
1:    2  →    500.00 µg/mL
1:    4  →    250.00 µg/mL
...
1:  512  →      1.95 µg/mL
```

Example 2.4: Eligibility checker as a function

Encapsulate the trial eligibility logic in a function that returns True or False.

```
def is_eligible(age, weight_kg, is_pregnant):
    """Eligibility for a phase-II oncology study."""
    if is_pregnant:
        return False
    if age < 18 or age > 65:
        return False
    if weight_kg <= 45:
        return False
    return True

print(is_eligible(32, 58, False))   # True
print(is_eligible(70, 70, False))   # False, age
print(is_eligible(40, 40, False))   # False, weight
```

2.6 Chapter summary

- Conditional statements: if, if-else, if-elif-else and nested forms select among different paths.
- Indentation, not braces, marks the body of every block. Always four spaces.
- Falsy values are: False, 0, 0.0, "", [], (), {}, None. Everything else is truthy.
- for loops walk through a sequence; while loops repeat as long as a condition is true.
- range() generates integers; enumerate() pairs each item with its index.
- break leaves a loop entirely; continue jumps to the next iteration.
- Functions package up reusable behaviour. They have parameters, an optional docstring, and a return value.

- Default parameter values and named arguments make function calls more readable.
- Decompose a problem into small functions, each with a single responsibility.

2.7 Self-assessment: multiple-choice questions

Q1. What does the keyword 'elif' stand for in Python?

(a) else if
(b) every if
(c) end if
(d) exception if

Q2. Which of these is the correct way to start a for loop over a list named patients?

(a) for patient on patients:
(b) foreach patient in patients:
(c) for patient in patients:
(d) loop patient over patients:

Q3. What is the output of range(2, 8, 2) ?

(a) 2, 3, 4, 5, 6, 7
(b) 2, 4, 6
(c) 2, 4, 6, 8
(d) 2, 4

Q4. Which keyword exits a loop completely?

(a) continue
(b) stop
(c) exit
(d) break

Q5. Which keyword skips the remainder of the current iteration?

(a) pass
(b) skip
(c) continue
(d) break

Q6. How is a function defined in Python?

(a) function name():
(b) def name():
(c) fn name():
(d) create name():

Q7. Which of the following is FALSY in Python?

(a) "0"
(b) [0]
(c) 0.0

(d) "False"

Q8. A function with no return statement returns:

(a) 0
(b) False
(c) None
(d) an empty string

Q9. Which call below uses keyword arguments?

(a) calc(60, 10)
(b) calc(weight_kg=60, mg_per_kg=10)
(c) calc[60, 10]
(d) calc.weight_kg(60).mg_per_kg(10)

Q10. Which of the following BMI values would be classified as 'Obese' by the WHO standard cut-offs?

(a) 22.4
(b) 27.9
(c) 29.5
(d) 31.2

Q11. What does the symbol += mean?

(a) assignment
(b) addition only
(c) add and assign back
(d) compare for equality

Q12. What will the loop for i in range(3): print(i) print?

(a) 1 2 3
(b) 0 1 2 3
(c) 0 1 2
(d) 1 2

Q13. Which of the following is NOT a feature of Python functions?

(a) docstrings
(b) default values for parameters
(c) multiple return values
(d) type declarations of parameters are mandatory

Q14. A while loop whose condition is permanently True is called:

(a) infinite loop
(b) iterative loop
(c) definite loop
(d) broken loop

Q15. Which expression is the conditional one-liner equivalent to: if a > 0: x = 'pos' else: x = 'neg' ?

(a) x = 'pos' if a > 0 else 'neg'
(b) x = 'pos' ? a > 0 : 'neg'

(c) x = if a > 0 'pos' else 'neg'
(d) x = 'pos' :: a > 0 :: 'neg'

ANSWERS AND EXPLANATIONS

1. (a) else if elif is short for 'else if'.

2. (c) for patient in patients: Python's for-in syntax.

3. (b) 2, 4, 6 range(start, stop, step). The stop value is excluded.

4. (d) break break exits the enclosing loop.

5. (c) continue continue skips to the next iteration.

6. (b) def name(): def is the function-definition keyword.

7. (c) 0.0 numeric zero is falsy. Non-empty strings and lists are truthy even if their contents are 0.

8. (c) None omitting return is equivalent to returning None.

9. (b) calc(weight_kg=60, mg_per_kg=10) named arguments use the parameter names.

10. (d) 31.2 BMI ≥ 30 is the WHO obese cut-off.

11. (c) add and assign back x += 5 is equivalent to x = x + 5.

12. (c) 0 1 2 range is half-open: it includes the start but excludes the stop.

13. (d) parameter type declarations are mandatory Python lets you add type hints, but they are optional.

14. (a) infinite loop it will run forever unless interrupted.

15. (a) x = 'pos' if a > 0 else 'neg' Python's conditional expression.

2.8 Short-answer questions

1. Explain, with a flowchart, the difference between if-else and if-elif-else.
2. State two situations in which a while loop is more appropriate than a for loop.
3. Define the terms ‘parameter’ and ‘argument’ as used in Python functions.
4. What is the difference between break and continue inside a loop?
5. Give one pharmaceutical example each for the four BMI categories used in this chapter.
6. Why is it considered good practice to keep functions short and single-purpose?
7. Write a Python function safe_divide(a, b) that returns a / b when b is not zero, and the string 'undefined' otherwise.

2.9 Practical exercises

LAB 2.1: PAEDIATRIC PARACETAMOL CALCULATOR

A paediatric dose of paracetamol is 15 mg/kg, given every 4–6 hours, with a maximum single dose of 1000 mg and a maximum daily dose of 60 mg/kg or 4000 mg, whichever is lower. Write a Python program that asks for the child's weight in kilograms and prints the recommended single dose and the maximum daily dose.

LAB 2.2: BMI COHORT SUMMARY

Given the following patient data (you may hard-code it as a list of tuples), use a for loop to print the name, BMI and category of each patient, and finally the count of patients in each category. Use the function categorise_bmi() from §2.4.2.

Sample data: [('Mr Rao', 1.70, 84), ('Mrs Iyer', 1.62, 58), ('Master Khan', 1.20, 28), ('Ms Das', 1.65, 92), ('Mr Singh', 1.78, 70)].

LAB 2.3: SERIAL-DILUTION SCHEDULE PRINTER

In a stability study you start with a stock concentration of 1280 µg/mL of an API. You halve it on each step of a serial dilution until it falls below 1 µg/mL. Write a program that uses a while-loop to print, for each step, the step number, the dilution factor (1:1, 1:2, 1:4, …) and the resulting concentration.

Note: earlier editions of this book used a steroid-tapering schedule for this exercise. We replaced it because realistic clinical tapers are slow (typically 5 mg per week or slower below 20 mg of prednisolone-equivalent) to allow recovery of the hypothalamic-pituitary-adrenal axis. Programming a fast-and-fictional taper risks fixing an incorrect schedule in the student's memory. Serial dilution exercises the same loop pattern with no clinical risk.

LAB 2.4: TRIAL ELIGIBILITY BATCH RUN

Take the eligibility function from §2.5 and run it over a list of ten candidate dictionaries (each with age, weight_kg and is_pregnant). Print a small report saying how many candidates are eligible and listing the IDs of the ineligible ones.

UNIT 3

Data Structures & File Handling

3.0 Learning outcomes for this unit

By the end of this unit you should be able to:

1. **Use lists, tuples and dictionaries** to store and organise pharmaceutical data of mixed shapes.
2. **Index and slice** sequences confidently to extract subsets of patient or drug records.
3. **Manipulate strings** with the methods you learnt in Unit 1, applied to whole files of clinical data.
4. **Create and use NumPy arrays** for fast numerical work.
5. **Read and write CSV files** containing healthcare data: both with the csv module and with NumPy.
6. **Recognise the structure of a clean healthcare dataset** and the kind of cleaning that small datasets typically require.

These outcomes correspond to course outcome CO 3 in the BP101T syllabus.

3.1 Lists

A list is an ordered, mutable sequence of items. By a wide margin it is the data structure you will reach for most often in Python. Square brackets create one:

```
nsaids = ["Paracetamol", "Ibuprofen", "Aspirin", "Diclofenac", "Naproxen"]
doses_mg = [500, 400, 300, 50, 250]
mixed = ["P-1042", 56, 64.5, True]   # any types in the same list
```

Three properties matter. A list is *ordered* the position of each item is meaningful and stable. A list is *mutable* you can add, remove and replace items after the list is created. And a list can hold values of any type, even mixed types in the same list. (Whether you *should* mix types in one list is a different question. Usually no.)

3.1.1 Indexing and slicing

Items are addressed by position, starting from zero. Negative indices count from the end.

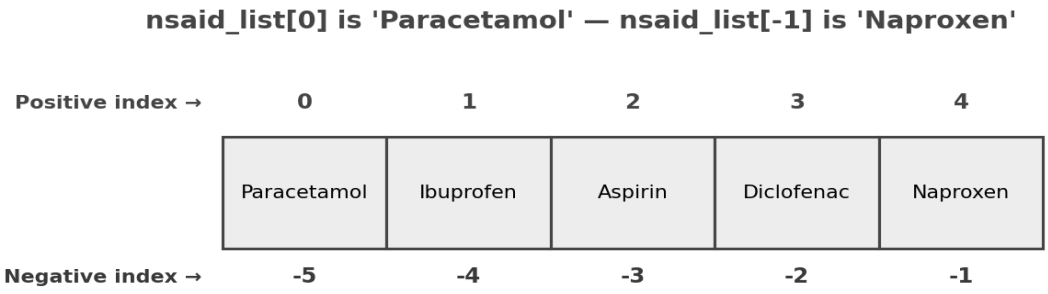

Figure 3.1 Positive indices count from the start; negative indices count from the end.

```
nsaids[0]      # 'Paracetamol'  -- the first item
nsaids[-1]     # 'Naproxen'     -- the last item
nsaids[1:3]    # ['Ibuprofen', 'Aspirin'] -- a slice
nsaids[:2]     # ['Paracetamol', 'Ibuprofen'] -- from the start
nsaids[-2:]    # ['Diclofenac', 'Naproxen'] -- last two
nsaids[::-1]   # the whole list reversed
```

A slice `nsaids[a:b]` gives you a *new* list containing items from index `a` up to (but not including) index `b`. The third part of a slice is the step: `nsaids[::2]` takes every second item, and `nsaids[::-1]` reverses the list. There is more than one way to remember the half-open convention; the easiest one is to count the items in the slice, exactly `b - a` of them.

3.1.2 Common list operations

Operation	What it does	Example	Notes
.append(x)	Add x to the end	doses.append(100)	list grows by one
.extend(L)	Add all items of L to the end	doses.extend([200, 300])	list grows by len(L)
.insert(i,x)	Insert x at index i	doses.insert(0, 25)	everything shifts right
.remove(x)	Delete the first occurrence of x	doses.remove(50)	ValueError if not found
.pop()	Remove and return the last item	last = doses.pop()	
.sort()	Sort in place (default ascending)	doses.sort()	use reverse=True for desc
len(L)	Number of items	len(doses)	→ int
x in L	Membership test	"Aspirin" in nsaids	True

```
doses_mg = [500, 400, 300, 50, 250]
doses_mg.append(100)
doses_mg.sort()
print(doses_mg)
print("max =", max(doses_mg))
print("min =", min(doses_mg))
print("sum =", sum(doses_mg))
```

Output

```
[50, 100, 250, 300, 400, 500]
max = 500
min = 50
sum = 1600
```

3.1.3 Iterating over a list

You will spend much of your Python career writing for-loops that walk over lists:

```
doses_mg = [500, 400, 300, 50, 250]
total = 0
for d in doses_mg:
    total += d
print("Total mg dispensed:", total)
```

Output

```
Total mg dispensed: 1500
```

Of course sum(doses_mg) gives the same answer in one line, and you should prefer the built-in. Python's built-in functions are usually faster and clearer than the equivalent for-loop. But the loop pattern is general (you can do anything inside it, not just add) and once you have written it the first time, every variation comes more easily.

3.1.4 List comprehensions

When you want to build a new list from an existing one (square every value, double every dose, extract every patient older than 60) Python provides a compact construct called a *list comprehension*.

```
doses_mg = [500, 400, 300, 50, 250]

halved = [d / 2 for d in doses_mg]
high   = [d for d in doses_mg if d >= 300]

print(halved)   # [250.0, 200.0, 150.0, 25.0, 125.0]
print(high)     # [500, 400, 300]
```

Read it left to right: ‘the new list is (for each d in doses_mg) d divided by two’. The if-clause filters the source list. List comprehensions are not strictly necessary; you can always write the equivalent for-loop. They are simply a more compact way of saying the same thing.

3.2 Tuples

A tuple looks like a list but is created with parentheses, and once created it cannot be changed. The technical word for this is *immutable*. An immutable container is the right choice when the items belong together as a group and are not going to be edited individually, vital signs taken at a single time-point, the (latitude, longitude) of a clinic, an (R, G, B) colour.

```
vitals = (98.4, 120, 80)       # temp °F, systolic, diastolic
print(vitals[0])               # 98.4
# vitals[0] = 99.0             # → TypeError
```

Two niceties. The parentheses are sometimes optional: `vitals = 98.4, 120, 80` also creates a tuple: Python recognises any comma-separated values as a tuple. And you can *unpack* a tuple into separate variables in one line:

```
temp_f, sys, dia = vitals
print("Systolic =", sys, "Diastolic =", dia)
```

We have already used unpacking in §2.3.3, when a function returned two values. Tuple unpacking is one of Python's most pleasant idioms, and you will use it daily.

3.3 Dictionaries

A dictionary stores key–value pairs. It is the right tool when each item is identified by a label rather than by a position, and so it is ideally suited for representing a single patient, a single drug, or a single ADR record.

