B.PHARM · FIRST SEMESTER · BP101T

Python made simple for pharmacy students

A from-zero teaching guide with computer basics, defined terminology, line-by-line explanations, worked pharmaceutical examples, worksheets, practical tasks, and answer keys for Units I-V.

Syllabus mapped | No prior coding needed | Code + output | Practice included

Pills and Python
Instagram: @pillsandpython
Website: pillsandpython.com
YouTube: @PillsAndPython

Course map

Course pattern: 2 theory hours per week, 2 credits, 30 total hours. Each of the five units is planned for 6 hours.

Unit	Main question	What you will be able to do
I. Python basics	How does a program store and process values?	Install tools; use variables, types, operators, input, output, strings, and libraries.
II. Logic + functions	How can a program decide, repeat, and reuse?	Write conditions, loops, and functions; build dose and body mass index programs.
III. Data + files	How can many records be organized?	Use lists, tuples, dictionaries, NumPy arrays, and CSV files.
IV. Pandas	How can a dataset be cleaned and summarized?	Load, inspect, clean, filter, group, and aggregate tabular data.
V. Visualization	How can patterns be communicated?	Create and interpret line, histogram, scatter, and box plots.

Course outcomes in plain language

Explain
Describe Python fundamentals and why libraries are useful.

Build logic
Use control structures and functions to solve a defined problem.

Handle data
Store, retrieve, and manipulate structured healthcare data.

Analyze + visualize
Use NumPy, Pandas, and Matplotlib to explore data responsibly.

Scope note: These notes follow the supplied BP101T syllabus. Practice datasets are clearly marked as illustrative and are not clinical evidence.

How to study and run the code

1. Read → 2. Predict → 3. Run → 4. Explain → 5. Change

Choose a coding workspace

Tool	Best beginner use	Why it helps
Jupyter Notebook	Short examples and plots	Run one cell at a time; keep notes beside code.
PyCharm	Larger programs	Project tools and debugger in one place.
VS Code	Lightweight coding	Extensions, terminal, and files in one window.

First run

```
print("Pills and Python: ready to learn")
```

```
Pills and Python: ready to learn
```

IDE versus text editor

An integrated development environment (IDE) can run, debug, and manage code. A plain text editor mainly edits text.

Good practical habit

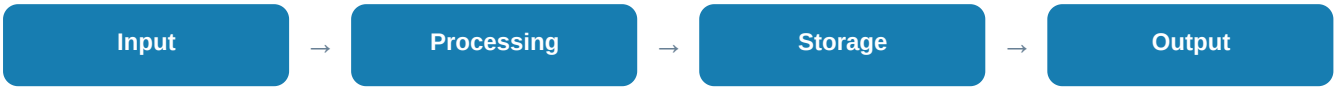
Create one folder for the course. Keep each unit in a separate notebook or `.py` file, and save sample CSV files beside it.

When an error appears

1. Read the last line of the message.
2. Find the line number.
3. Check spelling, brackets, quotes, indentation, and data type.
4. Change one thing, then run again.

Before Python: know your computer

A **computer** is an electronic machine that accepts input, processes it according to instructions, stores data, and produces output. Programming means writing those instructions in a language the computer can follow.



Four words you must know

Hardware
The physical parts you can touch: keyboard, mouse, monitor, processor, and storage drive.

Software
Instructions and programs that run on hardware: Windows, a browser, Python, or Jupyter Notebook.

Data
Facts supplied to a computer, such as a medicine name, strength, temperature, or stock quantity.

Program
An ordered set of instructions written to complete a task.

Pharmacy example

You type stock quantities with the keyboard (input). The program compares each quantity with its reorder level (processing), may save the records in a file (storage), and displays a low-stock list (output).

Important: A computer does not understand the intention behind vague instructions. It follows the written instruction exactly. Good programs therefore use clear steps and valid data.

Files, folders, and the screen

A **file** is a named collection of stored information. A **folder** is a container used to organize files. A **file extension** is the ending after a dot that suggests a file's format.

Example	Meaning	Typical use
`dose.py`	Python source file	Stores program instructions
`stock.csv`	Comma-separated values file	Stores a simple table
`report.xlsx`	Excel workbook	Stores spreadsheet data
`notes.ipynb`	Jupyter notebook	Stores code, output, and explanations

Basic actions

- **Click:** press and release the left mouse button once.
- **Double-click:** click twice quickly to open an item.
- **Right-click:** open a menu of actions for an item.
- **Cursor:** the blinking mark that shows where typed text will appear.
- **Window:** the rectangular area in which an application appears.
- **Save:** write current changes to storage. Use `Ctrl+S` often.

A useful course folder

```
BP101T_Python/
    Unit_1/
    Unit_2/
    practice_data/
    charts/
```

Filename rule: Use meaningful names without unusual symbols: `unit1_variables.py` is clearer than `new final 2.py`.

What coding words mean

Code is text written in a programming language. **Python** is a high-level, general-purpose programming language designed to be readable. A **programming language** supplies vocabulary and grammar for giving a computer instructions.

Translation from code to action

Term	Beginner definition
Source code	The Python instructions written by a programmer.
Syntax	The grammar rules for writing valid code.
Interpreter	Software that reads and executes Python code.
Statement	A complete instruction, such as an assignment or print call.
Expression	Code that produces a value, such as `2 * 500`.
Algorithm	A finite, ordered method for solving a problem.
Bug	An error or unwanted behavior in a program.
Debugging	Finding, understanding, and correcting a bug.

First algorithm: total tablets

1. Read tablets per dose.
2. Read doses per day.
3. Read number of days.
4. Multiply the three values.
5. Display the total with a unit.

Key distinction: An algorithm is the plan; Python code is one way to express that plan.

Install and open a Python workspace

Option A: Jupyter Notebook

1. Install a current Python distribution or use a computer on which Python and Jupyter are already installed.
2. Open a terminal: Command Prompt or PowerShell on Windows, or Terminal on macOS/Linux.
3. Type `python --version` (sometimes `python3 --version`) and press Enter.
4. Start Jupyter using the installed launcher or the command `jupyter notebook`.
5. Create a new Python notebook. A rectangular **cell** accepts code.
6. Type `print("Hello")`. Run the cell with the Run button or `Shift+Enter`.

Option B: a `.py` file

1. Open an integrated development environment (IDE) such as PyCharm or Visual Studio Code.
2. Create a file named `first_program.py`.
3. Type the code, save the file, and select Run.

```
print("Hello, future pharmacist programmer!")
```

```
Hello, future pharmacist programmer!
```

If nothing runs

- Confirm Python is installed by checking its version.
- Confirm you selected a Python interpreter in the IDE.
- Save the file before running it.
- Copy the final error line into your study log and explain what you tried.

Terminal versus Python: Installation commands such as `python -m pip install pandas` belong in the terminal. Python statements such as `import pandas` belong in a notebook cell or `.py` file.

Read one line of Python

Read code from the inside outward. In the line below, Python first receives text from the keyboard, then converts that text to a decimal number, and finally stores the number under a name.

```
weight_kg = float(input("Enter weight in kg: "))
```

Part	What it means
`"Enter weight in kg: "`	A string shown as a prompt.
`input(...)`	Waits for keyboard input and returns text.
`float(...)`	Converts suitable text to a decimal number.
`=`	Assignment: stores the value on the right.
`weight_kg`	The variable name used later.

Punctuation has a job

- **Parentheses `( )`:** call a function or control calculation order.
- **Quotes `" "` or `' '`:** mark text.
- **Colon `:`:** begins an indented block after `if`, `for`, `while`, or `def`.
- **Comma `,`:** separates items or function arguments.
- **Hash `#`:** begins a comment ignored by Python.
- **Indentation:** spaces at the start of a line that show which statements belong together.

Reading habit: For every line, ask: What values enter? What operation happens? Where is the result stored or shown?

Variables: labels for values

A **variable** is a meaningful name that refers to a value held while a program runs. Think of it as a labeled container: the label is the variable name and the content is the value.

```
drug_name = "Paracetamol"
strength_mg = 500
strength_mg = 650
```

The first statement creates the name `drug_name`. The second makes `strength_mg` refer to 500. The third **reassigns** that same name to 650; it does not test equality.

Vocabulary

Term	Definition
Identifier	A name used for a variable, function, or other program object.
Assignment	Binding a name to a value using =.
Value	The stored information, such as `500` or `"oral"`.
Literal	A value written directly in code, such as `10`.
Reassignment	Giving an existing variable a new value.

Why names matter

Weak

```
x = 500
y = 3
```

The reader must guess what the numbers mean.

Clear

```
strength_mg = 500
doses_per_day = 3
```

Names carry meaning and units.

Do not confuse: = assigns a value. == asks whether two values are equal.

Data types: what kind of value is this?

A **data type** is a category that tells Python what a value represents and which operations are valid. The characters `500` can represent a number, while `"500"` represents text because it is inside quotes.

Type	Definition	Pharmacy example
`int`	Integer: whole number without a decimal part.	Number of tablets: `20`
`float`	Floating-point number: numerical value with decimal representation.	Volume: `5.5` mL
`str`	String: an ordered sequence of text characters.	Route: `"oral"`
`bool`	Boolean: logical value `True` or `False`.	`prescription_valid = True`
`NoneType`	Type of `None`, meaning no value is present.	Unknown result not yet recorded

Inspect values

```
strength = 500
print(type(strength))
print(isinstance(strength, int))
```

```
<class 'int'>
True
```

Why types matter

`500 + 50` performs arithmetic and returns `550`. `"500" + "50"` joins two strings and returns `"50050"`. Python is behaving correctly in both cases because the types are different.

