



Notes by PharmacoLearn

# Basics of Python Programming for Pharmaceutical Sciences

BP101T — B.Pharm Semester I

## Exam Success Notes

Theory-exam mastery: complete answers, the way examiners want them

As per PCI B.Pharm (NEP 2020) syllabus

Scan to connect with PharmacoLearn



YouTube  
Subscribe



Telegram  
Join the channel



Website  
Read the notes

© 2026 PharmacoLearn. All rights reserved.

No part of these notes may be reproduced, stored, or transmitted without prior written permission.  
[sites.google.com/view/pharmacolearn-notes](https://sites.google.com/view/pharmacolearn-notes)

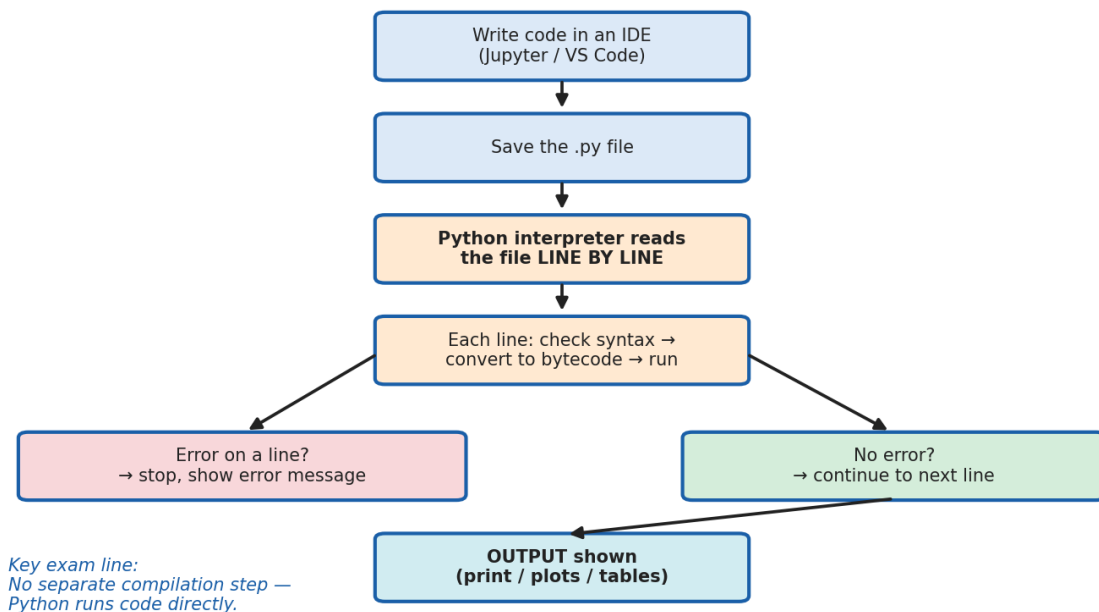
# Unit I — Introduction to Python Programming

## What Python is and how it runs

Python is a high-level, interpreted programming language. "Interpreted" means there is no separate compilation step — the Python interpreter reads your file and runs it line by line, converting each line to machine instructions as it goes. If one line has an error, the program stops at that line and reports it.

- High-level — you write in readable, English-like statements, not machine code.
- Interpreted — no compiler needed; run the file directly and see output immediately.
- Dynamically typed — you never declare a variable's type; Python works it out from the value you assign.
- Why pharmacy? — quick dose calculations, cleaning datasets, analysing PK studies and drawing graphs, all in a few lines.

### How a Python program runs (interpreted, line by line)



How a Python program runs, line by line

Exam tip: The one-line answer for "What is an interpreted language?" is: a language in which the interpreter executes the source code directly, line by line, without a separate compilation step. Write exactly this and you have the 2-mark answer.