```
patient = {
   "id":        "P-1042",
   "name":      "Mrs Iyer",
   "age":       56,
   "weight_kg":  64.5,
   "is_diabetic": True,
}

print(patient["name"])    # 'Mrs Iyer'
patient["age"] = 57       # mutate one field
patient["hba1c"] = 7.4    # add a new field
```

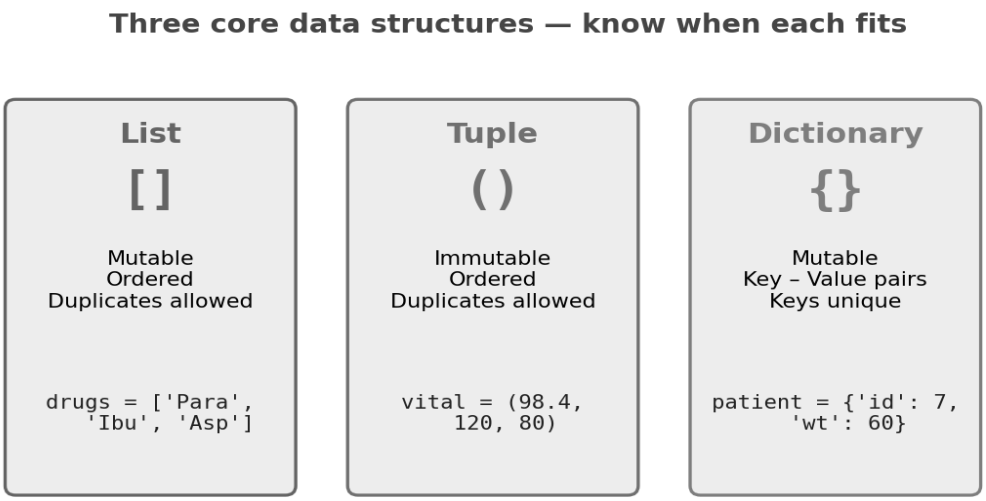

Figure 3.2 When to reach for a list, a tuple or a dictionary.

3.3.1 Useful dictionary operations

Operation	What it does	Example	Notes
d[k]	Get the value at key k	patient["age"]	KeyError if missing
d.get(k, x)	Get value at k or default x	patient.get("phone", "-")	no exception
d[k] = v	Set or update the value at k	patient["age"] = 57	
.keys()	All keys (an iterable view)	patient.keys()	
.values()	All values	patient.values()	
.items()	All (key, value) pairs	patient.items()	useful in for loops
k in d	Membership test on keys	"id" in patient	True
del d[k]	Remove the key	del patient["phone"]	KeyError if missing

Iterating over a dictionary walks its keys by default. Iterating over .items() walks key-value pairs:

```
for k, v in patient.items():
    print(f"{k:>12}: {v}")
```

Output

```
        id: P-1042
      name: Mrs Iyer
       age: 57
 weight_kg: 64.5
is_diabetic: True
      hba1c: 7.4
```

PHARMA EXAMPLE

A drug record. Pharmaceutical reference data are most naturally stored as dictionaries. Each drug becomes a dict with keys for 'name', 'brand_names', 'dose_form', 'strength_mg', 'atc_code', 'pregnancy_category', and so on. A list of such dictionaries is the simplest in-memory representation of a small formulary, and it scales surprisingly far before you really need a database.

3.3.2 A list of dictionaries

In real datasets you almost always have many records of the same shape, many patients, many drugs, many ADRs. The standard representation is a list of dictionaries:

```
drugs = [
    {"name": "Paracetamol",  "strength_mg": 500, "category": "Analgesic"},
    {"name": "Ibuprofen",    "strength_mg": 400, "category": "NSAID"},
    {"name": "Atorvastatin", "strength_mg":  20, "category": "Statin"},
    {"name": "Metformin",    "strength_mg": 500, "category": "Anti-diabetic"},
]

# Print the names of every NSAID in the list:
for d in drugs:
    if d["category"] == "NSAID":
        print(d["name"], "-", d["strength_mg"], "mg")
```

Output

```
Ibuprofen - 400 mg
```

3.4 String manipulation revisited

Unit 1 introduced strings; here we look at one or two more tricks that come up when data are messy, which, in the real world, they almost always are.

3.4.1 split() and join()

The .split(sep) method breaks a string into a list of pieces. The inverse, sep.join(list), glues a list of strings together with a separator. Together they let you reformat data between layouts.

```
row = "P-1042, Mrs Iyer, 56, 64.5"
fields = row.split(",")          # ['P-1042', ' Mrs Iyer', ' 56', ' 64.5']

# Strip the spaces that sneak in after each comma:
fields = [f.strip() for f in fields]
print(fields)

# Re-join with a tab separator, ready to write to a TSV file:
tsv_row = "\t".join(fields)
print(tsv_row)
```

Output

```
['P-1042', 'Mrs Iyer', '56', '64.5']
P-1042    Mrs Iyer 56          64.5
```

3.4.2 Searching with in and find()

To check whether a substring appears anywhere inside a string, use the in operator. To get the position of the first occurrence, use .find(), which returns −1 if the substring is absent.

```
note = "Patient developed a maculopapular rash after starting amoxicillin."

if "rash" in note:
    print("Possible ADR.")

pos = note.find("amoxicillin")
print("amoxicillin starts at position", pos)
```

3.4.3 startswith, endswith, isnumeric

Boolean methods that often save you reaching for a regular expression:

Method	Meaning	Example
.startswith(s)	True if string begins with s	"P-1042".startswith("P-")
.endswith(s)	True if string ends with s	"data.csv".endswith(".csv")
.isnumeric()	True if string is all digits	"42".isnumeric()

Method	Meaning	Example
.isalpha()	True if string is all letters	"Para".isalpha()

3.5 Introduction to NumPy arrays

Lists are flexible but slow when you need to do arithmetic on thousands of numbers. NumPy (short for Numerical Python) is a third-party library that provides a fast, memory-compact array type, together with hundreds of mathematical functions that operate on whole arrays at once. The conventional alias is `np`:

```
import numpy as np

# from a Python list
doses = np.array([500, 400, 300, 50, 250])

# arithmetic happens element-wise
halved = doses / 2
print(halved)        # [250. 200. 150.  25. 125.]
print("mean =", doses.mean(), "  std =", doses.std())
```

Output

```
[250. 200. 150.  25. 125.]
mean = 300.0   std = 168.13
```

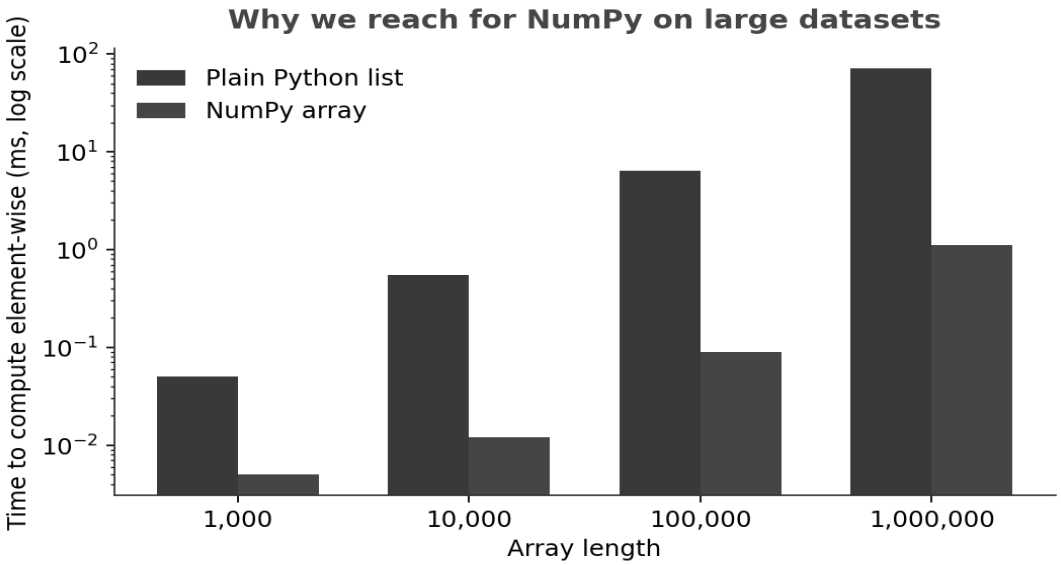

Figure 3.3 Illustrative timing comparison. NumPy is typically faster than a plain Python list for element-wise numerical work, and the gap widens with array size. The exact speedup depends on the operation, the array size, and your hardware; the figure shows a synthetic scenario, not a benchmark. A fair benchmark on your own laptop is one line of Jupyter magic away, try %timeit on a list comprehension and the equivalent NumPy expression.

3.5.1 Creating arrays

Constructor	Meaning	Example	Result
np.array([..])	From an existing list or tuple	np.array([1, 2, 3])	→ array([1,2,3])
np.zeros(n)	Array of zeros	np.zeros(5)	→ array([0,0,0,0,0])
np.ones(n)	Array of ones	np.ones(3)	→ array([1,1,1])
np.arange(a,b,s)	Like range() but a NumPy array	np.arange(0, 10, 2)	→ [0,2,4,6,8]
np.linspace(a,b,n)	n evenly-spaced points from a to b	np.linspace(0, 24, 7)	→ 7 time-points across 24 h
np.random.normal(m,s,n)	n random samples ~ $N(m, s^2)$	np.random.normal(120, 12, 5)	simulate BP readings

3.5.2 Element-wise arithmetic

Once data are inside a NumPy array, arithmetic operates on the whole thing without writing a loop. This is sometimes called 'vectorised' computation; whatever you call it, it is fast:

```
import numpy as np

weights = np.array([55, 62, 70, 80, 95])
doses   = 10 * weights              # element-wise: 10 mg/kg dosing
capped  = np.minimum(doses, 800)    # cap at 800 mg

print("weights =", weights)
print("doses   =", doses)
print("capped  =", capped)
```

Output

```
weights = [55 62 70 80 95]
doses   = [550 620 700 800 950]
capped  = [550 620 700 800 800]
```

3.5.3 Indexing and slicing

NumPy arrays use the same indexing and slicing syntax as lists, but they also support boolean masks. Masks are particularly useful for selecting rows of a dataset:

```
auc = np.array([320, 612, 498, 812, 280, 405])
```

```
high = auc > 500
print(high)             # [False  True False  True False False]
print(auc[high])        # [612 812]
print("high count =", high.sum())   # True is treated as 1
```

3.5.4 Two-dimensional arrays

NumPy can also represent matrices and tables. Pass a list of lists to np.array:

```
table = np.array([
    [10, 100, 200],
    [20, 220, 410],
    [40, 440, 815],
])

print(table.shape)      # (3, 3)  -- rows, cols
print(table[:, 0])      # first column = [10 20 40]
print(table[0, :])      # first row    = [10 100 200]
print(table.mean(axis=0))  # mean of each column
```

3.6 Reading and writing CSV files

CSV (comma-separated values) is the lingua franca of small healthcare datasets. Almost every electronic health record system, every clinical-trial spreadsheet and every pharmacovigilance report can be exported as a CSV. Reading them is therefore one of the most important everyday skills for a Python-using pharmacist.

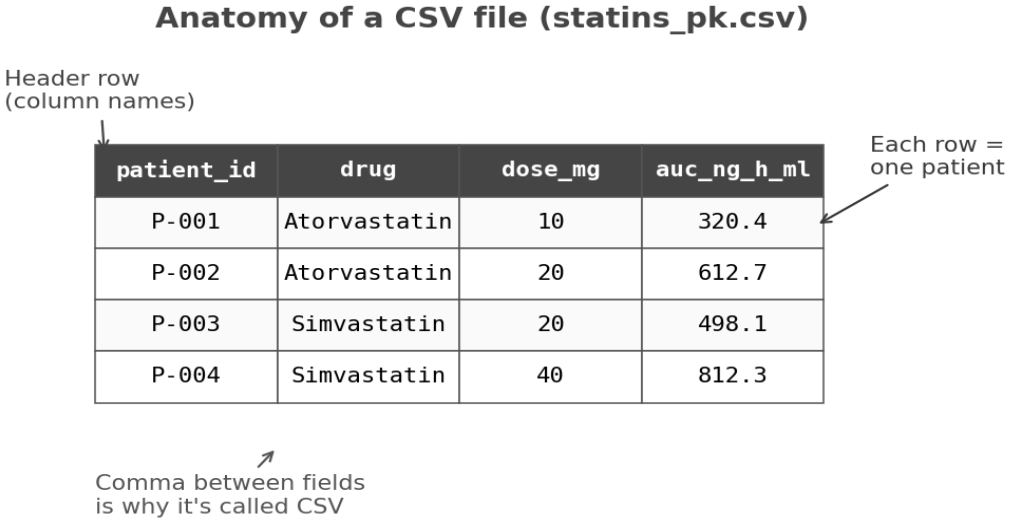

Figure 3.4 The anatomy of a CSV file. The first line is the header; subsequent lines are data rows; commas separate the fields.

3.6.1 Reading with the standard csv module

Python's standard library ships with a csv module that needs no separate installation. The cleanest way to use it is via csv.DictReader, which yields each row as a dictionary keyed by the column name.

Listing 3.1 Read a small statin pharmacokinetics CSV and print every row whose AUC exceeded 500 ng·h/mL.

```
import csv

with open("statins_pk.csv") as f:
    reader = csv.DictReader(f)
    for row in reader:
        auc = float(row["auc_ng_h_ml"])
        if auc > 500:
            print(row["patient_id"], row["drug"], row["dose_mg"], auc)
```

COMMON PITFALL

Why the with-statement? The with open(...) as f: idiom guarantees that the file is closed cleanly when the block ends, even if an error occurs halfway through. If you ever see code that opens a file but never closes it, that code has a quiet bug waiting to happen on someone else's machine. Use with-statements always; the habit is worth more than the small effort of forming it.

3.6.2 Writing a CSV

To write a CSV use csv.DictWriter. You decide the column order in the fieldnames list, then write the header followed by rows. A small twist: open the file with newline="" so that the csv module can manage line endings consistently across operating systems. (If you forget, your file will work on Linux and break on Windows. Or vice versa. The newline argument quietly fixes it.)