Precision note: A `float` is an approximate binary representation. Do not treat ordinary beginner code as validated software for regulated calculations.

Operators: symbols that perform work

An **operator** is a symbol or keyword that acts on one or more values. The values on which it acts are called **operands**. In `3 * 500`, `*` is the operator and 3 and 500 are operands.

Operator	Name	Example result
`+`	Addition	`10 + 5` is 15
`-`	Subtraction	`10 - 5` is 5
`*`	Multiplication	`10 * 5` is 50
`/`	True division	`10 / 4` is 2.5
`//`	Floor division	`10 // 4` is 2
`%`	Remainder	`10 % 4` is 2
`**`	Exponentiation	`2 ** 3` is 8

Three operator families

Arithmetic operators calculate numbers. **Comparison operators** such as > and == produce `True` or `False`. **Logical operators** `and`, `or`, and `not` combine or reverse Boolean conditions.

```
has_prescription = True
stock = 12
print(has_prescription and stock > 0)
```

```
True
```

Use parentheses: `daily_mg = (mg_per_kg * weight_kg) * doses_per_day` makes the intended order easier to inspect.

Input, output, and conversion

Input is information supplied to a program. **Output** is information produced by a program. In basic Python, `input()` reads keyboard text and `print()` displays output.

```
name = input("Medicine name: ")
quantity_text = input("Quantity: ")
quantity = int(quantity_text)
print("Medicine:", name)
print("Quantity:", quantity)
```

Why conversion is needed

`input()` always returns a string, even if the user types digits. **Type casting** means explicitly converting a value to another type.

Function	Conversion	Example
`int(x)`	Suitable value to integer	`int("20")` → 20
`float(x)`	Suitable value to decimal number	`float("5.5")` → 5.5
`str(x)`	Value to text	`str(500)` → `"500"`
`bool(x)`	Value to Boolean	Requires careful interpretation

Formatted output

An **f-string** begins with `f` and places expressions inside braces.

```
price = 12.5
print(f"Price: ₹{price:.2f}")
```

```
Price: ₹12.50
```

Input validation: `int("five")` raises `ValueError`. A robust program checks or catches invalid input instead of assuming every entry is correct.

Strings: text as a sequence

A **string** is an ordered sequence of characters enclosed in quotes. Characters may be letters, digits, spaces, or symbols. Strings are **immutable**: an operation creates a new string rather than altering the original string object.

```
label = "  paracetamol 500 mg  "
clean_label = label.strip().title()
print(clean_label)
```

```
Paracetamol 500 Mg
```

Parts of a string

- **Index:** a character's position. Python starts at 0; negative indexes count from the end.
- **Slice:** a selected section written as `text[start:stop:step]`. The stop position is excluded.
- **Method:** a function attached to an object, called with dot notation such as `label.strip()`.
- **Concatenation:** joining strings with +.

Code	Meaning
`label[0]`	First character
`label[-1]`	Last character
`label[2:6]`	Characters at indexes 2 to 5
`"mg" in label`	Membership test
`label.lower()`	Create lowercase copy

Common surprise: `len(label)` counts spaces and symbols as characters.

Variables and data types

A **variable** is a name that refers to a value. The equals sign assigns the value on its right to the name on its left.

```
drug_name = "Paracetamol"   # string
strength_mg = 500          # integer
price = 2.75               # float
in_stock = True            # boolean

print(drug_name, strength_mg, price, in_stock)
```

```
Paracetamol 500 2.75 True
```

Type	Stores	Example
`int`	Whole number	`tablets = 20`
`float`	Decimal number	`volume_ml = 5.5`
`str`	Text	`route = "oral"`
`bool`	True or False	`sterile = True`

Naming rules

- Begin with a letter or underscore, not a number.
- Use letters, digits, and underscores; spaces are not allowed.
- Names are case-sensitive: `dose` and `Dose` are different.
- Prefer meaningful snake_case names such as `daily_dose_mg`.

Check a type: `type(strength_mg)` returns `<class 'int'>`.

Quick check

What types are `"250"`, `250`, `250.0`, and `False`? Answer: `str`, `int`, `float`, and `bool`.

Type casting and operators

Type casting converts a value from one type to another. This matters because `input()` always returns text.

```
strength_text = "250"
strength_mg = int(strength_text)
new_strength = strength_mg + 50
print(new_strength)
```

```
300
```

Group	Operators	Example
Arithmetic	`+ - * / // % **`	`total = 3 * 500`
Comparison	`== != > < >= <=`	`stock <= reorder`
Logical	`and or not`	`adult and weight > 40`

Division

`7 / 2` gives `3.5`.
`7 // 2` gives `3`.
`7 % 2` gives remainder `1`.

Order of operations

Parentheses first, then powers, multiplication/division, and addition/subtraction.

```
dose = (weight * 10) / 2
```

Worked example: cost per strip

```
box_price = 240.0
strips_in_box = 12
cost_per_strip = box_price / strips_in_box
print(f"₹{cost_per_strip:.2f}")
```

```
₹20.00
```

Common mistake: `"250" + "50"` gives `"25050"`, not 300. Convert the text first.

Input, output, and strings

`input()` reads text from the keyboard. `print()` displays a result. An f-string inserts values inside braces.

```
patient = input("Patient name: ")
weight = float(input("Weight in kg: "))
print(f"{patient} weighs {weight:.1f} kg")
```

Useful string operations

Operation	Code	Result
Length	`len("AMOX500")`	`7`
Uppercase	`"oral".upper()`	`"ORAL"`
Remove spaces	`" A102 ".strip()`	`"A102"`
Replace	`"5 mg".replace("mg","mL")`	`"5 mL"`
Split	`"A,500".split(",")`	`["A", "500"]`

Indexing and slicing

Python starts counting at 0. For `batch = "BATCH-A1024"`:

`batch[0]`
`'B'`

`batch[-1]`
`'4'`

`batch[6:]`
`'A1024'`

Slice rule: `text[start:stop]` includes `start` but excludes `stop`.

Errors are messages, not failure

An **error** is a condition that prevents correct execution or produces an incorrect result. A **traceback** is Python's report showing where an exception occurred. Read its final line first, then inspect the referenced line of code.

Error	Meaning	First check
`SyntaxError`	Code violates Python grammar.	Quotes, brackets, colon
`IndentationError`	Block spacing is invalid.	Consistent spaces
`NameError`	Name has not been defined.	Spelling and assignment order
`TypeError`	Operation is unsuitable for a value's type.	Use `type()`
`ValueError`	Type is acceptable but the value is invalid.	Input content
`IndexError`	Sequence position does not exist.	Length and index range
`KeyError`	Dictionary or DataFrame label is absent.	Exact key/column spelling
`FileNotFoundError`	Requested path cannot be found.	Folder, filename, extension

Debugging cycle



Learning habit: Keep an error log with four columns: error message, cause, correction, and one example. Errors become a revision resource.

Libraries, packages, and common errors

A **library** is reusable code. Python's standard library is installed with Python; third-party packages are installed separately.

Standard library

```
import math
print(math.sqrt(81))
```

```
9.0
```

Third-party package

```
python -m pip install pandas
python -m pip uninstall pandas
```

Run these commands in a terminal, not inside ordinary Python code.

Error	Meaning	Typical fix
`SyntaxError`	Python cannot understand the written structure.	Check quotes, brackets, colon.
`NameError`	A name was used before it was defined.	Check spelling and assignment.
`TypeError`	An operation received the wrong kind of value.	Inspect types; cast if appropriate.
`ValueError`	The type is possible, but the value cannot be converted.	Validate the input.

Debug this

```
age = input("Age: ")
next_year = age + 1
```

This raises a `TypeError` because `age` is text. Fix it with `age = int(input("Age: "))`.

Safe learning rule: Code examples are for education. A dose used in practice must come from an authorized prescription or approved clinical protocol, not from a learning program alone.

Worked examples: from values to results

Example 1: total tablets

```
tablets_each_dose = 2
doses_per_day = 3
days = 5
total = tablets_each_dose * doses_per_day * days
print("Total tablets:", total)
```

```
Total tablets: 30
```

Example 2: temperature conversion

```
celsius = 37.0
fahrenheit = (celsius * 9/5) + 32
print(f"{celsius}°C = {fahrenheit:.1f}°F")
```

```
37.0°C = 98.6°F
```

Example 3: parse a medicine label

```
label = " amoxicillin 500 mg "
clean = label.strip().title()
print(clean)
print("Contains mg:", "mg" in label.lower())
```

```
Amoxicillin 500 Mg
Contains mg: True
```

Read the code
Identify each input, operation, and output. Predict the output before running.

Change the code
Try 1 tablet, 4 doses per day, and 7 days. Then change the temperature to 38.5°C.