## Installing Python

- 1 Download the installer from the official website, [python.org](https://python.org).
- 2 Run it on Windows and tick "Add Python to PATH" during installation — without this, the terminal cannot find Python.
- 3 Verify the installation by opening a terminal and typing:

```
python --version
```

## Installing Python and an IDE

An IDE (Integrated Development Environment) is software that combines a code editor with programmer's tools — a debugger, a terminal, syntax highlighting, autocomplete and a package manager — in one window. The syllabus names three:

| IDE              | What it is                                                       | Best for                                                                                                      |
|------------------|------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| Jupyter Notebook | Runs in your browser; code executes in small blocks called cells | Data analysis — output, tables and plots appear right below each cell (you will use this most in Units III–V) |
| PyCharm          | A full IDE built specifically for Python                         | Large projects; strong autocomplete and built-in debugger, but heavier to install                             |
| VS Code          | A lightweight editor with Python extensions                      | A fast middle ground for everyday coding                                                                      |

Remember: IDE = Integrated Development Environment. Jupyter, PyCharm and VS Code — know all three by name, because the question "Name any three Python IDEs" is a free 2 marks.

## Advantages of IDEs over text editors

A plain text editor (Notepad, Notepad++) only lets you type and save text. An IDE does far more:

| IDE                                                  | Plain text editor                               |
|------------------------------------------------------|-------------------------------------------------|
| Syntax highlighting + autocomplete as you type       | No code assistance at all                       |
| Built-in debugger — step through errors line by line | You debug by guessing                           |
| Integrated terminal and package installer            | You juggle separate tools                       |
| Error warnings appear while you type                 | Errors appear only when you run the code        |
| One-click run button                                 | You must save, open a terminal and run manually |

Exam tip: "Explain the advantages of IDEs over text editors with examples" is a classic 5-mark question. Open with the one-line IDE definition, give 4-5 points from the table above, and close by naming Jupyter as the IDE you would pick for data analysis and why.

## Python variables and data types

A variable is a named container that stores a value. Because Python is dynamically typed, you write **dose = 500** without declaring any type — Python decides from the value.

```
drug_name = "Paracetamol"    # string
dose_mg = 500                # integer
half_life = 2.5              # float
is_otc = True                # boolean
```

The four data types named in the syllabus:

| Type         | Meaning                           | Example            | Check with                                   |
|--------------|-----------------------------------|--------------------|----------------------------------------------|
| <b>int</b>   | Whole numbers                     | <b>500, -20</b>    | <b>type(500) → &lt;class 'int'&gt;</b>       |
| <b>float</b> | Decimal numbers                   | <b>2.5, 98.6</b>   | <b>type(2.5) → &lt;class 'float'&gt;</b>     |
| <b>str</b>   | Text in quotes (single or double) | <b>"Aspirin"</b>   | <b>type("Aspirin") → &lt;class 'str'&gt;</b> |
| <b>bool</b>  | Logical value, capitalised        | <b>True, False</b> | <b>type(True) → &lt;class 'bool'&gt;</b>     |

Use **type()** to check any variable's type: **type(dose\_mg)** returns **<class 'int'>**.

Trap alert: **True** and **False** start with capital letters. Writing **true** gives a **NameError** — a favourite small trap in exams.

## Type casting

Type casting means converting a value from one data type to another, using **int()**, **float()**, **str()** and **bool()**. It matters most with **input()**, which always returns a string — and you cannot do maths on strings.

```
weight = input("Enter weight in kg: ") # input() returns a string, e.g. "60"
weight = float(weight)                 # cast to float: 60.0
dose = weight * 10                     # now maths works: 600.0
print("Dose =", dose, "mg")
```

Rules worth memorising:

- **int(2.9)** gives **2** — it truncates, it does not round.
- **int("abc")** raises an error — only numeric-looking strings can be cast.
- **float("60")** gives **60.0**; **str(500)** gives **"500"**.

Exam tip: "Why does **input()** need type casting?" — the 2-mark answer: because **input()** always returns a string, and arithmetic on strings repeats text instead of calculating; casting with **int()** or **float()** converts the text to a number first.

## Operators

### Arithmetic operators

| Operator | Meaning                         | Example | Result | Pharma use                      |
|----------|---------------------------------|---------|--------|---------------------------------|
| +        | Addition                        | 5 + 2   | 7      | Total dose across tablets       |
| -        | Subtraction                     | 5 - 2   | 3      | Stock used                      |
| *        | Multiplication                  | 5 * 2   | 10     | Dose = weight × mg/kg           |
| /        | Division (always float)         | 5 / 2   | 2.5    | Dilution ratios                 |
| //       | Floor division (drops decimals) | 5 // 2  | 2      | Full strips from total tablets  |
| %        | Remainder (modulus)             | 5 % 2   | 1      | Checking even/odd batch numbers |
| **       | Power                           | 2 ** 3  | 8      | BMI uses height squared         |

### Comparison operators

These compare two values and return **True** or **False**: **==** (equal), **!=** (not equal), **>**, **<**, **>=**, **<=**.