```
import csv

rows = [
    {"patient_id": "P-001", "drug": "Atorvastatin", "auc": 320.4},
    {"patient_id": "P-002", "drug": "Atorvastatin", "auc": 612.7},
    {"patient_id": "P-003", "drug": "Simvastatin",  "auc": 498.1},
]

with open("out.csv", "w", newline="") as f:
    writer = csv.DictWriter(f, fieldnames=["patient_id", "drug", "auc"])
    writer.writeheader()
    writer.writerows(rows)
```

Output

```
patient_id,drug,auc
P-001,Atorvastatin,320.4
P-002,Atorvastatin,612.7
P-003,Simvastatin,498.1
```

3.6.3 Reading a CSV directly into a NumPy array

When every column is numeric, np.loadtxt() or np.genfromtxt() can load the file into a 2-D array in a single call. We will see in Unit 4 that Pandas does this even more elegantly, but it is useful to know the bare-NumPy version exists.

```
import numpy as np

# data file: dose_mg, auc
data = np.loadtxt("dose_auc.csv", delimiter=",", skiprows=1)
doses = data[:, 0]
aucs  = data[:, 1]
print("mean dose =", doses.mean(), "  mean AUC =", aucs.mean())
```

3.7 Working with healthcare datasets

A ‘structured healthcare dataset’ is one where each row represents a single observation (a single patient visit, a single dispensing event, a single ADR report) and each column represents a measurable attribute of that observation. This row-and-column shape is, happily, exactly the shape of a CSV. Almost any dataset you can download from Kaggle, the WHO data portal or the Indian government's data.gov.in site will arrive in this form.

3.7.1 Where to find practice datasets

The PCI syllabus points to several reliable sources. Three to bookmark today:

- **Kaggle** (kaggle.com/datasets): search for ‘healthcare’, ‘pharmacy’, ‘adverse drug reactions’. Download requires a free account.
- **WHO Global Health Observatory** (who.int/data/gho): country-level data on disease burden, vaccinations, and the medicines used.
- **data.gov.in** (data.gov.in): the Indian government's open-data portal, with hospital admissions, drug recalls and pharmacopoeia information.
- **ClinicalTrials.gov** (clinicaltrials.gov) and **CTRI** (ctri.nic.in): registered clinical trials. The list is freely downloadable.

FROM THE BENCH

On using data responsibly. Every public dataset carries a licence. Some require attribution, some forbid commercial use, some forbid attempts to re-identify individuals. Read the licence before you publish anything based on the data. And never share patient-level data (even 'de-identified' data) outside your project.

3.7.2 A sample workflow

A typical first session with a new healthcare CSV follows the same five steps:

1. **Look at the file with your eyes** first. Open it in Notepad or Excel. How many rows? What are the columns? Any obviously broken rows?
2. **Read it into Python** with csv.DictReader (or, in Unit 4, with Pandas).
3. **Check the column types** by printing the first row. Numbers should be cast to float; dates should be parsed.
4. **Cleaning.** Strip whitespace, fix obvious typos in categorical columns, decide what to do with missing values.
5. **First analysis.** Compute means, counts, simple cross-tabulations. This is where the data first starts to talk back.

Most of the rest of this book is about the second through fifth steps.

3.7.3 Putting steps 1 and 2 together

Specification: an ADR report file (adr.csv) has columns drug, age, gender, severity, description. Read the file, count how many reports involve each drug, and print the result sorted by frequency.

```
import csv
from collections import Counter

drugs = []
with open("adr.csv") as f:
    for row in csv.DictReader(f):
        drugs.append(row["drug"].strip())

counts = Counter(drugs)
for drug, n in counts.most_common():
    print(f"{drug:>15}: {n}")
```

collections.Counter is a small but extremely useful subclass of dictionary that counts how often each item appears in an iterable. Most pharmacovigilance summaries you will read can be reproduced with this pattern alone, and being able to reproduce them yourself is the difference between trusting a report and just nodding at it.

3.8 Worked examples

Example 3.1: Average dose per drug

Given a list of dictionaries with keys ‘drug’ and ‘dose_mg’, compute the average dose for each drug.

```
records = [
    {"drug": "Para", "dose_mg": 500},
    {"drug": "Para", "dose_mg": 1000},
    {"drug": "Ibu",  "dose_mg": 400},
    {"drug": "Ibu",  "dose_mg": 600},
    {"drug": "Asp",  "dose_mg": 75},
]

totals = {}    # drug -> sum of doses
counts = {}    # drug -> count of doses

for r in records:
    drug = r["drug"]
    totals[drug] = totals.get(drug, 0) + r["dose_mg"]
    counts[drug] = counts.get(drug, 0) + 1

for drug in totals:
    print(f"{drug}: mean dose = {totals[drug] / counts[drug]:.1f} mg")
```

Output

```
Para: mean dose = 750.0 mg
Ibu: mean dose = 500.0 mg
Asp: mean dose = 75.0 mg
```

Example 3.2: Filtering with a list comprehension

Keep only the rows where age is at least 18 and gender is 'F'.

```
records = [
    {"id": "P1", "age": 20, "gender": "F"},
    {"id": "P2", "age": 17, "gender": "F"},
    {"id": "P3", "age": 32, "gender": "M"},
    {"id": "P4", "age": 45, "gender": "F"},
]

kept = [r for r in records if r["age"] >= 18 and r["gender"] == "F"]
for r in kept:
    print(r)
```

```
Output
{'id': 'P1', 'age': 20, 'gender': 'F'}
{'id': 'P4', 'age': 45, 'gender': 'F'}
```

Example 3.3: NumPy summary

Given a NumPy array of plasma drug concentrations, print the mean, the standard deviation, and the position of the maximum value.

```
import numpy as np
conc = np.array([0.0, 1.2, 3.4, 5.0, 4.7, 3.1, 1.8, 0.9, 0.4])
print("mean =", conc.mean())
print("std  =", conc.std())
print("argmax =", conc.argmax())
print("max conc reached at index", conc.argmax(), "value =", conc.max())
```

3.9 Chapter summary

- Lists are ordered, mutable, indexed from zero. Slices [a:b] return new lists.
- Tuples are like immutable lists. Use them for data that should not change.
- Dictionaries map keys to values; perfect for one-record-per-thing modelling.
- List comprehensions build new lists in one expression, the idiomatic way to filter or transform.
- NumPy arrays are fast, vectorised numerical containers; arithmetic happens element-wise.
- Use the with-statement to read or write CSV files cleanly.
- csv.DictReader yields each row as a dictionary; csv.DictWriter is its inverse.
- Most healthcare datasets are tabular. Always inspect a file with your eyes before automating it.

3.10 Self-assessment: multiple-choice questions

Q1. What is the result of [1, 2, 3] + [4, 5] ?

(a) [5, 7, 3]
(b) [1, 2, 3, 4, 5]
(c) TypeError
(d) [10]

Q2. Which type is mutable?

(a) str
(b) tuple
(c) list
(d) int

Q3. What does ages[-2:] give for the list [10, 20, 30, 40] ?

(a) [10, 20]
(b) [20, 30]
(c) [30, 40]
(d) [40]

Q4. Which of the following creates an empty dictionary?

(a) []
(b) ()
(c) {}
(d) set()

Q5. Which dictionary method returns a default if the key is missing?

(a) .find()
(b) .get()
(c) .lookup()
(d) .value()

Q6. What is printed by print("a,b,c".split(",")) ?

(a) a b c
(b) ['a', 'b', 'c']
(c) abc
(d) ['a,b,c']

Q7. Which library is the standard for fast numerical arrays?

(a) math
(b) numpy
(c) matplotlib
(d) sklearn

Q8. What does np.array([1,2,3]) * 2 return?

(a) [1,2,3,1,2,3]
(b) [2,4,6]
(c) TypeError
(d) [1,4,9]

Q9. Which Python statement guarantees that a file is closed correctly?

(a) open(...).close()
(b) with open(...) as f:
(c) try: open(...)
(d) auto-close()

Q10. csv.DictReader yields each row as:

(a) a list
(b) a tuple
(c) a dictionary

(d) a NumPy array

Q11. What is the type of (3,) ?

(a) int
(b) list
(c) tuple
(d) set

Q12. Which expression keeps only positive numbers from nums = [3, -1, 4, -5, 9] ?

(a) [n for n in nums if n > 0]
(b) [n if n > 0 in nums]
(c) filter[nums > 0]
(d) nums.positive()

Q13. How do you delete the key 'phone' from dictionary d ?

(a) d.delete('phone')
(b) del d['phone']
(c) remove(d, 'phone')
(d) d.remove('phone')

Q14. Which NumPy function gives 100 evenly-spaced points between 0 and 10?

(a) np.range(0, 10, 100)
(b) np.linspace(0, 10, 100)
(c) np.arange(0, 10, 100)
(d) np.spaced(0, 10, 100)

Q15. What does collections.Counter do?

(a) counts the length of a string
(b) counts how often each item appears in an iterable
(c) creates a counter widget
(d) returns a number larger than the input

ANSWERS AND EXPLANATIONS

1. (b) [1,2,3,4,5] + concatenates lists.

2. (c) list strings, tuples and ints are all immutable in Python.

3. (c) [30, 40] the last two items.

4. (c) {} curly braces create an empty dictionary (an empty set is set()).

5. (b) .get() d.get(key, default) returns default when the key is absent.

6. (b) ['a', 'b', 'c'] split returns a list.

7. (b) numpy NumPy is the standard for numerical arrays.

8. (b) [2, 4, 6] element-wise scalar multiplication.

9. (b) with open(...) as f: the with-statement closes the file at block end.

10. (c) a dictionary DictReader uses the header row as keys.

11. (c) tuple a single-element tuple needs the trailing comma to disambiguate from grouping parentheses.

12. (a) [n for n in nums if n > 0] a filtering list comprehension.

13. (b) del d['phone'] del is the keyword for removing a key.

14. (b) np.linspace(0, 10, 100) start, stop, n. The endpoint is included.

15. (b) counts how often each item appears in an iterable Counter is a dictionary subclass.

3.11 Short-answer questions

1. Distinguish between a list and a tuple. Give one pharmacy-flavoured example where each is the better choice.
2. Why does a dictionary key have to be hashable? Give two examples of valid keys and one example of an invalid key.
3. Explain what a list comprehension is. Rewrite the loop: squares = []; for n in range(5): squares.append(n*n) as a comprehension.
4. Compare and contrast lists with NumPy arrays. When is each preferable?
5. Write the Python code to read a CSV file ‘pk.csv’ that has columns ‘drug’ and ‘auc’, and print the names of all rows where AUC exceeds 500.
6. State three habits that distinguish a clean healthcare dataset from a messy one.

3.12 Practical exercises

LAB 3.1: BUILD A TINY FORMULARY

Create a list of at least eight drug-record dictionaries, each with the keys ‘name’, ‘strength_mg’, ‘category’, and ‘pregnancy_category’. Then write functions: search_by_name(q) that returns all drugs whose name contains q, and list_categories() that returns the unique set of categories present.

LAB 3.2: VITAL-SIGNS CSV

Create a CSV file vitals.csv with columns patient_id, systolic, diastolic, pulse for ten made-up patients. Read it back with csv.DictReader and print the average systolic and the highest pulse.

LAB 3.3: NUMPY CONCENTRATION ANALYSIS

Suppose plasma drug concentrations were measured at hourly intervals for 12 hours: [0.0, 1.2, 3.4, 5.0, 4.7, 3.1, 1.8, 0.9, 0.4, 0.2, 0.1, 0.0]. Use NumPy to compute the Cmax, the time of Cmax, and the mean concentration over the 12 hours.

LAB 3.4: ADR TOP-FIVE REPORT

Take any ADR-like CSV (you may use the one from Lab 4.2). Use the Counter pattern from §3.7.3 to print the five drugs most frequently mentioned. Save the result back to a new CSV called 'top_five.csv'.

UNIT 4

Data Handling with Pandas

4.0 Learning outcomes for this unit

By the end of this unit you should be able to:

1. **Create and inspect** Pandas Series and DataFrame objects from lists, dictionaries, CSV and Excel files.
2. **Use head, tail, info and describe** to get a quick first impression of any new dataset.
3. **Clean a small pharmacy dataset** by detecting missing values, trimming whitespace, and standardising categorical values.
4. **Filter and select** rows and columns of a DataFrame using boolean masks and label-based indexing.
5. **Group rows by category and aggregate** with mean, count, sum and custom functions to produce concise summaries of pharmaceutical data.

These outcomes correspond to course outcome CO 4 in the BP101T syllabus.

4.1 Why Pandas?

When Wes McKinney started writing what would become Pandas in 2008, he was working at a hedge fund and was thoroughly fed up of moving data back and forth between Excel and Python. The name is a contraction of 'panel data', the econometrics term for tables of observations indexed by both time and entity. Pandas was designed for tables. And pharmacy, like medicine, runs on tables. A clinical trial is a table. A pharmacovigilance database is a table. A dissolution study is a table. So Pandas is worth learning early, and well.

What does Pandas give us beyond what we already had with lists, dictionaries and NumPy? Three things, really. It gives us a single object (the DataFrame) that holds an entire table together with its column names and row labels. It gives us a wide vocabulary of methods for asking sensible questions of that table, selection, filtering, grouping, joining, aggregating. And it gives us first-class support for the messiness of real data: missing values, mixed types, dates that arrive in three different formats from three different hospitals.

By convention, Pandas is imported as `pd` the same convention everyone in the scientific Python world has settled on.

```
import pandas as pd
import numpy as np          # often imported alongside
```

4.2 Series and DataFrame

Pandas has two central data containers.

4.2.1 Series: a single column with a name

A Series is essentially a one-dimensional NumPy array with an index attached. Think of it as one column of a spreadsheet.

```
import pandas as pd

doses = pd.Series([500, 400, 300, 50, 250],
                  index=["Para", "Ibu", "Asp", "Dic", "Nap"],
                  name="dose_mg")
print(doses)
print("Mean:", doses.mean())
print("Above 200:", (doses > 200).sum())
```

Output

```
Para    500
Ibu     400
Asp     300
Dic      50
Nap     250
Name: dose_mg, dtype: int64
Mean: 300.0
Above 200: 4
```

The labels on the left are the index. By default Pandas would have used 0, 1, 2, … but here we asked it to use drug abbreviations. `doses["Para"]` gives us 500: like a dictionary lookup; `doses.iloc[0]` gives us 500: like a list lookup. Both styles are available simultaneously.

4.2.2 DataFrame: many columns sharing one index

A DataFrame is what you get when several Series share the same index, a table with labelled rows and labelled columns. We will spend most of the rest of this book operating on DataFrames.