Worksheet 1: Python foundations

A. Recall

1. Define variable, data type, type casting, and library.

2. Name four basic Python data types with one pharmacy example each.

3. State two advantages of an IDE over a plain text editor.

B. Predict the output

```
a = 5
b = 2
print(a / b)
print(a // b)
print(a % b)
```

4. Write all three outputs before running the code.

C. Debug

```
drug = Paracetamol
print("Drug:" drug)
```

5. Find and correct both errors.

D. Write programs

6. Ask for a patient's name and age, then print a complete sentence.

7. Input unit price and quantity; display total cost to two decimal places.

8. Store a batch code and print only its final four characters.

Practical task

Create `unit1_label.py`. Ask for drug name, strength, dosage form, and batch number. Clean extra spaces, standardize the drug name with `title()`, and print a neat four-line label.

Self-check: Can you explain every symbol in your program without reading from the screen?

Conditions and Boolean thinking

A **condition** is an expression evaluated as `True` or `False`. A **selection statement** lets a program choose which block to execute.

Keyword	Definition
`if`	Tests the first condition and runs its block when true.
`elif`	Tests another condition only if earlier branches were false.
`else`	Runs when no preceding condition was true.
Branch	One possible path through a decision.
Block	Indented group of statements controlled together.

```
stock = 7
reorder_level = 10

if stock == 0:
    print("Out of stock")
elif stock <= reorder_level:
    print("Reorder")
else:
    print("Stock adequate")
```

```
Reorder
```

Trace it

1. `stock == 0` is false, so its block is skipped.
2. `stock <= reorder_level` means `7 <= 10`, which is true.
3. Python prints `Reorder` and skips the final `else`.

Order matters: In an `if/elif/else` chain, Python executes the first true branch only. Put specific or high-priority tests in a sensible order.

Conditions make decisions

A condition is an expression that becomes `True` or `False`. Python runs only the indented block belonging to the selected branch.

```
temperature = 8

if temperature > 8:
    print("Check storage condition")
elif temperature < 2:
    print("Too cold")
else:
    print("Within 2-8°C range")
```

```
Within 2-8°C range
```

Pattern	Use
`if`	Run something only when a condition is true.
`if ... else`	Choose between two paths.
`if ... elif ... else`	Choose among several ordered paths.
Nested `if`	Check a second condition inside the first.

Nested example

```
prescription_present = True
age = 12
if prescription_present:
    if age < 18:
        print("Verify pediatric instructions")
    else:
        print("Continue standard verification")
else:
    print("Prescription required")
```

Reasoning tip: Write the decision as a sentence first. Then convert each part into a comparison or logical expression.

Loops: repeat without copying code

A **loop** repeats a block of code. One pass through the block is an **iteration**. The variable that changes as a loop progresses is often called a **loop variable** or counter.

For loop

A `for` loop visits each item in an **iterable**, an object that can supply items one at a time.

```
drugs = ["ORS", "Zinc", "Paracetamol"]
for drug in drugs:
    print(drug)
```

The loop variable `drug` refers to one list item during each iteration.

While loop

A `while` loop repeats while its condition is true.

```
attempts = 3
while attempts > 0:
    print("Attempts left:", attempts)
    attempts = attempts - 1
```

Loop control

Keyword	Effect
`break`	Stops the nearest loop immediately.
`continue`	Skips the remaining statements in this iteration.
`pass`	Does nothing; acts as a temporary placeholder.

Infinite loop: If a `while` condition never becomes false, the loop continues indefinitely. Ensure something in the loop changes the condition.

Loops repeat work

A `for` loop repeats over a sequence. A `while` loop repeats while a condition remains true.

For loop

```
drugs = ["A", "B", "C"]
for drug in drugs:
    print(drug)
```

Use when the items or number of repetitions are known.

While loop

```
count = 3
while count > 0:
    print(count)
    count -= 1
```

Use when repetition depends on a changing condition.

`range()` patterns

Code	Values produced
`range(5)`	0, 1, 2, 3, 4
`range(1, 5)`	1, 2, 3, 4
`range(2, 11, 2)`	2, 4, 6, 8, 10

Daily dose table

```
single_dose_mg = 250
for dose_no in range(1, 4):
    print(f"Dose {dose_no}: {single_dose_mg} mg")
```

```
Dose 1: 250 mg
Dose 2: 250 mg
Dose 3: 250 mg
```

Avoid infinite loops: A `while` loop must eventually change the value tested in its condition.

Break, continue, and loop tracing

`break` stops the nearest loop. `continue` skips the rest of the current repetition and moves to the next one.

```
stock_values = [12, 8, 0, 15]
for stock in stock_values:
    if stock == 0:
        print("Out of stock found")
        break
    print("Checked:", stock)
```

```
Checked: 12
Checked: 8
Out of stock found
```

```
readings = [98, None, 102, 100]
for value in readings:
    if value is None:
        continue
    print("Valid reading:", value)
```

```
Valid reading: 98
Valid reading: 102
Valid reading: 100
```

Trace table method

Step	Current value	Condition	Action
1	98	`value is None` → False	Print 98
2	None	True	Continue
3	102	False	Print 102

Choose carefully: Use `break` when further searching is unnecessary. Use `continue` when one record should be skipped but later records still matter.

Functions: name a reusable procedure

A **function** is a named, reusable block of code that performs a defined task. Functions reduce repetition, separate responsibilities, and make testing easier.

```
def total_quantity(units_per_dose, doses_per_day, days):
    total = units_per_dose * doses_per_day * days
    return total

result = total_quantity(2, 3, 5)
print(result)
```

```
30
```

Term	Definition
`def`	Keyword that begins a function definition.
Function name	Identifier used to call the function.
Parameter	Input name in the function definition.
Argument	Actual value supplied in a function call.
`return`	Sends a result back to the caller and ends the function.
Local variable	Name created inside the function and normally unavailable outside it.

Call versus print

Calling the function performs its instructions. `return` gives a value back so it can be stored or reused. `print()` only displays a value; it is not a replacement for returning a result.

Design rule: A short function should do one clearly named job. A name such as `calculate_total_tablets` explains more than `do_it`.

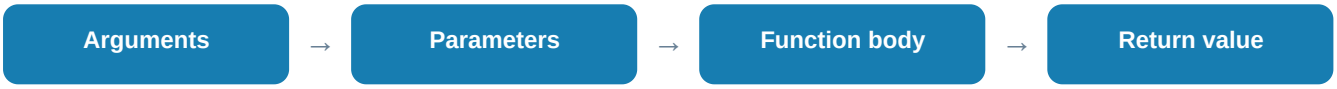
Functions make programs reusable

A **function** is a named block of code that performs one job. Define it with `def`, call it by name, pass inputs as arguments, and send a result back with `return`.

```
def total_quantity(units_per_dose, doses_per_day, days):
    total = units_per_dose * doses_per_day * days
    return total

needed = total_quantity(2, 3, 5)
print(needed)
```

```
30
```



Term	Meaning
Parameter	Name inside the function definition, such as `days`.
Argument	Actual value supplied in a call, such as `5`.
Return value	Result sent back to the caller.
Local variable	Name created inside a function; normally used there only.

Default argument

```
def label(drug, route="oral"):
    return f"{drug} - {route}"

print(label("Cetirizine"))
```

```
Cetirizine - oral
```

Worked applications: dosage math and BMI

Weight-based calculation

The formula below is an arithmetic demonstration. The factor must come from an authorized source in real use.

```
def calculated_dose(weight_kg, mg_per_kg):
    return weight_kg * mg_per_kg

result = calculated_dose(24, 10)
print(f"Calculated amount: {result:.0f} mg")
```

```
Calculated amount: 240 mg
```

Body mass index (BMI)

BMI = weight in kilograms ÷ height in metres squared.

```
def bmi(weight_kg, height_m):
    return weight_kg / (height_m ** 2)

value = bmi(60, 1.65)
print(f"BMI: {value:.1f}")
```

```
BMI: 22.0
```

Add input validation

```
weight = float(input("Weight (kg): "))
height = float(input("Height (m): "))
if weight <= 0 or height <= 0:
    print("Enter positive values")
else:
    print(f"BMI: {bmi(weight, height):.1f}")
```

Interpretation caution: A computed value is not automatically a diagnosis or prescribing decision. Programming supports a workflow; professional judgment and approved guidance remain necessary.

Worksheet 2: logic and functions

A. Choose the structure

1. Use `if`, `for`, or `while`: (a) inspect every batch in a list; (b) repeat until a valid positive number is entered; (c) print a warning only when stock is below 10.

B. Predict and trace

```
total = 0
for x in range(1, 5):
    if x == 3:
        continue
    total += x
print(total)
```

2. Complete a trace table and predict the output.

C. Debug

```
def area(length width)
return length * width
print(area(4, 5))
```

3. Correct the colon, comma, and indentation errors.

D. Write programs

4. Print “Reorder” when stock is below the reorder level; otherwise print “Sufficient”.

5. Use a loop to print doses 1 through 5.

6. Write `total_quantity()` and call it with values entered by the user.

7. Write a BMI function that rejects zero or negative height.

Practical task

Create a menu with three choices: total quantity, BMI, and exit. Use functions for both calculations and a loop to keep the menu running until exit is selected.

Lists, tuples, and dictionaries

A **data structure** organizes several values so a program can store and retrieve them predictably.