Trap alert: **=** assigns a value (**dose = 500**); **==** compares two values (**dose == 500**). Mixing them up is the single most common coding error in this paper — examiners deliberately test it.

### Logical operators

**and**, **or**, **not** combine conditions:

```
age = 25
weight = 60
eligible = (age >= 18) and (weight >= 50)  # True only if BOTH are true
```

- **and** — **True** only when both sides are **True**.
- **or** — **True** when at least one side is **True**.
- **not** — flips the value: **not True** is **False**.

## Input and output operations

Output with **print()**:

```
drug = "Ibuprofen"
dose = 400
print(drug, dose)                # Ibuprofen 400
print(f"{drug} {dose}mg, twice daily") # Ibuprofen 400mg, twice daily
```

The second style is an f-string — put **f** before the quotes and **{...}** around any variable, and Python inserts its value. Use f-strings in every exam program; they make output neat and readable.

Input with **input()**:

```
name = input("Enter drug name: ")    # always a string
age = int(input("Enter age: "))      # cast immediately – safest habit
```

Remember: **input()** always returns a string. Cast it on the same line with **int()** or **float()** so you never forget.

## String operations and manipulation

Strings support a rich set of methods — you will reuse these in Unit III to clean messy dataset text:

```
s = "  Paracetamol 500mg "
```

```
s.strip()           # 'Paracetamol 500mg' - trims both ends
s.upper()           # '  PARACETAMOL 500MG '
s.lower()           # '  paracetamol 500mg '
s.replace("500mg", "650mg") # replaces one part
"a,b,c".split(",")   # ['a', 'b', 'c'] - splits into a list
"-".join(["2026", "09", "24"]) # '2026-09-24' - joins a list into one string
len("Aspirin")       # 7 - length
"Aspirin" + " 75mg"  # 'Aspirin 75mg' - concatenation
"Na" * 3             # 'NaNaN' - repetition
"para" in "paracetamol" # True - membership test
```

| Method                          | What it does                   |
|---------------------------------|--------------------------------|
| <b>strip()</b>                  | Removes spaces from both ends  |
| <b>upper()</b> / <b>lower()</b> | Converts case                  |
| <b>replace(old, new)</b>        | Substitutes part of the string |
| <b>split(sep)</b>               | Breaks a string into a list    |
| <b>join(list)</b>               | Joins a list into one string   |
| <b>len()</b>                    | Number of characters           |

Indexing starts at 0: "**Aspirin**"[0] is '**A**'. Slicing extracts a part: "**Paracetamol**"[0:4] is '**Para**' (start included, stop excluded).

Exam tip: String methods appear in both Unit I and Unit III. Learn the six methods in the table once; in Unit III the exam focus shifts to applying them to clean dataset text (e.g. stripping spaces and lowercasing drug names).

## Standard libraries vs third-party libraries

A library (module) is a collection of ready-made functions you import instead of writing yourself.

- Standard library — modules that ship with Python. Just **import** them; nothing to install. Examples: **math** (square roots, logarithms — handy for pH and half-life maths), **statistics** (mean, median), **datetime** (dates), **csv** (reading files), **random**, **os**.
- Third-party libraries — written by the community and installed separately with pip, Python's package installer. The three you will meet in this paper: NumPy (fast numerical arrays), Pandas (tables and datasets), Matplotlib (graphs).

```
pip install numpy
pip install pandas matplotlib
pip uninstall numpy      # removes a library
pip list                 # shows all installed libraries
```

Exam tip: "Differentiate between standard and third-party libraries" — one line each: standard libraries come pre-installed with Python and need only **import**; third-party libraries are installed separately using pip. Two examples each seals the answer.

## Very short questions (2 marks)

- 1 Expand IDE and name any two Python IDEs.
- 2 Why does **input()** need type casting? Give a one-line reason.
- 3 What is the difference between / and // in Python?
- 4 Write the pip command to install the Pandas library.

- 5 What is a third-party library? Give one example.