```
data = {
    "patient_id": ["P001", "P002", "P003", "P004"],
    "age":        [54, 47, 61, 59],
    "drug":       ["Atorvastatin", "Atorvastatin", "Simvastatin", "Simvastatin"],
    "dose_mg":    [10, 20, 20, 40],
    "auc":        [320.4, 612.7, 498.1, 812.3],
}
df = pd.DataFrame(data)
print(df)
```

Output

```
  patient_id  age        drug  dose_mg   auc
0       P001   54  Atorvastatin       10  320.4
1       P002   47  Atorvastatin       20  612.7
2       P003   61   Simvastatin       20  498.1
3       P004   59   Simvastatin       40  812.3
```

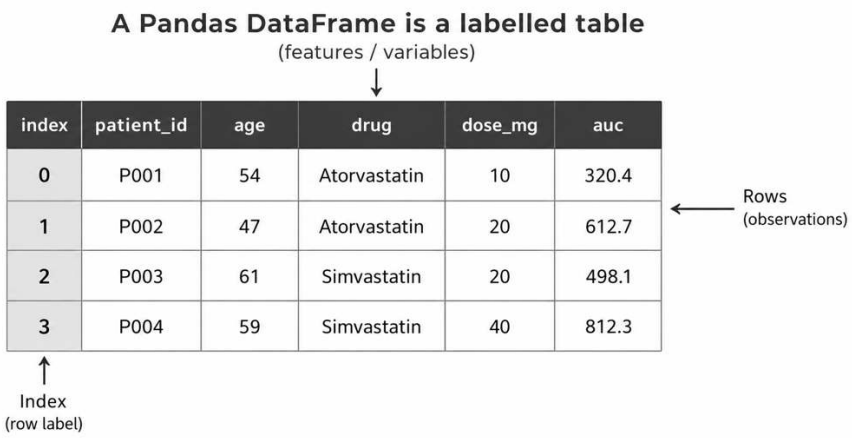

Figure 4.1 A DataFrame is a labelled table. Each column is a Series; each row is one observation; an index labels each row.

Index, columns and values

```
df.index        # RangeIndex(start=0, stop=4, step=1)
df.columns      # Index(['patient_id', 'age', 'drug', 'dose_mg', 'auc'])
df.values       # the numbers/strings as a 2-D NumPy array
df.shape        # (4, 5) -- (rows, columns)
```

4.3 Reading CSV and Excel files

Pandas's real charm shows up the moment you read a real file. The two functions you will use most often are pd.read_csv() and pd.read_excel().

```
import pandas as pd

pk_df  = pd.read_csv("statins_pk.csv")
adr_df = pd.read_excel("adr_reports.xlsx", sheet_name="Q1")
```

That single call pd.read_csv("statins_pk.csv") parses the header row, infers the column types, handles quoted strings that contain commas, and gives you back a DataFrame ready for analysis. It accepts dozens of optional arguments for the messy real-world cases. The most useful are listed below.

Argument	What it controls	When to use	Example
sep=";"	Use ; instead of , as the field separator	European-style CSVs	
encoding="utf-8"	Specify the file encoding	Hospital exports often use 'latin-1'	
dtype={"id": str}	Force a column to a specific type	Stop Pandas from converting IDs like '0042' to 42	
parse_dates=["visit_date"]	Parse dates as datetime	Lets you do .dt.year, .dt.month, etc.	
na_values=["NA", "-", ""]	Treat these strings as missing	Don't trust the file's idea of 'missing'	
skiprows=2	Skip the first n lines	When the file starts with a banner	

PHARMA EXAMPLE

Reading a real ADR Excel file. The Pharmacovigilance Programme of India (PvPI) publishes monthly newsletters and periodic causality-assessment summary files. The Excel exports typically use an 'NA' string for missing data, carry a banner of three rows above the headers, and store report dates as text in DD/MM/YYYY format. A robust read therefore looks like:

```
df = pd.read_excel(
    "pvpi_q3.xlsx",
    sheet_name="reports",
    skiprows=3,
    na_values=["NA", "-", ""],
    parse_dates=["report_date"],
    dayfirst=True,
)
```

4.3.1 Writing back out

Once you have cleaned and analysed a DataFrame, you will often want to save the result. The mirror-image functions are .to_csv() and .to_excel():

```
df.to_csv("clean_adr.csv", index=False)
df.to_excel("clean_adr.xlsx", sheet_name="Q3", index=False)
```

The keyword index=False stops Pandas from writing the integer row index into the file as an extra column, almost always what you want. Forget it once and you will end up with a column called ‘Unnamed: 0’ in your output.

4.4 Inspecting a new dataset

The first five things to do with any DataFrame you have just loaded are:

Method	Returns	Example	Why
.head(n)	First n rows (default 5)	df.head()	Look at the data with your eyes
.tail(n)	Last n rows	df.tail(3)	Spot trailing rubbish rows
.info()	Column names, dtypes, non-null counts	df.info()	Spot wrong types and missing data
.describe()	Summary stats of numeric columns	df.describe()	Means, mins, maxes, quartiles
.shape	Tuple (rows, cols)	df.shape	Sanity check
.columns	Column names as an Index	df.columns	Discover what is available
.dtypes	Type of each column	df.dtypes	Spot strings where numbers should be

Listing 4.1 Inspecting a small statin PK study.

```
import pandas as pd
df = pd.read_csv("statins_pk.csv")

print(df.shape)
print(df.columns)
print()
print(df.head(3))
print()
df.info()
print()
print(df.describe())
```

Output

```
(20, 5)
Index(['patient_id', 'age', 'drug', 'dose_mg', 'auc_ng_h_ml'], dtype='object')

  patient_id  age         drug  dose_mg  auc_ng_h_ml
0       P001   54  Atorvastatin       10        320.4
1       P002   47  Atorvastatin       20        612.7
2       P003   61   Simvastatin       20        498.1

RangeIndex: 20 entries, 0 to 19
Columns: 5 (patient_id, age, drug, dose_mg, auc_ng_h_ml)
```

```
dtypes: object(2), int64(2), float64(1)
Non-null counts:  20  19  20  20  18        <-- 'age' has 1 missing, 'auc' has 2 missing

            age      dose_mg   auc_ng_h_ml
count    19.000     20.000        18.000
mean     54.421     22.500       472.150
min      31.000      5.000       110.300
max      72.000     80.000      1010.700
```

QUICK CHECK

Look at the output above. Without scrolling further, what would you do next? Pause and think. Answer: I would look at the rows with missing age and missing AUC values, to decide how to handle them. We turn to that question now.

4.5 Cleaning and missing values

Real data are dirty. Patient names arrive with mixed cases. Drug names have stray spaces. Cells are empty because someone forgot to record the value, or because the lab's machine broke that morning. The first half of any real analysis is therefore about cleaning. Pandas treats missing values uniformly using the special object NaN (Not a Number). It is what shows up wherever a value is missing.

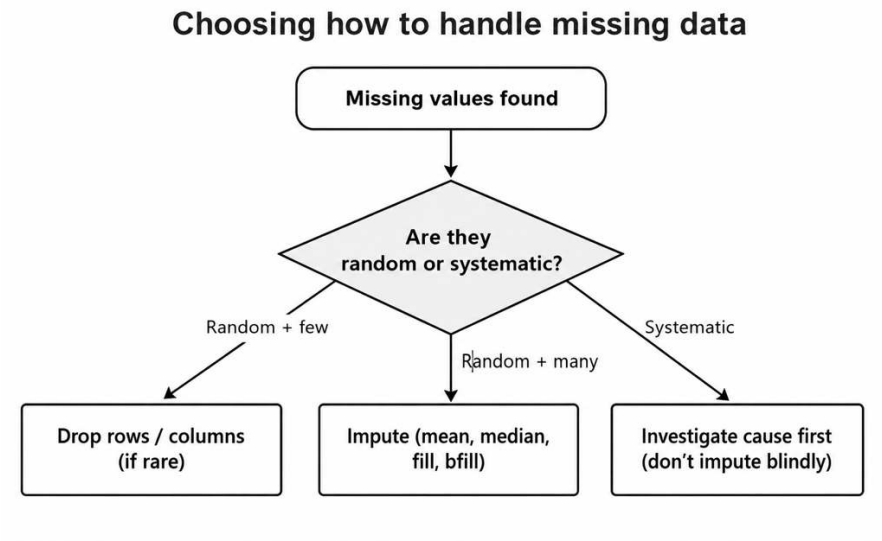

Figure 4.2 A simple decision flow for handling missing values.

4.5.1 Detecting missing values

```
df.isna()          # boolean DataFrame, same shape as df
df.isna().sum()    # count of missing per column
df.isna().any()    # True for columns that have at least one missing
df["age"].isna()   # boolean Series for one column
```

```
print(df.isna().sum())
```

Output

```
patient_id     0
age            1
drug           0
dose_mg        0
auc_ng_h_ml    2
dtype: int64
```

4.5.2 Strategies for missing values

Pandas gives you three broad strategies.

Drop the rows or columns

```
df.dropna()                    # drop any row with a missing value
df.dropna(subset=["auc_ng_h_ml"]) # drop only rows where AUC is missing
df.dropna(axis=1)              # drop columns with any missing value
```

Fill (impute) the missing values

```
df["age"].fillna(df["age"].median())    # fill with the median
df["auc_ng_h_ml"].fillna(0)             # fill with zero (rarely sensible)
df.ffill()                              # forward-fill from the row above
df.bfill()                              # backward-fill from the row below
```

(Note: in older code you may see df.fillna(method="ffill"). That signature is being phased out from Pandas 2.1 onwards in favour of the dedicated ffill() and bfill() methods shown above.)

Leave them in and let aggregations skip them

Most aggregating methods (mean, sum, std) silently skip NaN values, so the simplest strategy is sometimes to do nothing. df["age"].mean() ignores the missing row by default and gives you the mean of the 19 patients with known age. Just be careful not to confuse '19 patients' with 'all 20 patients' when you write up the result.

> **COMMON PITFALL**
> It is tempting to fill all missing values with zero or the column mean and move on. Resist the temptation. A missing AUC may indicate that the patient withdrew from the study, that the assay failed, or that the data were censored deliberately. Each cause demands a different treatment. Always understand why a value is missing before you decide what to do with it.

4.5.3 Cleaning text columns

Pandas exposes the string methods you saw in Unit 1 through the .str accessor. They run vectorised over every value in a column.

```
df["drug"] = df["drug"].str.strip()      # trim whitespace
df["drug"] = df["drug"].str.title()      # 'atorvastatin' → 'Atorvastatin'
df["drug"] = df["drug"].str.replace("  ", " ")

df["drug"].unique()                  # what categories are present?
df["drug"].value_counts()               # frequency of each
```

4.6 Filtering and selecting data

Once a DataFrame is clean, the next thing you usually want is a subset of it. Pandas gives you several ways.

4.6.1 Selecting columns

```
df["auc_ng_h_ml"]              # one column, returns Series
df[["patient_id", "auc_ng_h_ml"]]  # several columns, returns DataFrame
```

4.6.2 Selecting rows by position or label

```
df.iloc[0]          # first row, by integer position (Series)
df.iloc[0:5]         # first five rows
```

```
df.iloc[0, 1]        # cell at row 0, column 1
df.loc[3]            # row whose label is 3 (Series)
df.loc[3, "auc_ng_h_ml"]   # cell at label 3, column 'auc_ng_h_ml'
```

TIP

iloc vs loc. Read .iloc as ‘integer location’, it ignores the label and uses position. Read .loc as ‘label location’, it uses the row index and column name.

4.6.3 Filtering with boolean masks

This is the workhorse. Build a boolean Series with a comparison and use it to slice the DataFrame:

```
high = df[df["auc_ng_h_ml"] > 500]
print(high)
```

Output

```
  patient_id  age         drug  dose_mg  auc_ng_h_ml
1       P002   47  Atorvastatin       20        612.7
3       P004   59   Simvastatin       40        812.3
...
```

Combine multiple conditions with & (and), | (or) and ~ (not). Each condition must be in parentheses, and you must use the symbol forms (not the Python keywords ‘and/or/not’) because the operators have to work element-wise:

```
mask = (df["drug"] == "Atorvastatin") & (df["dose_mg"] >= 20)
print(df[mask])
```

4.6.4 Adding and replacing columns

```
df["clearance"] = df["dose_mg"] / df["auc_ng_h_ml"] * 1000   # mL/h
df["is_on_statin"] = df["drug"].str.endswith("statin")
```

4.6.5 Sorting

```
df.sort_values(by="auc_ng_h_ml", ascending=False)
df.sort_values(by=["drug", "dose_mg"])    # sort by two columns
```

4.7 Group-by and aggregation

Almost every interesting question about a clinical-trial table is of the form ‘mean X by Y’, mean AUC by drug, count of ADRs by reporting hospital, median age by treatment arm. Pandas has a single, beautifully designed mechanism for this: .groupby.

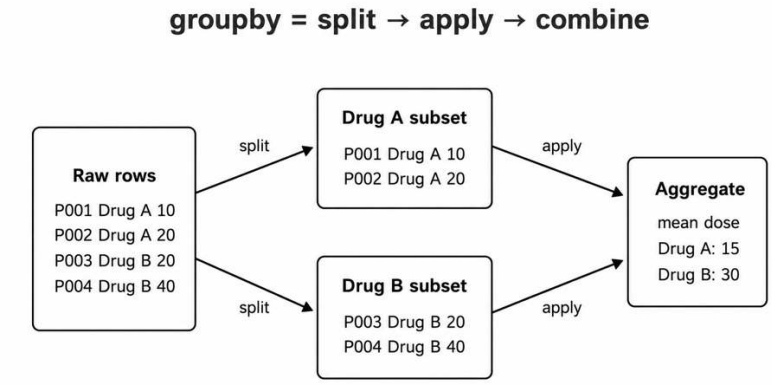

Figure 4.3 groupby = split → apply → combine. Pandas splits the table into one piece per group, applies a function to each, then combines the results.