Structure	Definition	Best fit
List `[]`	Ordered, mutable collection.	Medicine names that may change
Tuple `()`	Ordered, immutable collection.	Fixed coordinates or reference values
Dictionary `{}`	Mutable mapping of unique keys to values.	One medicine record with named fields

Mutable means an object can be changed after creation. **Immutable** means it cannot be changed in place. **Ordered** means items retain a predictable sequence.

```
drugs = ["ORS", "Zinc"]
drugs.append("Paracetamol")

reference = (2, 8)  # storage range in °C

medicine = {
    "name": "ORS",
    "stock": 25,
    "reorder_level": 10
}
print(medicine["stock"])
```

```
25
```

Access patterns

- A list or tuple uses a numerical index: `drugs[0]`.
- A dictionary uses a key: `medicine["name"]`.
- `medicine.get("price")` returns `None` when the key is absent, instead of raising `KeyError`.

Choose by meaning: Use a list for a sequence, a tuple for a fixed group, and a dictionary when each value needs a field name.

Lists store ordered, changeable collections

A list keeps several values in order. Lists are **mutable**, so their contents can change.

```
medicines = ["Paracetamol", "Cetirizine", "ORS"]
print(medicines[0])
medicines.append("Zinc")
medicines[1] = "Levocetirizine"
print(medicines)
```

```
Paracetamol
['Paracetamol', 'Levocetirizine', 'ORS', 'Zinc']
```

Task	Code
Add one item	`items.append(x)`
Insert at position	`items.insert(i, x)`
Remove by value	`items.remove(x)`
Remove and return last	`items.pop()`
Count items	`len(items)`
Sort in place	`items.sort()`

Indexing and slicing

```
levels = [10, 20, 30, 40, 50]
print(levels[1])      # 20
print(levels[1:4])    # [20, 30, 40]
print(levels[-2:])    # [40, 50]
```

Memory aid: A slice stops before the ending index. In `[1:4]`, positions 1, 2, and 3 are included.

Tuples and dictionaries

Tuple

An ordered collection that is normally not changed after creation.

```
dimensions = (10, 5, 2)
print(dimensions[0])
```

Good for a fixed group of related values.

Dictionary

Stores key-value pairs. Access a value by its key.

```
drug = {
  "name": "Amoxicillin",
  "strength_mg": 500,
  "stock": 24
}
print(drug["stock"])
```

Dictionary operations

```
drug["dosage_form"] = "capsule"   # add
drug_stock = drug.get("stock", 0)    # safe access
for key, value in drug.items():
    print(key, value)
```

Question	Best structure	Why
Daily temperatures in order	List	Ordered and changeable
Fixed coordinate	Tuple	Ordered and intended to stay fixed
One patient record	Dictionary	Named fields map to values

Key idea: Dictionary keys make fields self-explanatory, while `get()` can provide a default when a key is missing.

NumPy: numerical arrays

NumPy is a third-party Python library for numerical computing. Its central object is the **ndarray**, usually called an array: a fixed-size, multidimensional collection whose elements normally share one data type.

```
import numpy as np

concentration = np.array([0.0, 4.8, 7.2, 5.1])
print(concentration.shape)
print(concentration.dtype)
print(concentration.mean())
```

```
(4,)
float64
4.275
```

Term	Definition
Element	One value in an array.
Shape	Size along each dimension; `(4,)` means one dimension with four values.
Dimension/axis	A direction along which values are arranged.
`dtype`	The array's element data type.
Vectorization	Applying an operation to many elements without writing an explicit Python loop.

Why array arithmetic differs

```
values = np.array([1, 2, 3])
print(values * 10)
```

```
[10 20 30]
```

The multiplication is performed element by element. By contrast, the Python list `[1, 2, 3] * 2` repeats the list.

Interpretation: Fast calculation does not guarantee correct units, study design, or clinical meaning. Check those separately.

NumPy arrays support numerical work

NumPy provides fast, convenient arrays for numerical data. Install it once, then import it by the conventional name np.

```
import numpy as np

concentrations = np.array([0.0, 4.2, 7.8, 5.1])
print(concentrations + 1)
print(concentrations * 2)
print(concentrations.mean())
```

```
[1.  5.2 8.8 6.1]
[ 0.   8.4 15.6 10.2]
4.275
```

Array creation

Code	Purpose
`np.array([1,2,3])`	Create from values
`np.zeros(4)`	Four zeros
`np.ones(3)`	Three ones
`np.arange(0,10,2)`	0, 2, 4, 6, 8

List arithmetic

```
[1, 2] * 2
# [1, 2, 1, 2]
```

Array arithmetic

```
np.array([1, 2]) * 2
# [2, 4]
```

Core idea: Operations are applied element by element. This is called vectorized computation.

CSV files: tables stored as text

CSV means comma-separated values. A CSV file is plain text arranged as rows and columns. The first row often contains **headers**, which name the columns.

```
drug,stock,reorder_level
ORS,25,10
Zinc,8,12
```

Term	Definition
Row/record	One observation, such as one medicine.
Column/field	One property recorded for all rows.
Delimiter	Character that separates fields, commonly a comma.
Header	First row containing field names.
File path	Text describing where a file is stored.

Read safely with the standard library

```
import csv

with open("stock.csv", "r", newline="", encoding="utf-8") as file:
    reader = csv.DictReader(file)
    for row in reader:
        stock = int(row["stock"])
        print(row["drug"], stock)
```

`with` creates a context that closes the file automatically. Mode `"r"` means read. Text from CSV is initially read as strings, so numeric fields may need conversion.

Data quality check: Confirm headers, units, missing cells, unexpected spaces, and whether each row contains the expected number of fields.

Read and write CSV files

A comma-separated values (CSV) file stores a table as plain text. Python's built-in `csv` module can read and write it.

Write a file

```
import csv

rows = [
    ["drug", "stock"],
    ["Paracetamol", 40],
    ["ORS", 18]
]
with open("stock.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(rows)
```

Read the same file

```
with open("stock.csv", "r", newline="") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

```
['drug', 'stock']
['Paracetamol', '40']
['ORS', '18']
```

Why `with`?
It closes the file automatically, even if an error occurs.

Important type detail
CSV values are read as strings. Convert numerical fields before arithmetic.

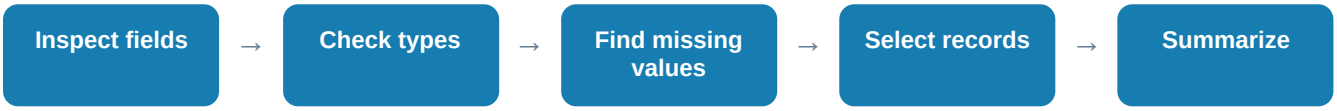
Header-based reading

```
with open("stock.csv", newline="") as file:
    for row in csv.DictReader(file):
        print(row["drug"], int(row["stock"]))
```

Understand a structured healthcare dataset

A structured dataset arranges observations in rows and variables in columns. A clear **data dictionary** explains each column.

patient_id	drug	dose_mg	response
P001	Drug A	50	Improved
P002	Drug B	25	No change
P003	Drug A	50	Improved



Basic access with dictionaries

```
records = [
 {"patient_id":"P001", "drug":"Drug A", "dose_mg":50},
 {"patient_id":"P002", "drug":"Drug B", "dose_mg":25}
]
for record in records:
    if record["dose_mg"] >= 50:
        print(record["patient_id"])
```

```
P001
```

Responsible data use: Remove or protect personal identifiers, use data only for the approved purpose, and follow institutional and legal requirements. The example identifiers above are fictional.

Mini project: stock checker

This program combines a list of dictionaries, a loop, a condition, and a CSV export.

```
import csv

items = [
 {"drug":"Paracetamol", "stock":40, "reorder":20},
 {"drug":"ORS", "stock":8, "reorder":10},
 {"drug":"Cetirizine", "stock":15, "reorder":15}
]

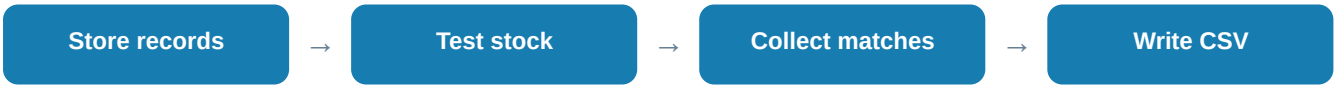
low_stock = []
for item in items:
    if item["stock"] <= item["reorder"]:
        low_stock.append(item)

with open("reorder.csv", "w", newline="") as file:
    fields = ["drug", "stock", "reorder"]
    writer = csv.DictWriter(file, fieldnames=fields)
    writer.writeheader()
    writer.writerows(low_stock)

print("Items to reorder:", len(low_stock))
```

```
Items to reorder: 2
```

How the program flows



Extension: Ask the user for a drug name and report whether it appears in `low_stock`.

Worksheet 3: structures and files

A. Match

1. Choose list, tuple, or dictionary: (a) an unchanging RGB color; (b) medicine names that may be edited; (c) a record with fields for name, strength, and stock.

B. Predict

```
values = [10, 20, 30, 40]
values.append(50)
print(values[1:4])
print(len(values))
```

2. Write both outputs.

C. NumPy

```
import numpy as np
x = np.array([2, 4, 6])
print(x * 5)
print(x.mean())
```

3. Predict the two outputs and explain why array multiplication differs from list repetition.

D. Write and debug

4. Create a dictionary for one drug and update its stock.

5. Read `dose_mg` from a CSV row and convert it before adding 10.

6. Write a list slice that returns the last three items.

Practical task

Create `inventory.csv` with columns `drug`, `strength_mg`, `stock`, and `reorder_level`. Read every row, convert numerical columns, and write matching low-stock records to `reorder.csv`.

Evidence to submit: source code, both CSV files, terminal output, and a five-sentence explanation of the logic.

Pandas: a table Python can analyze

Pandas is a third-party library for labeled, tabular data. A **Series** is a one-dimensional labeled collection. A **DataFrame** is a two-dimensional table with labeled rows and columns.