### Short questions (5 marks)

- 1 What is an IDE? Explain the advantages of IDEs over text editors with examples.
- 2 Explain the four Python data types with suitable pharma examples.
- 3 Differentiate between standard libraries and third-party libraries with examples.

### Long questions (10 marks)

- 1 Explain Python variables, data types and type casting with examples. Why is type casting necessary when taking user input?
- 2 Describe the arithmetic, comparison and logical operators in Python with examples. Highlight the common errors students make with `=` vs `==` and `/` vs `//`.

### How to approach these answers

- Very-short Q1: one-line expansion + two names; nothing more.
- Very-short Q2-Q5: one definition + one example or one command each.
- Short Q1: open with the IDE definition, give 4-5 table points, close with your choice (Jupyter for data work).
- Short Q2: table of four types with one example each; mention **type()**.
- Short Q3: two short definitions + two examples each side; mention pip.
- Long Q1: structure — intro (dynamic typing), variables + four types with examples, type casting with the **input()** example, truncation/error rules, conclusion.
- Long Q2: structure — intro, three operator groups with tables, a "common errors" subhead (`=` vs `==`, `/` vs `//`, **true** vs **True**), conclusion.

## Unit II — Control Structures & Functions

### Conditional statements

Conditional statements run different code depending on whether a condition is **True** or **False**. Python marks which lines belong to the condition with indentation (spaces at the start of the line) — there are no curly brackets like in other languages. Wrong indentation is a syntax error, not a style issue.

#### if, if-else, if-elif-else

```
age = 20

if age >= 18:
    print("Adult dose applicable")
else:
    print("Paediatric dose required")
```

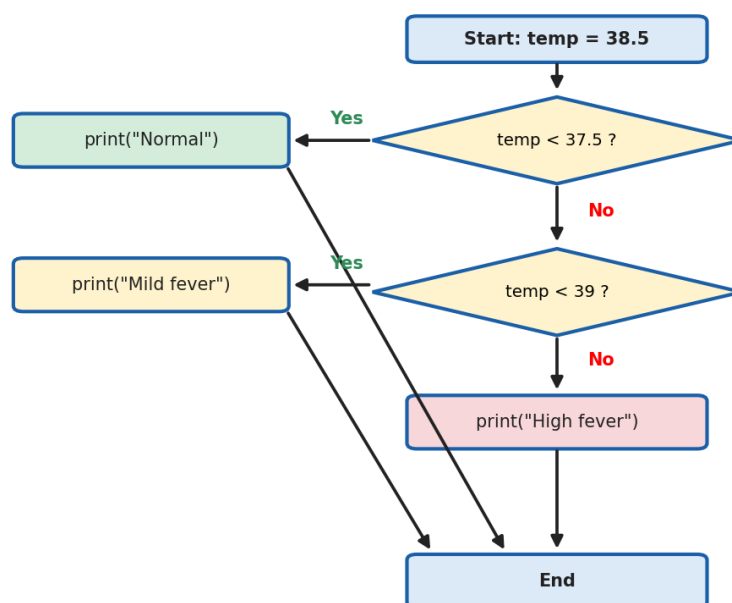
When there are more than two possibilities, chain them with **elif** (short for "else if"):

```
temp = 38.5

if temp < 37.5:
    print("Normal")
elif temp < 39:
    print("Mild fever - monitor")
else:
    print("High fever - intervene")
```

Only one block ever runs — the first whose condition is **True**. The rest are skipped entirely.

#### How if / elif / else executes (fever check example)



*Only ONE block runs — the first whose condition is True.*

How if / elif / else executes

### Nested conditions

An if statement inside another if statement is called nesting — use it when the second decision only matters after the first:

```

age = 65
has_allergy = False

if age >= 60:
    if has_allergy:
        print("Use alternative drug")
    else:
        print("Reduce dose - elderly patient")
else:
    print("Standard adult dose")

```

Exam tip: Examiners love asking you to "write a program to check patient eligibility for a drug based on age and weight using nested if." The pattern is always: outer **if** for the first criterion, inner **if-else** for the second. Practise writing it from memory.