```
by_drug = df.groupby("drug")

print(by_drug["auc_ng_h_ml"].mean())
print(by_drug["auc_ng_h_ml"].agg(["count", "mean", "std"]))
```

Output

```
drug
Atorvastatin    466.55
Simvastatin     655.20
Name: auc_ng_h_ml, dtype: float64

            count    mean      std
drug
Atorvastatin    10  466.55   124.18
Simvastatin     10  655.20    98.04
```

4.7.1 Multiple grouping columns

You can group by more than one column at once, producing a hierarchical (multi-level) index in the result:

```
df.groupby(["drug", "dose_mg"])["auc_ng_h_ml"].mean()
```

4.7.2 Custom aggregations

Pass a dictionary to .agg() to apply different aggregations to different columns:

```
summary = df.groupby("drug").agg(
    n=("patient_id", "count"),
    mean_auc=("auc_ng_h_ml", "mean"),
    median_age=("age", "median"),
)
print(summary)
```

Output

```
              n   mean_auc   median_age
drug
Atorvastatin  10   466.55       54.0
Simvastatin   10   655.20       60.5
```

The AUC numbers in this teaching dataset are synthetic and chosen for the purposes of illustrating groupby, they do not represent published single-dose AUC values for atorvastatin or simvastatin, which are typically much lower.

FROM THE BENCH

This three-line summary is the kind of table that ends up as Table 1 of a clinical paper or as the lead slide of an R&D meeting. The same pattern works whether the groups are drugs, hospitals, treatment arms, or batches of a manufacturing process.

4.7.3 Pivot tables

When you want a two-way summary (say mean AUC for every (drug × dose) combination) Pandas offers .pivot_table():

```
pivot = df.pivot_table(
    index="drug",
```

```
    columns="dose_mg",
    values="auc_ng_h_ml",
    aggfunc="mean",
)
print(pivot)
```

4.8 Worked examples

Example 4.1: Quarterly ADR summary

Given an ADR DataFrame with columns drug, severity, age and report_date, produce a summary that shows, for each drug, the count of reports and the proportion of severe reactions.

```
import pandas as pd
df = pd.read_csv("adr.csv")

df["is_severe"] = (df["severity"] == "severe")

summary = df.groupby("drug").agg(
    reports=("drug", "count"),
    pct_severe=("is_severe", "mean"),
)
summary["pct_severe"] = (summary["pct_severe"] * 100).round(1)
print(summary.sort_values("reports", ascending=False).head(10))
```

Example 4.2: Cleaning a messy weight column

A weight column has been imported as text because some cells contain ‘kg’ at the end. Strip the unit, convert to numeric, and report how many values failed to convert.

```
df["weight_kg"] = df["weight"].str.replace("kg", "").str.strip()
df["weight_kg"] = pd.to_numeric(df["weight_kg"], errors="coerce")

print("Failed to parse:", df["weight_kg"].isna().sum())
print("Mean weight:",  df["weight_kg"].mean().round(1))
```

The errors="coerce" keyword tells Pandas to set values that cannot be parsed to NaN, instead of raising an exception. That is almost always the safer choice when you are cleaning real-world data.

Example 4.3: Top-five prescribed antibiotics

Given a dispensing log with columns drug_name and date, find the five antibiotics most often dispensed and the percentage of all dispensings each one accounts for.

```
abx = ["Amoxicillin", "Cefuroxime", "Ciprofloxacin", "Doxycycline",
       "Clindamycin", "Azithromycin", "Metronidazole"]

ab_df = df[df["drug_name"].isin(abx)]

top = ab_df["drug_name"].value_counts().head(5)
top_pct = top / len(df) * 100
result = pd.DataFrame({"count": top, "pct_of_total": top_pct.round(2)})
print(result)
```

Example 4.4: Joining two tables

A patients DataFrame has columns patient_id and age. A doses DataFrame has patient_id, drug and auc. Join them so we can later check whether AUC is correlated with age.

```
merged = doses.merge(patients, on="patient_id", how="inner")
print(merged.head())
print("Correlation between age and AUC:", merged["age"].corr(merged["auc"]).round(3))
```

The join key is patient_id. how="inner" keeps only rows present in both tables; how="left" keeps all rows from the left and fills the right with NaN where there is no match. The default in Pandas is ‘inner’, which is sometimes too aggressive and silently drops rows you wanted to keep. Specify the join type explicitly .

4.9 Chapter summary

- Series = a labelled column. DataFrame = a labelled table of Series sharing one index.
- Read with pd.read_csv / pd.read_excel; write with .to_csv / .to_excel.
- Always run .head, .info and .describe on a new dataset before doing anything else.
- Detect missing values with .isna(); decide whether to drop, fill or ignore each case on its own merits.
- Vectorised string methods live under .str.
- Filter rows with boolean masks; combine with & | ~ and parentheses.
- .loc selects by label; .iloc selects by integer position.
- groupby splits the table by a column, applies a function, then combines, the universal pattern for ‘summary by category’.
- pivot_table gives a two-way summary.
- merge joins two DataFrames on a shared key.

4.10 Self-assessment: multiple-choice questions

Q1. Which library is used by convention as ‘pd’?

(a) NumPy
(b) Pandas
(c) Plotly
(d) Pillow

Q2. What is a Series in Pandas?

(a) a 2-D table
(b) a 1-D labelled array
(c) a list of dictionaries
(d) a SQL connection

Q3. Which method reads a CSV file?

(a) pd.import_csv()
(b) pd.load_csv()
(c) pd.read_csv()
(d) pd.csv_read()

Q4. Which method gives the first 5 rows of a DataFrame by default?

(a) df.first()
(b) df.head()
(c) df.top()
(d) df.preview()

Q5. Which method returns a row count, dtypes and non-null counts?

(a) df.describe()
(b) df.summary()
(c) df.info()
(d) df.head()

Q6. Which method gives a quick statistical summary of numeric columns?

(a) df.summary()
(b) df.describe()
(c) df.stats()
(d) df.report()

Q7. What does df.isna().sum() return?

(a) the rows with missing values
(b) the number of missing values per column
(c) True if any value is missing
(d) the column with the most missing values

Q8. Which keyword should you pass to to_csv to avoid writing the row index as a column?

(a) index_col=0

(b) header=False
(c) index=False
(d) no_index=True

Q9. Which symbol joins boolean conditions on a DataFrame?

(a) and
(b) &
(c) *
(d) +

Q10. Which method joins two DataFrames on a shared key?

(a) df.append()
(b) df.merge()
(c) df.combine()
(d) df.concat()

Q11. Which accessor exposes string methods on a column?

(a) .text
(b) .str
(c) .chr
(d) .asstr

Q12. Which call selects rows where age > 60?

(a) df.where(age > 60)
(b) df[age > 60]
(c) df[df.age > 60]
(d) df.select(age > 60)

Q13. After df.groupby("drug")["auc"].mean(), what is the type of the result?

(a) DataFrame
(b) Series
(c) list
(d) dict

Q14. Which keyword in pd.read_csv coerces unparseable values to NaN?

(a) errors="coerce" (in pd.to_numeric, not read_csv)
(b) na_values=[]
(c) skiprows=0
(d) encoding="utf-8"

Q15. Which of the following is NOT a valid way to handle missing data?

(a) drop the rows
(b) fill with mean
(c) fill with median
(d) secretly delete the data file

ANSWERS AND EXPLANATIONS

1. (b) Pandas pd is the conventional alias.

2. (b) a 1-D labelled array a single labelled column.

3. (c) pd.read_csv() the standard CSV reader.

4. (b) df.head() defaults to 5 rows; pass an integer for any other count.

5. (c) df.info() shows columns, dtypes and non-null counts.

6. (b) df.describe() gives count, mean, std, min, quartiles, max.

7. (b) the number of missing values per column True is treated as 1, summed per column.

8. (c) index=False suppresses the row index column.

9. (b) & with parentheses around each clause.

10. (b) df.merge() SQL-style joins.

11. (b) .str strings; .dt for dates, .cat for categories.

12. (c) df[df.age > 60] boolean mask indexing.

13. (b) Series one column per group, indexed by the grouping key.

14. (a) errors="coerce" (in pd.to_numeric) Pandas converts un-parseable values to NaN.

15. (d) the others are legitimate; the last is misconduct.

4.11 Short-answer questions

1. Compare Series and DataFrame, with a one-line example of each.
2. Why is reading data with pd.read_csv() preferred over the standard csv module for analysis?
3. List four arguments of pd.read_csv that are useful when reading messy hospital data.
4. Explain the split-apply-combine paradigm with a pharmacy example.
5. Differentiate between .loc and .iloc with a short example.
6. Describe two strategies for handling missing values in a clinical trial dataset, and give the pros and cons of each.
7. What does df.groupby('drug').agg(['mean','std']) produce?

4.12 Practical exercises

LAB 4.1: PK STUDY SUMMARY

Download or create a CSV called pk.csv with at least 30 rows and columns patient_id, drug, dose_mg, auc_ng_h_ml. Use Pandas to (a) report the number of patients per drug, (b) the mean and standard deviation of AUC per drug, and (c) the patient with the highest AUC.

LAB 4.2: ADR CLEANING AND COUNTING

Take any ADR Excel or CSV file. Trim whitespace and standardise the case of the drug-name column. Count reports per drug and write the cleaned data and the summary table back to two new files.

LAB 4.3: MISSING-VALUE REPORT

Given any dataset with at least 100 rows, write a function missing_report(df) that returns a DataFrame with columns column, n_missing, pct_missing, sorted by pct_missing descending.

LAB 4.4: PIVOT OF DOSE-RESPONSE

Suppose you have a dataset with columns drug, dose_mg and auc. Use pivot_table to compute the mean AUC for every (drug × dose) combination. Verify that, within each drug, AUC increases roughly linearly with dose.

UNIT 5

Data Visualisation with Matplotlib

5.0 Learning outcomes for this unit

By the end of this unit you should be able to:

1. **Construct line plots, histograms, scatter plots, box plots and bar plots** with Matplotlib.
2. **Add appropriate axis labels, titles, legends and tick marks** so that a plot stands on its own outside your notebook.
3. **Visualise three pharmaceutical datasets called out by the syllabus** concentration-time curves (oral and IV), ADR reporting rates, and dissolution profiles.
4. **Read a plot scientifically** and translate its visual features back into pharmacological language.

These outcomes correspond to course outcome CO 5 in the BP101T syllabus.

5.1 The Matplotlib mental model

Matplotlib is the oldest and most widely used plotting library in scientific Python. It was written by John Hunter in 2003 to give people coming from MATLAB a familiar plotting interface in Python, and the influence of MATLAB shows in the names of the functions you will type. Once you have learnt Matplotlib, almost every other plotting library in Python feels like a cousin: Seaborn, Pandas's built-in plotting, even some of the dashboards built on top of Streamlit and Dash all sit on top of Matplotlib's ideas.

The conventional import is:

```
import matplotlib.pyplot as plt
import numpy as np
```

Almost every plot you will draw follows the same six-step recipe:

1. **Prepare the data** (usually two NumPy arrays or two Pandas Series).
2. **Create a figure and axes** with plt.subplots().
3. **Draw the data** with one of the plotting methods on the axes, ax.plot(), ax.hist(), ax.scatter() and so on.
4. **Label** the axes, the title and the legend.
5. **Polish** grid, ticks, colours.
6. **Show or save** with plt.show() or fig.savefig("out.png", dpi=200).

TIP

Figures and axes. A *figure* is the entire image: the whole rectangle of pixels. An *axes* is one plot inside it. A figure can have many axes (subplots) arranged in a grid. The split sounds confusing at first;

after the first afternoon it feels obvious. The single phrase fig, ax = plt.subplots() creates a figure with one axes inside it, the case you will use ninety-five per cent of the time.

5.2 Line plots: concentration-time curves

The line plot is the workhorse of pharmacokinetics. The x-axis is almost always time after dose; the y-axis is plasma concentration. Every PK textbook in the world is, in some sense, a string of line plots one after another.

5.2.1 Your first PK plot

After an intravenous bolus of 8 mg/L the drug eliminates with first-order kinetics, k_e = 0.25 per hour. Plot plasma concentration over 12 hours.

```
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(0, 12, 200)        # 200 time-points
k_e = 0.25                          # /h
C0 = 8.0                            # mg/L
C  = C0 * np.exp(-k_e * t)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(t, C, color="#1F4E79", linewidth=2, label="IV bolus")
ax.set_xlabel("Time after dose (h)")
ax.set_ylabel("Plasma concentration (mg/L)")
ax.set_title("First-order elimination after IV bolus")
ax.legend()
ax.grid(True, linestyle=":", alpha=0.5)
fig.savefig("iv_curve.png", dpi=200)
plt.show()
```

5.2.2 Comparing two routes of administration

On the same axes, compare the IV bolus curve from the previous example with an oral dose modelled by a one-compartment first-order absorption model.