```
import pandas as pd

data = {
    "drug": ["A", "B", "A"],
    "dose_mg": [50, 100, 75]
}
df = pd.DataFrame(data)
print(df)
```

```
  drug  dose_mg
0    A       50
1    B      100
2    A       75
```

Term	Definition
Index	Labels used to identify rows.
Column	Labeled variable containing values down the table.
Cell	Value at one row-column intersection.
Missing value	Absent entry, often represented as `NaN`.
Data type	Kind of values in a column, available through `dtypes`.

Inspect before changing

```
print(df.shape)      # rows, columns
print(df.columns)    # column labels
print(df.dtypes)     # column types
print(df.head())     # first rows
```

Mental model: A DataFrame is similar to a spreadsheet table, but every step can be written as code and repeated exactly on another file with the same structure.

Pandas turns tables into programmable objects

Pandas is a third-party library for working with tabular data. Two central structures are the one-dimensional **Series** and two-dimensional **DataFrame**.

Series

```
import pandas as pd
s = pd.Series([12, 18, 9],
              name="stock")
print(s)
```

One labeled column of values.

DataFrame

```
df = pd.DataFrame({
 "drug":["A","B"],
 "dose_mg":[50,25]
})
print(df)
```

A table with rows and columns.

Selection basics

```
print(df["drug"])            # one column
print(df[["drug","dose_mg"]]) # several columns
print(df.loc[0, "drug"])      # label-based access
print(df.iloc[0, 1])           # position-based access
```

Expression	Returns
`df.shape`	(number of rows, number of columns)
`df.columns`	Column labels
`df.dtypes`	Data type of each column

Load CSV and Excel data

```
import pandas as pd

pk = pd.read_csv("pk_study.csv")
adr = pd.read_excel("adr_reports.xlsx", sheet_name="Reports")
```

Read with intent

Question	Check
Did the file load?	`df.shape` and `df.head()`
Are headers correct?	`df.columns`
Are numbers actually numeric?	`df.dtypes`
Are missing entries recognized?	`df.isna().sum()`

Useful options

```
df = pd.read_csv(
    "pk_study.csv",
    usecols=["patient_id", "time_h", "conc_mg_L"],
    na_values=["NA", "missing", "-"]
)
```

Path problem? Use `import os; print(os.getcwd())` to see the current working folder. Put the data file there or supply the correct path.

Save a cleaned table

```
df.to_csv("pk_study_clean.csv", index=False)
# Excel output needs an appropriate Excel engine:
df.to_excel("pk_study_clean.xlsx", index=False)
```

Use `index=False` when the DataFrame index should not become an extra file column.

Inspect before you analyze

Inspection prevents silent mistakes. Never jump straight to a chart or average.

Method	Question answered
`df.head()`	What do the first five rows look like?
`df.tail()`	What do the final five rows look like?
`df.info()`	How many non-null values and what data types?
`df.describe()`	What are count, mean, spread, and range for numeric columns?

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

Read `describe()` carefully

- **count:** number of non-missing values used.
- **mean:** arithmetic average.
- **std:** standard deviation, a measure of spread.
- **min/max:** smallest and largest values.
- **25%, 50%, 75%:** quartiles; 50% is the median.

Check units: A column named `dose` is ambiguous. Prefer `dose_mg`. A number without its unit is easy to misuse.

Cleaning data is a documented decision

Data cleaning means detecting and addressing missing, inconsistent, duplicated, or incorrectly typed values before analysis. Cleaning is not merely deleting inconvenient rows; every action should have a reason.

Problem	Example	Possible action
Missing value	Age is blank	Investigate; drop, fill, or retain
Inconsistent text	`oral`, `ORAL`	Strip spaces and standardize case
Wrong type	Dose stored as text	Convert with error checking
Duplicate row	Same report entered twice	Verify before removal
Impossible value	Negative age	Check source; flag as invalid

```
df["route"] = df["route"].str.strip().str.upper()
df["dose_mg"] = pd.to_numeric(df["dose_mg"], errors="coerce")
print(df.isna().sum())
```

Missing-data choices

- `dropna()` removes rows or columns containing missing values.
- `fillna(value)` replaces missing values using a stated value or rule.
- `isna()` identifies missing positions.

Never hide the decision: Save cleaned data separately, preserve the raw file, and record what was changed. Different cleaning choices can change the result.

Clean missing and inconsistent data

Detect → Understand → Choose action → Document

Missing values

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

complete = df.dropna(subset=["time_h", "conc_mg_L"])
df["age"] = df["age"].fillna(df["age"].median())
```

Dropping and filling answer different questions. Do not select a method only because it removes the warning.

Convert and standardize

```
df["dose_mg"] = pd.to_numeric(df["dose_mg"], errors="coerce")
df["route"] = df["route"].str.strip().str.upper()
df = df.drop_duplicates()
```

Problem	Example	Possible action
Missing	Blank concentration	Investigate; drop or impute with justification
Wrong type	`"50 mg"` in numeric field	Parse or mark invalid
Inconsistent text	`oral`, `Oral`	Strip and standardize case
Duplicate	Same report entered twice	Confirm key, then remove duplicate

Do not hide uncertainty: A filled value is still an estimate. Record the method and preserve the raw file.

Select, filter, group, and summarize

Selection chooses particular rows or columns. **Filtering** keeps rows that satisfy a Boolean condition. **Grouping** divides rows into categories. **Aggregation** reduces many values to summaries such as a count, mean, minimum, or maximum.

```
high_dose = df[df["dose_mg"] > 50]
summary = df.groupby("drug")["dose_mg"].mean()
print(summary)
```

Line-by-line

1. `df["dose_mg"] > 50` produces a Boolean Series.
2. `df[condition]` keeps rows where that Series is true.
3. `groupby("drug")` separates rows by drug label.
4. Selecting `["dose_mg"]` identifies the values to summarize.
5. `mean()` calculates the arithmetic mean within each group.

Method	Result
`count()`	Number of non-missing values
`size()`	Number of rows, including missing values
`mean()`	Arithmetic mean
`median()`	Middle ordered value
`sum()`	Total

Question first: Decide the scientific or operational question before selecting an aggregation. A mean and a count answer different questions.

Filter and select the records you need

Boolean filtering keeps rows where a condition is true.

```
high_dose = df[df["dose_mg"] >= 100]
oral_only = df[df["route"] == "ORAL"]

selected = df[
    (df["route"] == "ORAL") &
    (df["dose_mg"] >= 100)
]
print(selected[["patient_id", "dose_mg"]])
```

Logical operators in Pandas

Meaning	Operator	Example
AND	&	Both conditions must be true
OR	|	At least one condition is true
NOT	~	Reverse a condition

Parentheses are required: Write `(condition1) & (condition2)`. Python's plain `and`/`or` are not used for element-wise Series filtering.

Useful selectors

```
df[df["drug"].isin(["Drug A", "Drug B"])]
df[df["report_date"].between("2026-01-01", "2026-03-31")]
df[df["reaction"].str.contains("rash", case=False, na=False)]
```

Group and aggregate

`groupby()` follows the split-apply-combine idea: split rows into groups, apply a calculation, then combine the results.

```
summary = (
    df.groupby("drug")["dose_mg"]
      .agg(["count", "mean", "min", "max"])
      .reset_index()
)
print(summary)
```

Split by drug → **Calculate statistics** → **Return summary table**

Common aggregations

Method	Meaning
`count()`	Number of non-missing values
`sum()`	Total
`mean()`	Arithmetic average
`median()`	Middle value after sorting
`min()`, `max()`	Range endpoints

Count categories

```
reaction_counts = df["reaction"].value_counts(dropna=False)
route_percent = df["route"].value_counts(normalize=True) * 100
```

Rate versus count: A count is not a rate. To compare reporting rates, define and use a meaningful denominator.

Case study workflow: PK and ADR tables

Pharmacokinetic (PK) table

```
pk = pd.read_csv("pk_study.csv")
pk["time_h"] = pd.to_numeric(pk["time_h"], errors="coerce")
pk["conc_mg_L"] = pd.to_numeric(pk["conc_mg_L"], errors="coerce")
pk = pk.dropna(subset=["time_h", "conc_mg_L"])
mean_curve = pk.groupby("time_h")["conc_mg_L"].mean()
print(mean_curve)
```

Adverse drug reaction (ADR) table

```
adr = pd.read_excel("adr_reports.xlsx")
adr["drug"] = adr["drug"].str.strip().str.title()
adr["serious"] = adr["serious"].fillna("Unknown")
counts = adr.groupby(["drug", "serious"]).size()
print(counts)
```

PK question
How does concentration change over time? Keep time and concentration units explicit.