## Loops

Loops repeat a block of code. Choose the loop by the situation:

- **for** loop — use when you know how many times to repeat (a fixed list, or numbers from **range()**).
- **while** loop — use when you repeat until something changes and you don't know the count in advance.

```

# for loop - over a fixed sequence
drugs = ["Aspirin", "Paracetamol", "Ibuprofen"]
for drug in drugs:
    print(drug)

for i in range(5):          # 0, 1, 2, 3, 4
    print("Tablet", i + 1)

```

**range()** comes in three flavours — memorise them:

| Call                   | Produces         | Used for                                |
|------------------------|------------------|-----------------------------------------|
| <b>range(5)</b>        | 0, 1, 2, 3, 4    | Repeat n times                          |
| <b>range(1, 6)</b>     | 1, 2, 3, 4, 5    | Numbered lists starting at 1            |
| <b>range(0, 24, 2)</b> | 0, 2, 4, ..., 22 | Stepped sequences (third number = step) |

```

# while loop - repeats until the condition becomes False
count = 3
while count > 0:
    print(f"{count} tablets remaining")
    count = count - 1    # without this line, the loop NEVER ends

```

Trap alert: A **while** loop whose condition never becomes **False** is an infinite loop — the program hangs forever. Always make sure something inside the loop moves the condition towards **False**. Examiners frequently ask "what is an infinite loop and how do you avoid it?"

## break and continue

| Statement       | What it does                                                  | Example situation                                    |
|-----------------|---------------------------------------------------------------|------------------------------------------------------|
| <b>break</b>    | Exits the loop immediately — no more rounds at all            | Stop searching the stock list once the drug is found |
| <b>continue</b> | Skips the rest of the current round and jumps to the next one | Skip expired batches but keep processing the rest    |

```
# break - stop searching once found
stock = ["Aspirin", "Paracetamol", "Ibuprofen", "Amoxicillin"]
for item in stock:
    if item == "Ibuprofen":
        print("Found Ibuprofen - stopping search")
        break
    print("Checking:", item)

# continue - skip expired batches, process the rest
batches = ["B101", "B102-expired", "B103", "B104-expired"]
for batch in batches:
    if "expired" in batch:
        continue          # skip this round only
    print("Processing batch:", batch)
```

Exam tip: "Differentiate between break and continue with an example" is a standard 3–5 mark question. One line each from the table, plus the two mini-programs above, and the answer is complete.

## Defining and calling functions

A function is a named, reusable block of code. You define it once with **def** and call it by name whenever you need it — this is the heart of modular programming.

```
def greet():
    print("Welcome to PharmacoLearn")

greet()    # calling the function
greet()    # call it again - no rewriting needed
```

## Passing arguments and returning values

```
def calculate_dose(weight_kg, mg_per_kg):
    dose = weight_kg * mg_per_kg
    return dose          # sends the result back to the caller

d = calculate_dose(60, 10)  # arguments: 60 and 10
print("Dose:", d, "mg")    # Output: Dose: 600 mg
```

Key vocabulary — examiners test these terms directly:

- Parameters are the names in the definition (**weight\_kg, mg\_per\_kg**).
- Arguments are the actual values you pass in (**60, 10**).
- return** sends a value back to the caller and ends the function immediately. A function without **return** gives back **None**.
- Default arguments fill in a value when the caller skips it: **def calculate\_dose(weight\_kg, mg\_per\_kg=10)**: lets you call **calculate\_dose(60)**.

Remember: Arguments go in, **return** sends a value out. If a question asks "what does this function return?", trace the **return** line.

## Modular programs for pharmaceutical applications

Modular means breaking a program into small functions, each doing one job — easier to read, test and reuse. The syllabus names two programs explicitly; learn both by heart, they are the most probable coding questions in the paper.

### Dosage calculation — Clark's rule

Clark's rule: Child dose = Adult dose × (weight in kg ÷ 70)

```
def clarks_dose(adult_dose_mg, weight_kg):
    """Calculate paediatric dose using Clark's rule."""
    return adult_dose_mg * (weight_kg / 70)

def main():
    adult_dose = float(input("Enter adult dose (mg): "))
    weight = float(input("Enter child weight (kg): "))
    child_dose = clarks_dose(adult_dose, weight)
    print(f"Child dose: {child_dose:.1f} mg")

main()
```

**{child\_dose:.1f}** formats the answer to one decimal place — clean output earns presentation marks.

### BMI calculation

BMI = weight (kg) ÷ height in metres squared. The WHO categories:

| BMI          | Category      |
|--------------|---------------|
| Below 18.5   | Underweight   |
| 18.5 – 24.9  | Normal weight |
| 25 – 29.9    | Overweight    |
| 30 and above | Obese         |