```
ka, ke = 1.4, 0.25
F, D, V = 0.8, 100, 12
oral = (F * D * ka) / (V * (ka - ke)) * (np.exp(-ke * t) - np.exp(-ka * t))

fig, ax = plt.subplots(figsize=(8, 4.2))
ax.plot(t, C,    color="#1F4E79", linewidth=2, label="IV bolus")
ax.plot(t, oral, color="#C00000", linewidth=2, label="Oral tablet")
```

```
ax.set_xlabel("Time after dose (h)")
ax.set_ylabel("Plasma concentration (mg/L)")
ax.set_title("IV vs oral concentration-time curves")
ax.legend(frameon=False)
ax.grid(True, linestyle=":", alpha=0.6)
```

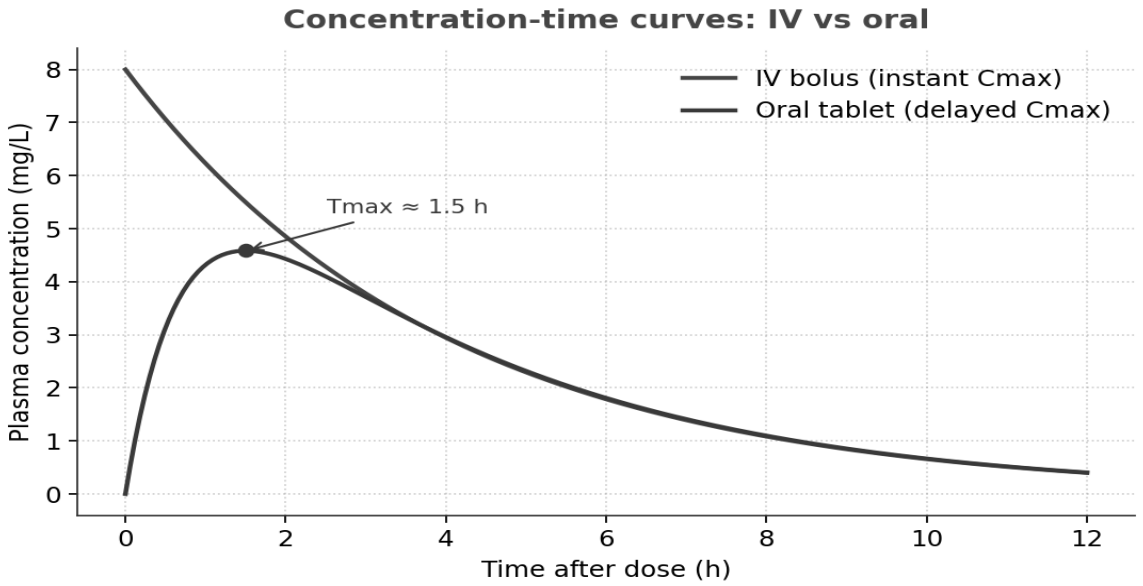

Figure 5.1 Concentration-time curves after IV bolus (blue) and oral administration (red). The oral curve has a clear absorption phase before peaking at Tmax.

FROM THE BENCH

Reading this figure scientifically. Three things any working pharmacist would notice at a glance: (i) Cmax of the IV curve is reached at t = 0 because the drug enters the blood instantaneously; (ii) Cmax of the oral curve is reached at about 1.6 h, the time at which the rate of absorption equals the rate of elimination; (iii) the late phase of both curves declines roughly in parallel, consistent with the same elimination rate constant. Plots are not just decoration; they are the natural language of pharmacokinetics.

5.2.3 Useful keyword arguments to plot()

Argument	What it controls	Values	Notes
color="#1F4E79"	Line colour	any hex or HTML name	default cycles by call
linewidth=2	Line thickness in points	default 1.5	use 2-3 for prints
linestyle="--"	Pattern: solid, dashed, dotted, …	'-', '--', ':', '-.'	

Argument	What it controls	Values	Notes
marker="o"	Add a marker at every point	'o', 's', '^', 'x'	use sparingly
label="IV"	Used by ax.legend()	any string	
alpha=0.7	Transparency 0-1	0=invisible, 1=opaque	useful with overlap

5.2.4 Log-scale axes

Pharmacokinetic curves often look prettier on a log-scale because elimination is exponential, which becomes a straight line on a semilogarithmic plot:

```
fig, ax = plt.subplots(figsize=(8, 4))
ax.semilogy(t, C, color="#1F4E79", linewidth=2)
ax.set_xlabel("Time (h)")
ax.set_ylabel("Concentration (mg/L, log)")
ax.set_title("Semilogarithmic plot of IV bolus")
```

5.3 Histograms and box plots

5.3.1 Histograms

A histogram is a bar plot of how often each value range occurs. It is the right first plot for any single-variable distribution, patient ages, dose ranges, creatinine readings, anything you have measured on a numeric scale.

```
rng = np.random.default_rng(11)
ages = np.concatenate([rng.normal(35, 8, 80), rng.normal(62, 10, 110)])
ages = np.clip(ages, 18, 90)        # truncate at adult range

fig, ax = plt.subplots(figsize=(7.5, 4))
ax.hist(ages, bins=18, color="#1F4E79", edgecolor="white")
ax.set_xlabel("Age (years)")
ax.set_ylabel("Number of patients")
ax.set_title("Patient-age distribution in a metformin cohort")
ax.grid(axis="y", linestyle=":", alpha=0.5)
```

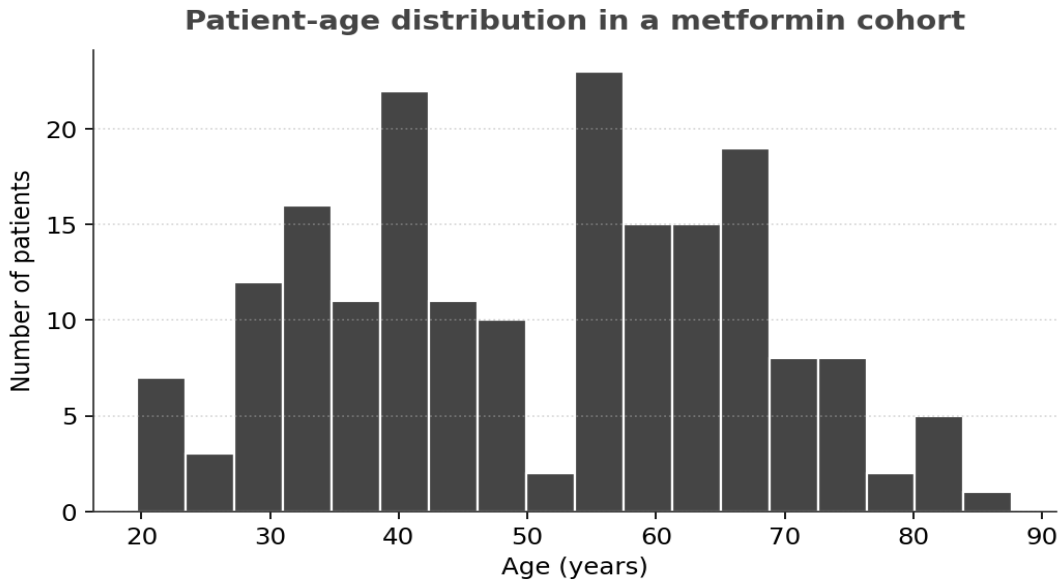

Figure 5.2 Patient-age distribution. The bimodal shape (a younger and an older peak) is consistent with a clinic that sees both newly-diagnosed patients and long-term users.

5.3.2 Box plots

A box plot is the standard way to compare distributions across categories. The box shows the inter-quartile range (25–75 %), the line in the middle is the median, the whiskers extend to most of the data, and points outside the whiskers are outliers.

```
rng = np.random.default_rng(7)
g1 = rng.normal(120, 15, 40)       # placebo SBP
g2 = rng.normal(108, 12, 40)       # drug A
g3 = rng.normal(95, 14, 40)        # drug B

fig, ax = plt.subplots(figsize=(7.5, 4.2))
bp = ax.boxplot([g1, g2, g3],
                labels=["Placebo", "Drug A", "Drug B"],
                patch_artist=True, widths=0.55,
                medianprops=dict(color="black"))
for patch, c in zip(bp["boxes"], ["#555", "#1F4E79", "#C00000"]):
    patch.set_facecolor(c); patch.set_alpha(0.6)
ax.set_ylabel("Systolic BP (mmHg)")
ax.set_title("BP reduction across three trial arms")
```

On compatibility: the keyword used to be `labels=` in older matplotlib versions and was renamed to `tick_labels=` in matplotlib 3.9. Both forms work in current releases, but `labels=` is safer if you do not know the version installed on a college lab machine.

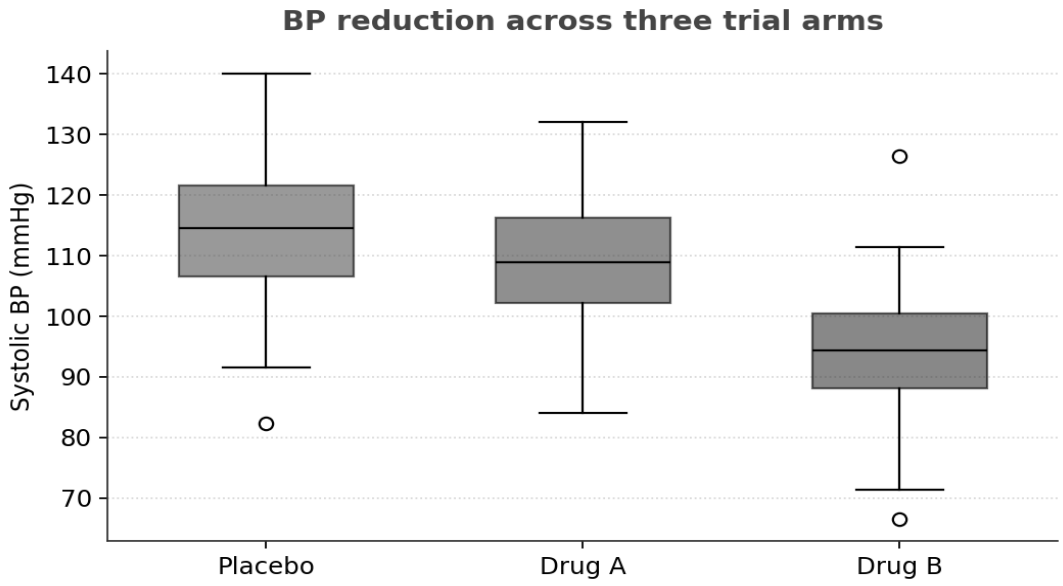

Figure 5.3 Systolic blood pressure across three trial arms. The two drug arms have lower medians than placebo, and Drug B has both a lower median and a tighter inter-quartile range.

5.4 Scatter plots: dose, AUC, and other relationships

A scatter plot draws one dot per (x, y) pair and is the visual workhorse for any ‘is X related to Y?’ question. Pharmacy is full of such questions: is AUC linear in dose? Does plasma concentration of an antibiotic predict its bactericidal effect? Is age correlated with creatinine clearance? The first move on any such question is to plot the points and look at the cloud.

```
rng = np.random.default_rng(42)
dose = np.array([5, 10, 10, 20, 20, 40, 40, 80, 80, 160])
auc  = dose * 32 + rng.normal(0, 50, dose.size) + 80

fig, ax = plt.subplots(figsize=(7.5, 4.2))
ax.scatter(dose, auc, color="#1F4E79", s=55,
           edgecolor="white", linewidth=1.5)

# linear fit
m, b = np.polyfit(dose, auc, 1)
xs = np.linspace(0, 170, 50)
ax.plot(xs, m * xs + b, "--", color="#C00000",
        label=f"AUC = {m:.1f}·Dose + {b:.0f}")

ax.set_xlabel("Dose (mg)")
ax.set_ylabel("AUC (ng·h/mL)")
ax.set_title("Dose-AUC linearity (a quick scatter check)")
ax.legend(frameon=False)
ax.grid(True, linestyle=":", alpha=0.5)
```

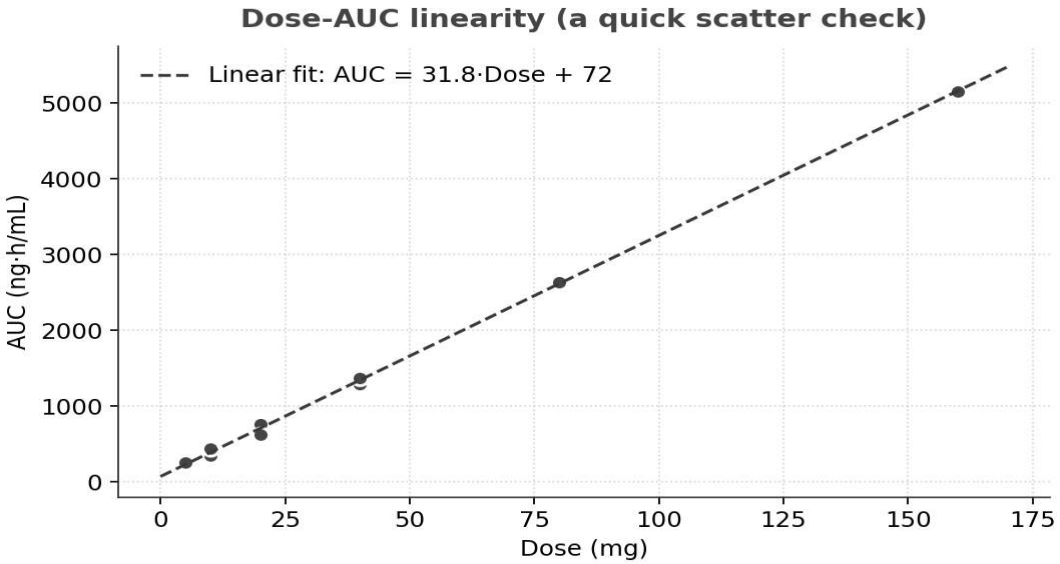

Figure 5.4 AUC against dose. The dashed line is a linear regression. Visually, the points scatter symmetrically around the line (there is no obvious curvature) supporting linear pharmacokinetics over the dose range tested.

QUICK CHECK

If the points in a dose-AUC plot curved upwards, what would that suggest about the drug? Pause and think before reading on. Answer: it would suggest non-linear, capacity-limited pharmacokinetics. The textbook example is phenytoin, its metabolism saturates within the therapeutic range itself, which is precisely why phenytoin requires therapeutic drug monitoring (TDM). A small dose increase from, say, 300 mg to 400 mg can produce a disproportionately large jump in plasma concentration.

5.5 Bar plots: ADR reporting rates

Bar plots are best when you compare a numeric quantity across a small number of discrete categories, drugs, hospitals, treatment arms.