ADR question
How many reports fall into each recorded category? Do not infer causality from counts alone.

Reproducible order

1. Keep the original file unchanged.
2. Inspect structure and data types.
3. Clean with stated rules.
4. Filter or group to answer a defined question.
5. Save the cleaned data and code.

Worksheet 4: Pandas

A. Explain

1. Distinguish a Series from a DataFrame.

2. What do `head()`, `info()`, and `describe()` reveal?

3. Why should the raw data file remain unchanged?

B. Write expressions

4. Select rows where `route` is ORAL and `dose_mg` is above 50.

5. Count missing values in each column.

6. Calculate the mean concentration for each time point.

C. Debug

```
selected = df[df["dose_mg"] > 50 and df["route"] == "ORAL"]
```

7. Replace the incorrect operator and add required parentheses.

Practical task

Create a fictional 12-row ADR dataset with columns `report_id`, `drug`, `reaction`, `serious`, and `age`. Include two missing ages and two inconsistent drug labels. Then:

1. Load and inspect the file.
2. Standardize drug labels.
3. Report missing values; choose and explain an age-handling method.
4. Filter serious reports.
5. Group counts by drug and seriousness.
6. Export a cleaned CSV.

Matplotlib: build and label a chart

Data visualization means representing data graphically to reveal patterns and communicate findings. **Matplotlib** is a Python plotting library. In its object-oriented approach, a **Figure** is the complete canvas and an **Axes** is the plotting area that holds data, labels, and a title.

```
import matplotlib.pyplot as plt

time_h = [0, 1, 2, 4, 8]
conc = [0, 4.2, 7.1, 5.0, 2.3]

fig, ax = plt.subplots()
ax.plot(time_h, conc, marker="o", label="Oral")
ax.set_xlabel("Time (hours)")
ax.set_ylabel("Concentration (mg/L)")
ax.set_title("Illustrative concentration-time profile")
ax.legend()
ax.grid(alpha=0.25)
plt.show()
```

Chart vocabulary

Term	Definition
x-axis	Usually the horizontal scale.
y-axis	Usually the vertical scale.
Marker	Symbol showing an observed data point.
Legend	Key that identifies plotted series.
Scale	Mapping between positions and numerical values.
Outlier	Observation unusually distant from others; not automatically an error.

Minimum standard: A scientific chart needs a meaningful title, clearly named axes, units where applicable, and a legend when more than one series is displayed.

Choose the chart that matches the question

Chart	Definition	Question answered
Line plot	Points connected in sequence, often across time.	How does a value change over ordered measurements?
Histogram	Adjacent bins show frequency distribution of one numerical variable.	How are values distributed?
Scatter plot	Each point represents paired numerical measurements.	Are two numerical variables associated?
Box plot	Compact display of median, quartiles, spread, and potential outliers.	How do distributions compare?

Interpret without overclaiming

Trend is a general direction in data. **Association** means variables vary together. **Causation** means a change in one variable produces a change in another. A scatter plot may show association, but it does not prove causation.

Weak statement
"The drug caused the improvement."

Supported statement
"In this illustrative dataset, the measured value increased as dose increased."

Read a chart in four steps

1. Read the title and identify the question.
2. Read both axes, units, and legend.
3. Describe the main pattern and unusual points.
4. State limits: sample, missing values, denominator, and whether the data are illustrative.

Visual honesty: Do not crop inconvenient data, omit units, use a misleading axis range, or imply a clinical conclusion from a classroom dataset.

Matplotlib turns numbers into charts

Matplotlib is a plotting library. The common pattern is: create a figure and axes, draw marks, label the chart, then display it.

```
import matplotlib.pyplot as plt

x = [0, 1, 2, 4, 6]
y = [0, 5.5, 8.2, 6.4, 4.6]

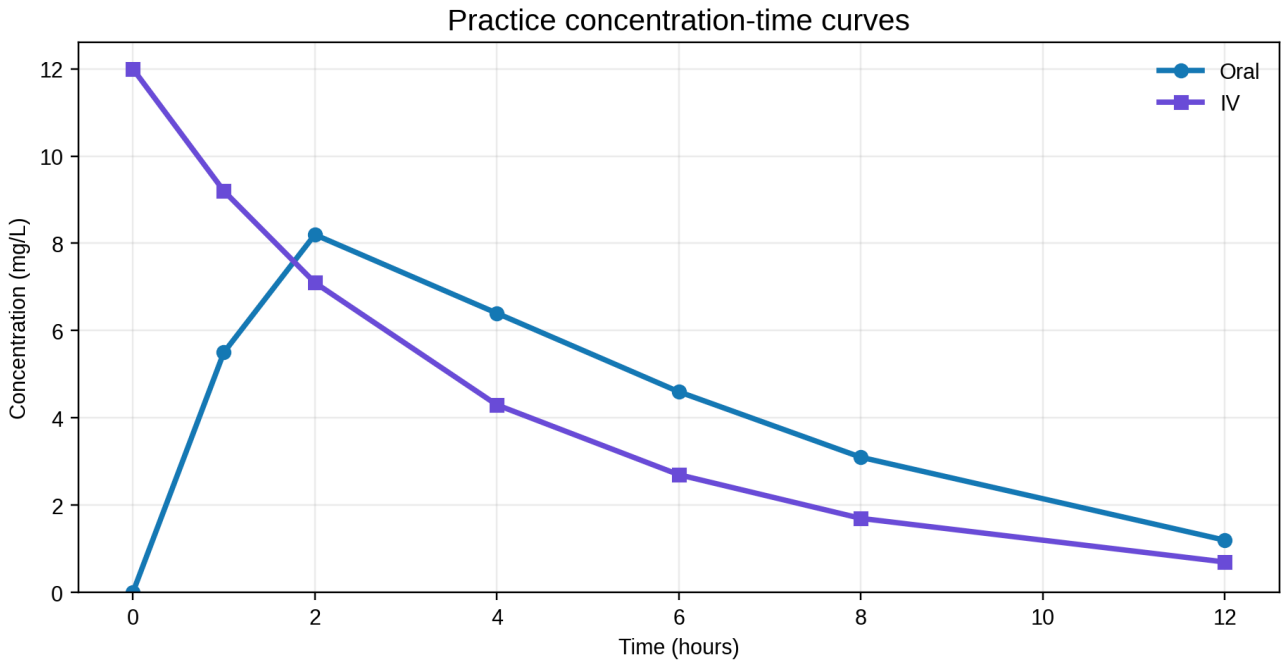
fig, ax = plt.subplots()
ax.plot(x, y, marker="o", label="Oral")
ax.set_xlabel("Time (hours)")
ax.set_ylabel("Concentration (mg/L)")
ax.set_title("Concentration-time curve")
ax.legend()
plt.show()
```

Choose a plot for the question

Plot	Best question
Line	How does a value change across ordered time?
Histogram	How are one variable's values distributed?
Scatter	How do two numerical variables relate?
Box plot	How do center, spread, and unusual values compare?

Every chart needs: a descriptive title, labeled axes, units, and a legend when more than one series is shown.

Line plots: concentration over time



Practice data only. Values are illustrative and are not clinical or product data.

Read the chart scientifically

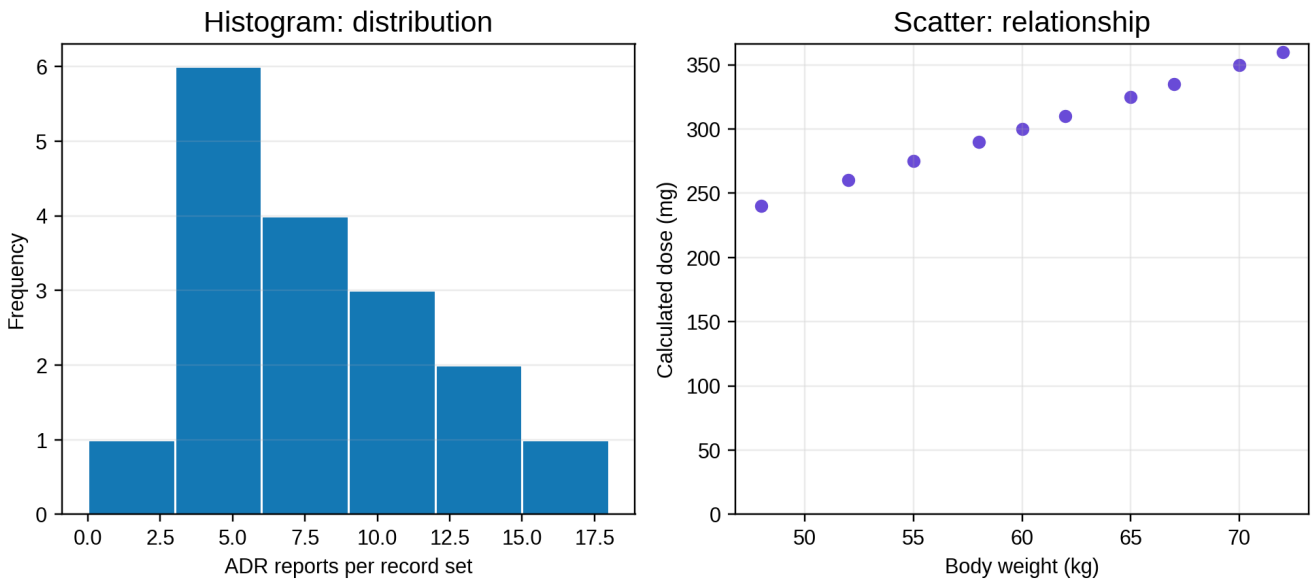
- The oral series begins at zero, rises to its highest plotted value at 2 hours, then declines.
- The intravenous (IV) series begins at its highest plotted value and declines across the shown time points.
- The lines summarize the sample values; they do not establish a dose recommendation or therapeutic range.

```
ax.plot(time, oral, marker="o", label="Oral")
ax.plot(time, iv, marker="s", label="IV")
ax.set_ylim(bottom=0)
ax.grid(alpha=0.25)
ax.legend()
```

Interpretation formula: Describe the pattern → identify evidence from axes/values → state the limit of the data.