```
drugs = ["Amoxicillin", "Diclofenac", "Metformin",
         "Ramipril", "Atorvastatin", "Phenytoin"]
rates = [12, 18, 8, 7, 5, 14]   # ADR reports per 1000 prescriptions

fig, ax = plt.subplots(figsize=(8, 4))
bars = ax.bar(drugs, rates, color="#1F4E79")
bars[1].set_color("#C00000"); bars[5].set_color("#C00000")    # highlight outliers
for b, r in zip(bars, rates):
    ax.text(b.get_x() + b.get_width() / 2, r + 0.3,
            str(r), ha="center")
ax.set_ylabel("ADR reports per 1000 prescriptions")
ax.set_title("Reported ADR rates from a hospital pharmacovigilance audit")
```

```
plt.setp(ax.get_xticklabels(), rotation=20, ha="right")
```

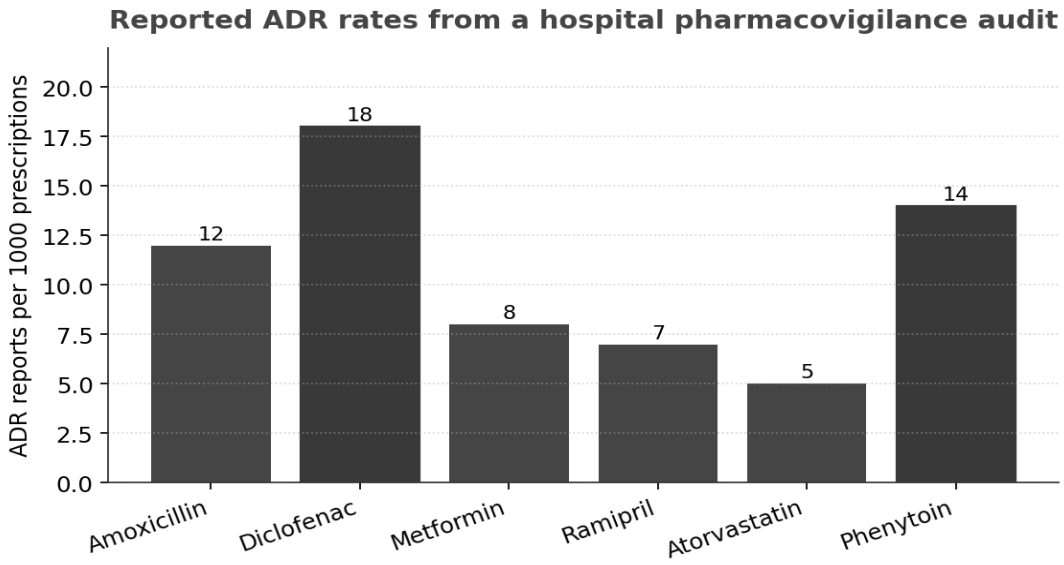

Figure 5.5 Reported ADR rates per 1000 prescriptions for six representative drugs in a hospital pharmacovigilance audit. Diclofenac and phenytoin (highlighted) stand out from the rest of the cohort.

5.6 Dissolution profiles

Dissolution testing is one of the everyday rituals of a formulation laboratory: at intervals after a tablet is dropped into a stirred vessel of buffer, an aliquot is removed and the dissolved drug is quantified by UV or HPLC. The result is a dissolution profile, the percentage of label-claim dissolved as a function of time. Three brands of paracetamol might be compared as follows:

```
t = np.array([0, 5, 10, 15, 20, 30, 45, 60])
brand_a = np.array([0, 32, 58, 76, 86, 94, 98, 100])
brand_b = np.array([0, 18, 34, 50, 64, 80, 90, 96])
brand_c = np.array([0,  8, 16, 28, 40, 60, 78, 88])

fig, ax = plt.subplots(figsize=(8, 4.4))
ax.plot(t, brand_a, "-o", label="Brand A (fast)",  color="#1F4E79")
ax.plot(t, brand_b, "-s", label="Brand B (medium)",color="#2E7D32")
ax.plot(t, brand_c, "-^", label="Brand C (slow)",  color="#C00000")
ax.axhline(80, linestyle="--", color="#555", alpha=0.7)
ax.text(2, 81.5, "USP/IP Q = 80 % in 30 min", color="#555")
ax.set_xlabel("Time (min)")
ax.set_ylabel("% drug dissolved")
ax.set_title("Dissolution profiles for three paracetamol brands")
ax.legend(frameon=False)
ax.grid(True, linestyle=":", alpha=0.6)
```

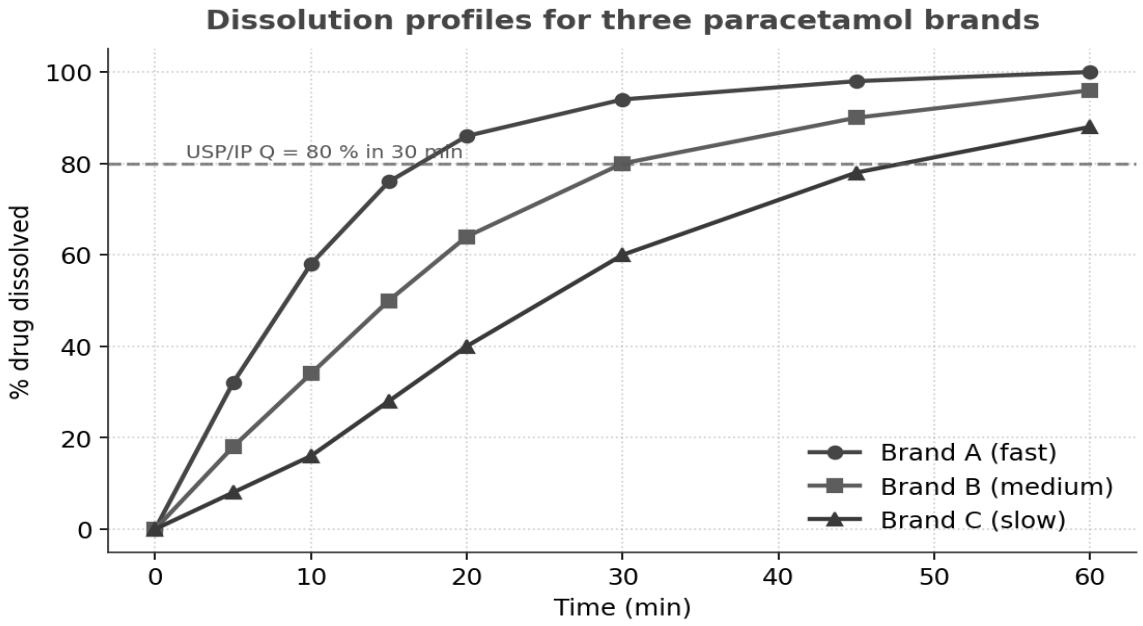

Figure 5.6 Dissolution profiles for three brands of paracetamol. The dashed horizontal line marks the pharmacopoeial acceptance criterion (Q = NLT 80 % in 30 min) for paracetamol IR tablets. Brand C does not meet that criterion at 30 min.

FROM THE BENCH

Why brand C fails. At 30 min, brand C has dissolved only 60 % of label claim, well below the Q value of 80 % required for paracetamol IR tablets in both the United States Pharmacopeia (USP <711>) and the Indian Pharmacopoeia. A formulation scientist would investigate disintegration time, particle size of the API, and the choice of disintegrant. Plots of this kind are decision-making documents in every formulation laboratory.

Note on Q values. The specific Q value depends on the monograph and the dosage form. For paracetamol IR tablets it is 80 %; other monographs specify different thresholds (often in the 70-85 % range). Always consult the current monograph before designing a dissolution test.

5.7 Subplots: many panels at once

Sometimes you want several plots side by side, say, a histogram next to a box plot of the same data, or four PK curves arranged in a 2 × 2 grid.

plt.subplots(rows, cols) returns the figure and a 2-D NumPy array of axes:

```
fig, axes = plt.subplots(1, 2, figsize=(11, 4))
axes[0].hist(ages, bins=18, color="#1F4E79", edgecolor="white")
axes[0].set_title("Distribution")
axes[0].set_xlabel("Age"); axes[0].set_ylabel("Count")
```

```
axes[1].boxplot(ages, vert=True, widths=0.4)
axes[1].set_title("Quartiles")
axes[1].set_ylabel("Age")

fig.suptitle("Two views of the same age column", fontsize=14)
fig.tight_layout()
```

fig.tight_layout() trims the white-space between subplots so the labels do not overlap. It is a small habit that makes the difference between an amateur figure and one that can sit in a printed paper.

5.8 Reading a plot scientifically

Drawing a plot is only half the job. The other half is reading what the plot is telling you and translating it back into pharmacological language. Three habits will serve you well.

5.8.1 Always look at the axes first

What units does the y-axis carry? Is the x-axis linear or logarithmic? Is the scale truncated, with a non-zero baseline that exaggerates the differences? Many misleading plots are honest in their numbers but dishonest in their axes. The first thing your eye should do is read the labels, not look at the curves.

5.8.2 Translate the geometry into chemistry

A line that is straight on a semilogarithmic plot of plasma concentration against time means first-order kinetics. A line that flattens at high concentrations on a Michaelis-Menten plot means saturation. A bar that is twice as tall as its neighbour means twice the count, twice the rate, twice the AUC, which in turn has a clinical consequence. The shape always tells a pharmacological story; train yourself to translate.

5.8.3 Beware of small samples

Eight data points in a scatter plot are not enough to detect a subtle linear relationship with confidence. The visual eye is excellent at finding patterns even where no patterns exist. When in doubt about a small dataset, follow the plot up with a statistical test. A plot suggests; a confidence interval confirms.

COMMON PITFALL

Bar charts that lie. If a bar chart's y-axis does not start at zero, even a tiny difference between bars looks huge. This is one of the most common visual deceptions in pharmaceutical marketing material, and you will see it again and again. When you plot a bar chart yourself, set the y-axis to start at zero unless you have a very good reason not to. When you read someone else's bar chart, check the axis. Always.

5.9 Worked examples

Example 5.1: A Cmax annotation

Plot an oral concentration-time curve and annotate the Tmax point with the value of Cmax.

```
ka, ke, F, D, V = 1.2, 0.18, 0.7, 200, 18
t = np.linspace(0, 24, 300)
C = (F * D * ka) / (V * (ka - ke)) * (np.exp(-ke * t) - np.exp(-ka * t))

tmax = t[np.argmax(C)]
cmax = C.max()

fig, ax = plt.subplots(figsize=(7.5, 4))
ax.plot(t, C, color="#1F4E79", linewidth=2)
ax.scatter([tmax], [cmax], color="#C00000", s=70, zorder=5)
ax.annotate(f"Cmax = {cmax:.2f} mg/L\nTmax = {tmax:.1f} h",
            xy=(tmax, cmax), xytext=(tmax + 3, cmax + 0.15),
            arrowprops=dict(arrowstyle="->"), color="#C00000")
ax.set_xlabel("Time (h)"); ax.set_ylabel("Concentration (mg/L)")
ax.set_title("Annotated oral PK curve")
```

Example 5.2: A two-panel ADR figure

Side by side, plot (a) the bar chart of ADR rates per drug and (b) the box plot of patient ages associated with each drug.

```
fig, axes = plt.subplots(1, 2, figsize=(11, 4))

axes[0].bar(drugs, rates, color="#1F4E79")
axes[0].set_title("ADR rate per 1000 prescriptions")
plt.setp(axes[0].get_xticklabels(), rotation=20, ha="right")

ages_by_drug = [ages_amox, ages_diclo, ages_metf, ages_rami, ages_atorv, ages_phen]
axes[1].boxplot(ages_by_drug, labels=drugs, patch_artist=True)
axes[1].set_title("Patient ages associated with each ADR")
plt.setp(axes[1].get_xticklabels(), rotation=20, ha="right")
fig.tight_layout()
```

Example 5.3: Plotting a DataFrame directly

Pandas DataFrames have a built-in .plot() method that wraps Matplotlib. Use it to plot mean AUC by drug as a bar chart.

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("statins_pk.csv")
summary = df.groupby("drug")["auc_ng_h_ml"].mean().sort_values()

ax = summary.plot(kind="bar", color="#1F4E79", figsize=(6, 4))
ax.set_ylabel("Mean AUC (ng·h/mL)")
ax.set_title("Mean AUC per drug")
plt.tight_layout()
```

5.10 Chapter summary

- Matplotlib is the foundational plotting library; the conventional alias is plt.
- A figure holds one or more axes; one axes is one plot.
- plt.subplots() returns (fig, ax).
- Use ax.plot for lines, ax.hist for histograms, ax.scatter for scatter plots, ax.boxplot for box plots, ax.bar for bars.
- Always label axes and add a title, a plot must stand on its own.
- PK curves: time on x, concentration on y. Semilog y exposes first-order kinetics as a straight line.
- Dissolution profiles: time on x, % dissolved on y. Compare to the USP Q value.
- Reading a plot is half the job; translate visual features back into pharmacological language.

5.11 Self-assessment: multiple-choice questions

Q1. Which alias is conventionally used for matplotlib.pyplot?

(a) plot
(b) plt
(c) mpl
(d) py

Q2. Which method creates a figure with one or more axes?

(a) plt.figure()
(b) plt.subplots()
(c) plt.axes()
(d) plt.create()

Q3. Which method is used for line plots?

(a) ax.bar()
(b) ax.line()
(c) ax.plot()
(d) ax.draw()

Q4. Which method draws a histogram?

(a) ax.histogram()
(b) ax.distplot()
(c) ax.hist()
(d) ax.bins()

Q5. What does ax.set_xlabel("Time") do?

(a) sets the figure title
(b) labels the x-axis
(c) labels the y-axis
(d) draws a vertical line

Q6. Which keyword adds an entry to the legend?

(a) title
(b) name
(c) label
(d) legend

Q7. Which method puts a logarithmic y-axis on the plot?

(a) ax.semilogy()
(b) ax.logy()
(c) ax.set_logscale_y()
(d) ax.exp()

Q8. In a box plot, the line in the middle of the box represents the:

(a) mean
(b) median
(c) mode
(d) minimum

Q9. What does ax.boxplot([g1, g2, g3], tick_labels=[…]) do?

(a) one box per group, labelled by category
(b) one stacked box
(c) draws three histograms
(d) raises an error

Q10. A scatter plot is the right choice when you want to show:

(a) category counts
(b) the relationship between two numeric variables
(c) the distribution of one variable
(d) a time-series curve

Q11. Which Matplotlib method saves a figure to disk?

(a) fig.write()
(b) fig.export()
(c) fig.savefig()

(d) plt.save()

Q12. Which keyword on plot() controls line style?