Histograms and scatter plots

Illustrative practice data



Histogram

Bars show how many observations fall inside numerical intervals called bins. It answers a distribution question; it does not compare named categories.

Scatter plot

Each point represents a pair of numerical values. A visible pattern may suggest association, but it does not by itself prove that one variable causes the other.

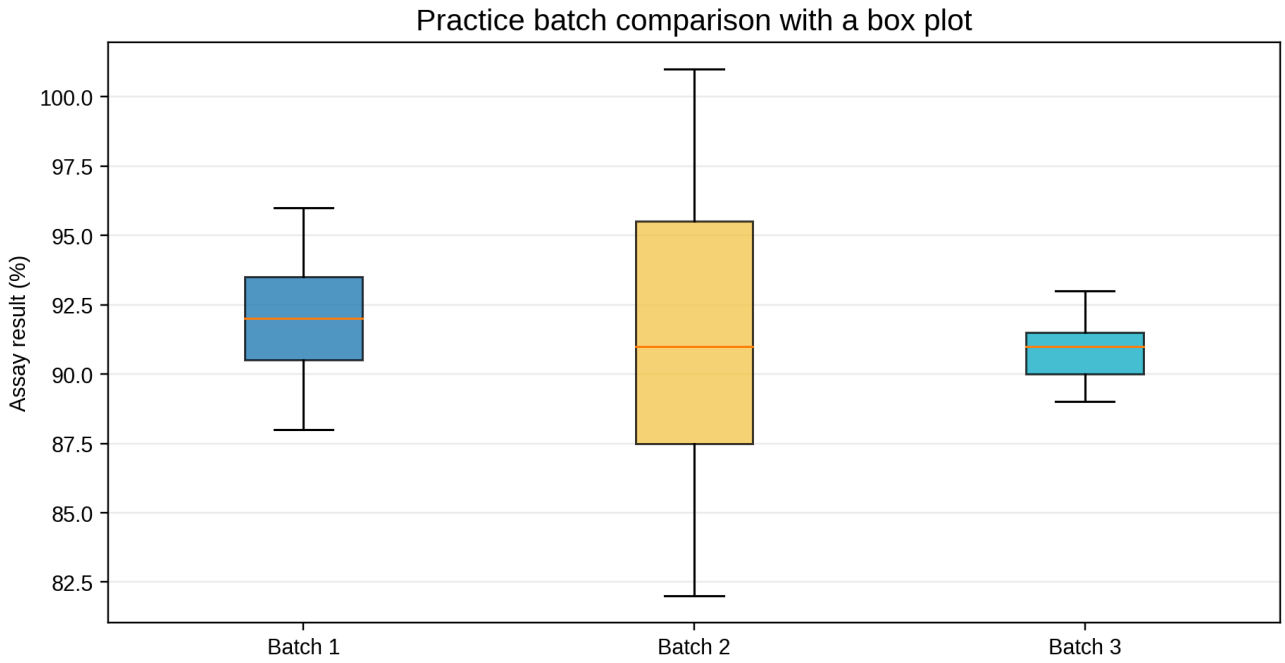
```
ax.hist(values, bins=6, edgecolor="white")
ax.set_xlabel("Value")
ax.set_ylabel("Frequency")

ax.scatter(weight_kg, dose_mg)
ax.set_xlabel("Body weight (kg)")
ax.set_ylabel("Calculated dose (mg)")
```

Common histogram error
Changing bin width can change the visual impression. State or choose bins thoughtfully.

Common scatter error
Connecting independent observations with a line can falsely imply order.

Box plots compare distributions



A box plot summarizes a distribution using the median, quartiles, whiskers, and possible outliers. It is useful for comparing spread across groups.

Median
Line inside the box.

Box
Middle 50% of values.

Whiskers
Typical range beyond the box.

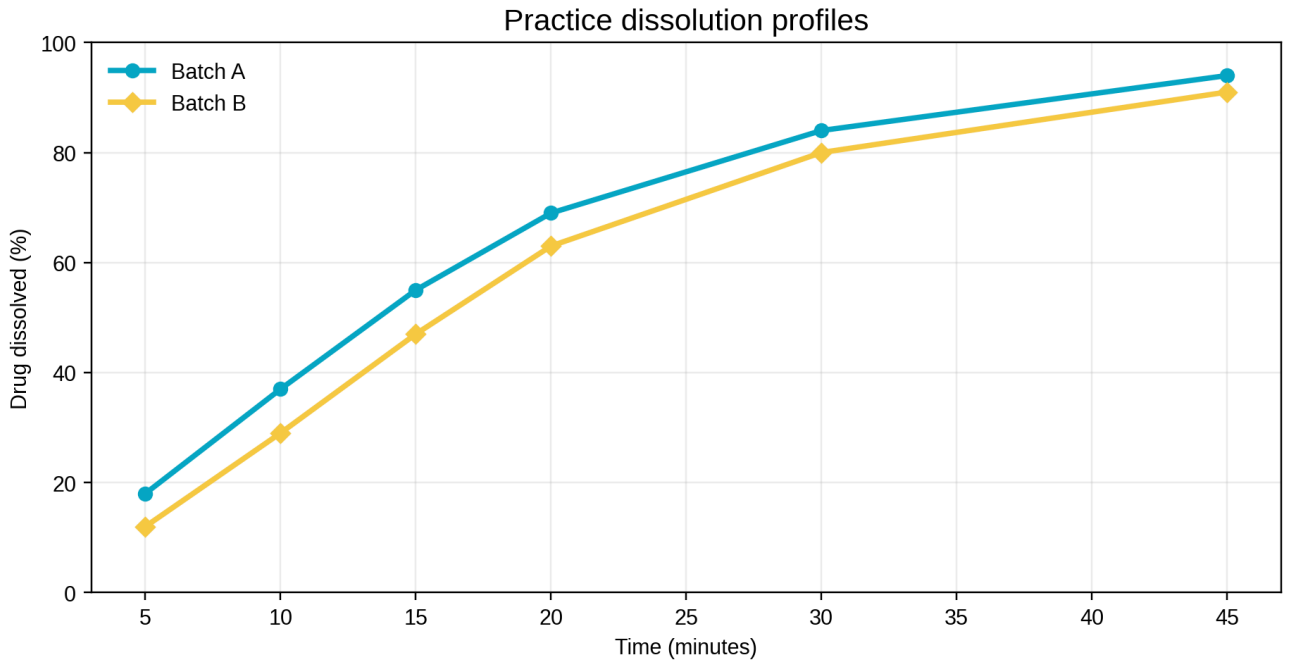
```
data = [batch1, batch2, batch3]
ax.boxplot(data, labels=["Batch 1", "Batch 2", "Batch 3"])
ax.set_ylabel("Assay result (%)")
ax.set_title("Batch comparison")
```

What this practice chart shows

Batch 3 has the narrowest visible spread in this sample. Batch 2 has the widest. A box plot supports comparison, but specification compliance requires the actual validated limits and method.

Do not overread: A small sample may not represent long-term process performance.

Dissolution profiles and clear labels



Line plots are suitable because time is ordered. Both axes carry units, and each batch has a separate legend entry.

```
ax.plot(time_min, batch_a, marker="o", label="Batch A")
ax.plot(time_min, batch_b, marker="D", label="Batch B")
ax.set_xlabel("Time (minutes)")
ax.set_ylabel("Drug dissolved (%)")
ax.set_ylim(0, 100)
ax.legend()
```

Write an interpretation

In the illustrative dataset, dissolved percentage rises over time for both batches. Batch A is higher than Batch B at each displayed time point. This statement describes the plotted values only; it does not determine product acceptance.

Save a figure

```
fig.tight_layout()
fig.savefig("dissolution.png", dpi=300, bbox_inches="tight")
```

For reports: Save at sufficient resolution, keep text readable, and include a caption explaining the dataset and units.

Design and interpret charts honestly

Chart quality checklist

1. **Question:** State what the chart should answer.
2. **Plot:** Match the chart type to time, distribution, relationship, or comparison.
3. **Scale:** Keep bar charts at a zero baseline; do not clip data.
4. **Labels:** Include names and units on axes.
5. **Legend:** Identify every series.
6. **Missing data:** Show a gap rather than pretending missing means zero.
7. **Interpretation:** Separate observation from explanation.

Weak statement	Better statement
“Drug A is better.”	“Drug A has the higher plotted mean in this sample.”
“Weight causes dose.”	“The plotted values show a positive association.”
“No ADR problem.”	“No reports appear in this dataset; under-reporting was not assessed.”