(a) dash
(b) linestyle
(c) stroke
(d) type

Q13. Which property of a bar chart can mislead the reader if not set carefully?

(a) the colour of the bars
(b) the order of categories
(c) the axis baseline (whether y starts at zero)
(d) the font size

Q14. What is the role of fig.tight_layout()?

(a) compresses the data
(b) trims white-space between subplots
(c) rotates the axis labels
(d) saves the figure

Q15. On a dissolution profile, the USP Q-value is shown as:

(a) a curve
(b) a horizontal threshold line
(c) a cluster of points
(d) a histogram

ANSWERS AND EXPLANATIONS

1. (b) plt a community convention.

2. (b) plt.subplots() returns (fig, ax).

3. (c) ax.plot() line plots.

4. (c) ax.hist() histogram method.

5. (b) labels the x-axis set_ylabel labels the y-axis.

6. (c) label used by ax.legend().

7. (a) ax.semilogy() a shortcut for log-scaled y.

8. (b) median the central line.

9. (a) one box per group, labelled by category the standard category comparison.

10. (b) the relationship between two numeric variables scatter is for X-Y relationships.

11. (c) fig.savefig() save to PNG/PDF/SVG.

12. (b) linestyle values: '-', '--', ':', '-.'

13. (c) the axis baseline a non-zero baseline can exaggerate differences.

14. (b) trims white-space between subplots prevents overlapping labels.

15. (b) a horizontal threshold line at the pharmacopoeial Q value (80 % at 30 min for paracetamol IR tablets).

5.12 Short-answer questions

1. Briefly describe the six-step recipe for drawing a Matplotlib plot.
2. State three reasons you would use a logarithmic y-axis when plotting plasma concentration against time.
3. Define and explain the use of axes, figure, and subplots in Matplotlib.
4. List four pieces of information that should accompany every published plot.
5. Why is a scatter plot superior to a line plot when you want to inspect a possible linear relationship between two measured variables?
6. Explain how a box plot conveys information about variability and outliers.
7. Sketch (in words) what the dissolution profile of a brand failing the USP Q test would look like.

5.13 Practical exercises

LAB 5.1: IV VS ORAL COMPARISON PLOT

Recreate the IV-vs-oral plot from §5.2.2 using parameters of your choice. Save the figure as iv_vs_oral.png at 200 DPI. Annotate Tmax of the oral curve.

LAB 5.2: ADR BAR CHART FROM A REAL CSV

Load any ADR CSV (you may use the one from Lab 4.2). Compute the count of reports per drug with Pandas, then plot the top 10 as a horizontal bar chart. Sort by count, descending.

LAB 5.3: BOX PLOT OF BIOCHEMISTRY BY TREATMENT ARM

Create a synthetic dataset of 60 patients in three treatment arms with simulated systolic BP readings. Use a box plot to compare the three arms. Add a meaningful title and y-axis label.

LAB 5.4: DISSOLUTION PROFILE WITH MARKERS

Plot the dissolution profile for two of your own paracetamol formulations (or use the values from §5.6). Use distinct markers for each brand, draw the pharmacopoeial Q line at 80 % (which is the requirement for paracetamol IR tablets in both USP and IP), and explain in two sentences below the figure whether either brand passes.

Sample sessional examination paper

BP101T · Basics of Python Programming for Pharmaceutical Sciences · Sessional examination · Time: 1 hour · Maximum marks: 20

INSTRUCTIONS

Answer all questions. Section A carries 5 marks (one mark each); Section B carries 6 marks (two marks each); Section C carries 9 marks (three marks each). Use of a non-programmable calculator is permitted. Pseudocode answers are acceptable where the question does not specifically ask for Python syntax.

Section A · Multiple-choice questions (5 × 1 mark)

Q1. Which of the following is NOT a primitive Python data type?

(a) int
(b) float
(c) tablet
(d) bool

Q2. Which keyword exits a loop completely?

(a) break
(b) continue
(c) stop
(d) exit

Q3. What is the type of the result of `3 / 1` ?

(a) int
(b) float
(c) str
(d) bool

Q4. Which library is conventionally aliased as `pd` ?

(a) NumPy
(b) Pandas
(c) Matplotlib
(d) Scipy

Q5. Which Matplotlib method draws a histogram?

(a) `ax.hist()`
(b) `ax.bar()`
(c) `ax.scatter()`
(d) `ax.plot()`

Section B · Short-answer questions (3 × 2 marks)

1. Distinguish between an integer (int) and a float in Python with one example each.
2. Write a Python function calc_dose(weight_kg, mg_per_kg=10) that returns the total dose. Show one call that uses the default and one that overrides it.
3. State three string methods you would use while cleaning a column of patient names. Briefly say what each does.

Section C · Long-answer questions (3 × 3 marks)

1. Write a Python program that reads height (m) and weight (kg) from the user, computes BMI, classifies the patient using WHO categories, and prints a clean message.
2. A list doses_mg = [500, 400, 300, 50, 250] contains the daily dose for five patients. Using a for loop, print the dose, the dose if halved (rounded to one decimal), and 'HIGH' for any dose above 350.
3. Explain, with a small Python example, the difference between .iloc and .loc in Pandas.

Sample end-semester examination paper

BP101T · Basics of Python Programming for Pharmaceutical Sciences · End-semester examination ·

Time: 2 hours · Maximum marks: 30

INSTRUCTIONS
Answer all questions. Section A is multiple-choice (10 × 1 = 10 marks). Section B is short-answer (4 × 2 = 8 marks). Section C is long-answer (3 × 4 = 12 marks). Where a question asks for code, full marks are awarded only for code that would actually run; deduce one mark for each obvious syntax error.

Section A · Multiple-choice questions (10 × 1 mark)

Q1. Which of these is the correct way to start a for loop over a list?

(a) for x on list:
(b) for x in list:
(c) foreach x in list:
(d) loop x in list:

Q2. Which Python type is mutable?

(a) str
(b) tuple
(c) list
(d) int

Q3. What does df.head() return by default?

(a) the first row
(b) the first 5 rows
(c) the column names
(d) the row count

Q4. Which method joins two DataFrames on a key column?

(a) concat
(b) merge
(c) append
(d) combine

Q5. What does (60 // 7) evaluate to?

(a) 7
(b) 8
(c) 8.57
(d) 60.0

Q6. Which Matplotlib method draws a scatter plot?

(a) ax.line()
(b) ax.dot()
(c) ax.scatter()
(d) ax.plot('o')

Q7. Which keyword in pd.read_csv parses dates as datetime?

(a) parse_dates
(b) dt
(c) convert_dates
(d) datetime=True

Q8. What does the operator ** do?

(a) multiplication
(b) exponentiation
(c) comment
(d) comparison

Q9. Which function counts how often each item appears in an iterable?

(a) set()
(b) Counter()
(c) frequency()
(d) len()

Q10. Which keyword skips to the next iteration of a loop?

(a) pass
(b) skip
(c) continue
(d) break

Section B · Short-answer questions (4 × 2 marks)

1. Define an Integrated Development Environment (IDE) and name two IDEs commonly used for Python.
2. Differentiate between a list and a tuple in Python with one pharmacy-flavoured example of each.
3. Briefly describe the split-apply-combine paradigm in Pandas, with a one-line example.
4. Why should the y-axis baseline of a bar chart usually start at zero?

Section C · Long-answer questions (3 × 4 marks)

1. Write a Python function adjust_dose(crcl, normal_dose) that returns the adjusted dose for a patient: full dose if CrCl ≥ 30 mL/min, half dose if 10 ≤ CrCl < 30, contraindicated otherwise. Demonstrate the function with three calls.
2. Given a CSV file pk.csv with columns drug, dose_mg and auc_ng_h_ml, write Python code (using Pandas) that prints the count of patients per drug and the mean AUC per drug. Indicate clearly which line does what.

3. Plot a concentration-time curve after an oral dose, modelled by C(t) = (F·D·ka) / (V·(ka−ke)) · (e^(−ke·t) − e^(−ka·t)) for parameters of your choosing. Annotate Tmax on the figure. Briefly interpret the shape of the curve in pharmacokinetic terms.

Glossary

A short, in-context glossary of the most important programming and pharmacy terms used in this book.

Term	Meaning
argument	A value passed in to a function when it is called. Compare ‘parameter’.
array	A contiguous, typed collection of values. In Python the typical array type is the NumPy ndarray.
AUC	Area Under the Curve; in pharmacokinetics, the integral of plasma concentration over time.
bool	A primitive Python type with only two values: True and False.
boolean mask	A boolean array used to select rows of a DataFrame, True keeps, False drops.
broadcasting	NumPy/Pandas mechanism that lets arrays of different (but compatible) shapes be combined element-wise.
Cmax	Peak plasma concentration of a drug after administration.
CSV	Comma-Separated Values; a plain-text tabular file format.
DataFrame	Pandas’ central data container: a labelled, two-dimensional table.
dictionary (dict)	A Python container of key-value pairs, written in curly braces.
dissolution profile	A plot of percentage of drug dissolved against time, generated by an in-vitro dissolution test.
dose ceiling	An upper limit applied to a calculated dose to avoid toxicity.
docstring	A documentation string written as the first statement of a function. Helpful for users; printed by help().
element-wise	Applied to each item separately; characteristic of NumPy and Pandas arithmetic.
float	A Python type that represents decimal numbers using IEEE 754 binary.
function	A reusable bundle of code with a name, optional parameters and a return value.
groupby	Pandas operation that splits a DataFrame by the unique values of one or more columns, applies a function to each group, then combines.
IDE	Integrated Development Environment: an editor plus interpreter, debugger and helpers.

Term	Meaning
import	Python statement that brings the contents of a library into the current program.
int	A Python type for whole numbers, of arbitrary size.
JSON	JavaScript Object Notation: a lightweight text data-exchange format that maps cleanly onto Python dict / list / str / number.
Jupyter Notebook	A browser-based, cell-by-cell IDE popular with data scientists.
k_e	First-order elimination rate constant in pharmacokinetics, units of inverse time.
list	An ordered, mutable Python container of values.
list comprehension	A compact one-line syntax to build a new list by transforming or filtering an existing iterable.
Matplotlib	The foundational scientific plotting library in Python.
method	A function attached to a value: object.method() syntax.
module	A single Python file that can be imported by another.
mutable	A property of a value that can be changed in place. Lists and dictionaries are mutable; strings, tuples, ints, floats are not.
NaN	Not a Number; Pandas' standard sentinel for missing values.
NumPy	Numerical Python; the foundational array library on which most scientific Python sits.
operator	A symbol that represents a built-in operation: +, -, *, /, ==, and so on.
package	A bundle of Python modules distributed together (eg. Pandas, NumPy).
Pandas	The library on top of NumPy for tabular data analysis.
parameter	A name in the function definition. Compare 'argument'.
pip	The standard Python installer for third-party packages.
PyCharm	A heavyweight Python IDE made by JetBrains.
return value	The value handed back by a function to its caller.
script	A Python source file run from start to finish.
Series	Pandas' one-dimensional labelled array; one column of a DataFrame.
slice	Notation [a:b] (or [a:b:s]) that selects part of a sequence.

Term	Meaning
string (str)	A sequence of characters; immutable in Python.
Tmax	The time at which Cmax is reached after dosing.
truthy / falsy	Adjectives describing how a value is interpreted in a boolean context. 0, 0.0, '', [], (), {}, None are falsy; everything else is truthy.
tuple	An immutable ordered Python container, written in parentheses.
vectorisation	Performing an operation on a whole array at once instead of element-by-element with a loop.
VS Code	Visual Studio Code; a lightweight editor that becomes a Python IDE with the Python extension.
with-statement	A Python construct that guarantees a resource (eg. an open file) is cleaned up when the block ends.

References and further reading

Books

- **Weiss, C. J.** *(2017). Scientific Computing for Chemists with Python.* Available at: weisscharlesj.github.io/SciCompforChemists. An open-access set of Jupyter notebooks aimed at chemistry students. The chapters on data manipulation and plotting transfer almost word-for-word to pharmacy and are an excellent companion to Units 4 and 5 of this book.
- **Perkovic, L.** *(2015). Introduction to Computing Using Python: An Application Development Focus,* 2nd edition. Hoboken: Wiley. A traditional introductory text that complements Units 1 and 2 of this book.
- **Sweigart, A.** *(2025). Automate the Boring Stuff with Python,* 3rd edition. Available at: automatetheboringstuff.com. Free under a Creative Commons licence. Especially useful for everyday automations (renaming files, sending emails, scheduled scripts) that complement Unit 3.
- **McKinney, W.** *(2022). Python for Data Analysis,* 3rd edition. O'Reilly. The definitive Pandas book, written by the library's creator. Recommended for deeper study after Unit 4.
- **VanderPlas, J.** *(2023). Python Data Science Handbook,* 2nd edition. O'Reilly. Free online at jakevdp.github.io/PythonDataScienceHandbook. An excellent companion to Units 4 and 5.

Online tutorials

- **W3Schools Python Tutorial.** w3schools.com/python. Bite-sized examples for revising syntax.
- **Real Python.** realpython.com. Higher-quality long-form tutorials, with many examples on Pandas and Matplotlib.
- **Pandas official documentation.** pandas.pydata.org. The user guide is well-organised; the cookbook is full of practical recipes.

Healthcare datasets for practice

- **Kaggle.** kaggle.com: search for 'healthcare', 'pharmacy', 'ADR'.
- **WHO Global Health Observatory.** who.int/data/gho: country-level health statistics including medicines.
- **data.gov.in.** Indian government open-data portal.
- **ClinicalTrials.gov** and **Clinical Trials Registry of India (CTRI).** clinicaltrials.gov, ctri.nic.in: registered trials and their protocols.
- **PvPI (Pharmacovigilance Programme of India).** ipc.gov.in/pvpi: ADR reports compiled across Indian hospitals.

Pharmacopoeias and clinical references

- *Indian Pharmacopoeia,* latest edition. The official pharmacopoeia is the authoritative reference for tests including dissolution. Whenever the program in this book mentions a USP Q-value, the equivalent IP specification should be checked for Indian practice.
- *British National Formulary,* latest edition. A reliable source of dosage information used in many of the examples in Units 1 and 2.