Count versus rate

Compare rates only when an appropriate denominator is available, such as reports per defined number of exposed patients or prescriptions. Name the denominator.

Reproducibility

Keep the source data, cleaning code, plotting code, and saved figure together so another student can repeat the result.

Worksheet 5: visualization

A. Choose the chart

1. Concentration at 0, 1, 2, 4, and 8 hours.

2. Distribution of patient ages.

3. Relationship between weight and a calculated amount.

4. Spread of assay results across four batches.

B. Repair the chart code

```
plt.plot(time, conc)
plt.title("Graph")
plt.show()
```

5. Add an informative title, both axis labels with units, markers, and a grid.

C. Interpret

6. Write three sentences about the dissolution chart: one pattern, one comparison, and one limitation.

7. Explain why an ADR count cannot automatically be called an ADR rate.

Practical task

Recreate the small practice datasets shown in the Unit V charts, save them as CSV files, and create:

1. a labeled oral/IV concentration-time line plot;
2. a dissolution plot with two batches;
3. a box plot comparing assay batches;
4. a 100-word interpretation that avoids causal or clinical claims.

Submission checklist: Code runs; labels and units are visible; legend matches series; images are saved; interpretation matches the data shown.

Answer key: Unit I

1-3. Recall

Variable: a name referring to a value. **Data type:** the kind of value and operations it supports. **Type casting:** converting a value to another type. **Library:** reusable code for common tasks.

Types: `int` (`stock=20`), `float` (`volume_ml=5.5`), `str` (`route="oral"`), and `bool` (`sterile=True`). An IDE can run/debug code and manage files/packages in one place.

4. Output

```
2.5
2
1
```

5. Corrected code

```
drug = "Paracetamol"
print("Drug:", drug)
```

6-8. Model answers

```
name = input("Name: ")
age = int(input("Age: "))
print(f"{name} is {age} years old.")

price = float(input("Unit price: "))
quantity = int(input("Quantity: "))
print(f"Total: ₹{price * quantity:.2f}")

batch = input("Batch code: ")
print(batch[-4:])
```

Practical task: key steps

Read four inputs, apply `strip()`, apply `title()` to the drug name, and use one f-string per label line. Test with leading and trailing spaces.

Answer key: Unit II

1. Structures

(a) `for`; (b) `while`; (c) `if`.

2. Trace

The loop adds 1, 2, and 4; it skips 3. Output: **7**.

3. Corrected function

```
def area(length, width):
    return length * width

print(area(4, 5))
```

4-7. Model answers

```
if stock < reorder_level:
    print("Reorder")
else:
    print("Sufficient")

for dose_no in range(1, 6):
    print("Dose", dose_no)

def total_quantity(units, per_day, days):
    return units * per_day * days

def bmi(weight, height):
    if height <= 0:
        return None
    return weight / height**2
```

Practical design: Put calculations in functions. Use a `while True` menu and `break` only for the exit choice.

Answer key: Unit III

1. Match

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

2. Output

```
[20, 30, 40]
5
```

3. NumPy output

```
[10 20 30]
4.0
```

A list multiplied by 5 repeats its items. A NumPy array multiplied by 5 multiplies each numerical element.

4-6. Model answers

```
drug = {"name":"ORS", "stock":12}
drug["stock"] = 20

new_dose = int(row["dose_mg"]) + 10

last_three = items[-3:]
```

CSV practical checklist

- Header names match exactly.
- `int()` converts stock and reorder level.
- The condition is explicit: choose < or <= and explain it.
- `newline=""` is used when opening CSV files.
- The output contains only matching records.

Answer key: Unit IV

1-3. Concepts

A Series is one-dimensional; a DataFrame has rows and columns. `head()` previews rows, `info()` reports non-null counts and types, and `describe()` summarizes numerical columns. The raw file remains unchanged so cleaning decisions can be audited or reversed.

4-7. Model expressions

```
oral_high = df[
    (df["route"] == "ORAL") &
    (df["dose_mg"] > 50)
]

missing = df.isna().sum()
mean_by_time = df.groupby("time_h")["conc_mg_L"].mean()

selected = df[
    (df["dose_mg"] > 50) &
    (df["route"] == "ORAL")
]
```

ADR practical: expected workflow

Use `read_csv()`, then `shape`, `head()`, `info()`, and missing-value counts. Standardize with `str.strip().str.title()`. Decide how missing ages affect the question before dropping or filling. Filter seriousness with a Boolean condition, group using `groupby().size()`, and export with `index=False`.

Answer key: Unit V

1-4. Chart choice

1. Line plot. 2. Histogram. 3. Scatter plot. 4. Box plot.

5. Repaired chart

```
plt.plot(time, conc, marker="o")
plt.xlabel("Time (hours)")
plt.ylabel("Concentration (mg/L)")
plt.title("Concentration-time profile")
plt.grid(alpha=0.25)
plt.show()
```

6. Model interpretation

In the practice dataset, dissolved percentage increases over time for both batches. Batch A is higher than Batch B at every displayed time point. These are illustrative values, so the figure cannot establish product acceptance or predict another batch.

7. Count versus rate

A count is the number of recorded reports. A rate divides a count by a defined exposure or population measure. Without that denominator, rates cannot be compared.

Practical review: The chart title should describe the comparison; axes need units; series need distinct labels; saved images should remain legible when inserted into a report.

Practical examination toolkit

Before coding

1. Underline inputs, outputs, and formulae.
2. Choose a data structure.
3. Write pseudocode in 4-8 steps.
4. Prepare one normal and one edge-case test.

Reusable pseudocode

```
START
READ required values
VALIDATE values
PROCESS with formula or rule
DISPLAY result with unit
END
```

Ten practical tasks to rehearse

1. Medicine label formatter.
2. Total quantity calculator.
3. BMI calculator with validation.
4. Low-stock checker.
5. List search with `break`.
6. CSV reader and writer.
7. NumPy concentration summary.
8. Pandas missing-value report.
9. Grouped ADR count table.
10. Labeled concentration-time plot.

Five viva questions

- Why does `input()` return a string?
- When would you use `continue` instead of `break`?
- How does a NumPy array differ from a list?
- Why inspect `dtypes` before analysis?
- What can a scatter plot not prove?

Last check: Run the exact file you submit. Confirm output, labels, units, and filename.

Rapid revision sheet

Python essentials

Need	Pattern
Read + convert	`x = float(input("Value: "))`
Two-way decision	`if condition: ... else: ...`
Repeat sequence	`for item in items: ...`
Reusable calculation	`def f(x): return result`
Last three list items	`items[-3:]`
Safe dictionary access	`d.get("key", default)`

Data essentials

NumPy
`np.array()`, element-wise arithmetic, `mean()`, indexing, slicing.

Pandas
`read_csv()`, `head()`, `info()`, `describe()`, filtering, `groupby()`.

Plot essentials

```
fig, ax = plt.subplots()
ax.plot(x, y, marker="o", label="Series")
ax.set_xlabel("Name (unit)")
ax.set_ylabel("Name (unit)")
ax.set_title("Clear title")
ax.legend()
plt.show()
```

High-value exam distinctions

= vs == · / vs // · list vs tuple · break vs continue · parameter vs argument · count vs rate · Series vs DataFrame

Glossary

Argument: Value supplied to a function call.

Boolean: `True` or `False`.

CSV: Plain-text tabular file using separators.

DataFrame: Pandas table with rows and columns.

Dictionary: Mutable key-value collection.

Function: Named reusable block of code.

IDE: Software combining coding, running, and debugging tools.

Index: Position or label used to access data.

Iteration: One repetition of a loop.

Library: Reusable collection of code.

List: Ordered, mutable Python collection.

Missing value: Data entry that is absent or unavailable.

NumPy array: Numerical collection supporting vectorized operations.

Parameter: Input name in a function definition.

Series: One-dimensional labeled Pandas structure.

String: Text value.

Tuple: Ordered collection intended to remain fixed.

Type casting: Converting one data type to another.

Variable: Name referring to a value.

Vectorization: Applying an operation across array elements without writing a Python loop.

Abbreviations

ADR: adverse drug reaction. **BMI:** body mass index. **IDE:** integrated development environment. **IV:** intravenous. **PK:** pharmacokinetic.

Sources and brand channels

Primary course sources supplied

- *B.Pharm Syllabus - Basics of Python Programming for Pharmaceutical Sciences (BP101T)*, 3 pages.
- *Basics of Python Programming for Pharmaceutical Sciences - Complete Notes, Units I-V*, Pills and Python Signature Edition, 113 pages.
- *Python Book Final Healthcare*, 343 pages.
- *Unit-1 Python Programming Intro Notes*, Noteskarts, 14 pages.

Web references supplied

Noteskarts BP101T page:
https://noteskarts.com/b-pharm-1st-sem-python-notes-pdf-new-syllabus-2026-bp101t/

Carewell Pharma B.Pharmacy notes:
https://www.carewellpharma.in/bpharmacy/notes/

Pills and Python

Instagram
@pillsandpython

Website
pillsandpython.com

YouTube
@PillsAndPython

Focus
Pharmacy · Data · AI · Healthcare

Educational use: Programming examples and practice datasets are designed for learning. They are not patient-specific medical advice, prescribing instructions, or validated clinical software.