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

def bmi_category(bmi):
    if bmi < 18.5:
        return "Underweight"
    elif bmi < 25:
        return "Normal weight"
    elif bmi < 30:
        return "Overweight"
    else:
        return "Obese"

weight = float(input("Enter weight (kg): "))
height = float(input("Enter height (m): "))
bmi = calculate_bmi(weight, height)
print(f"BMI: {bmi:.1f} - {bmi_category(bmi)}")
```

Mnemonic: BMI cut-offs — 18.5 / 25 / 30. Think "under-18-and-a-half is underweight, 25 tips you into overweight, 30 into obese." Examiners like these numbers because they are standard WHO values.

### Very short questions (2 marks)

- 1 What is the role of indentation in Python?
- 2 Differentiate between **break** and **continue** in one line each.
- 3 State Clark's rule for paediatric dose calculation.
- 4 What does the **return** statement do in a function?
- 5 Write the formula for BMI as a Python expression.

### Short questions (5 marks)

- 1 Explain if, if-else and if-elif-else with a suitable pharma example.
- 2 Differentiate between **for** loop and **while** loop with examples. What is an infinite loop?
- 3 What is a function? Explain defining, calling, passing arguments and returning values with an example.

### Long questions (10 marks)

- 1 Write a modular Python program to calculate the paediatric dose of a drug using Clark's rule. Explain each function used.
- 2 Write a Python program to calculate BMI from user-entered weight and height, and classify it into WHO categories using if-elif-else.

### How to approach these answers

- Very-short Q1-Q5: one line each — definition, formula or one-line distinction.
- Short Q1: definition of conditional statements + the fever example with if-elif-else; mention indentation.
- Short Q2: table comparing the two loops + **range()** examples; define infinite loop with the fix.
- Short Q3: definition, then a **def/return** example; define parameters vs arguments in one line each.
- Long Q1: structure — state Clark's rule, show the full program, then explain each function (clarks\_dose, main) line by line, close with a sample run.
- Long Q2: structure — BMI formula, full program, WHO category table, sample run, one-line conclusion.

## Unit III — Data Structures & File Handling

### Lists, tuples and dictionaries

These three structures store collections of data — a list of drugs in stock, a drug's fixed identity, or one patient's full record.

List — ordered, changeable (mutable), written in square brackets:

```
drugs = ["Aspirin", "Paracetamol", "Ibuprofen"]
drugs.append("Amoxicillin")      # add at the end
drugs[1] = "Paracetamol 650mg"  # change an item - allowed in lists
```

Tuple — ordered, unchangeable (immutable), written in round brackets:

```
tablet = ("Aspirin", 75, "mg")   # name, strength, unit
# tablet[1] = 100    -> ERROR: tuples cannot be changed
```

Use a tuple when the data must never change by accident — like a drug's fixed (name, strength) identity.

Dictionary — stores key-value pairs, written in curly braces. Ideal for records:

```

patient = {"id": "P001", "age": 45, "drug": "Aspirin", "dose_mg": 75}
print(patient["drug"])      # Aspirin - accessed by KEY, not position
patient["dose_mg"] = 150    # update a value by its key

```

### Python data structures compared — list vs tuple vs dictionary

|             | LIST [ ]                   | TUPLE ( )                   | DICTIONARY { }                     |
|-------------|----------------------------|-----------------------------|------------------------------------|
| Ordered     | Yes                        | Yes                         | Yes                                |
| Changeable  | Yes (mutable)              | No (immutable)              | Yes (values)                       |
| Accessed by | index number               | index number                | key name                           |
| Example     | ["Aspirin", "Paracetamol"] | ("Aspirin", 75, "mg")       | {"drug": "Aspirin", "dose_mg": 75} |
| Pharma use  | stock list, drug names     | fixed (name, strength) pair | one patient's record               |

Exam favourite: 'Differentiate between list, tuple and dictionary with examples.' — this table is your answer.

### List vs tuple vs dictionary compared

Exam tip: "Differentiate between list, tuple and dictionary with examples" is a guaranteed-style 5-mark question. Memorise the figure above as a table: brackets, ordered?, changeable?, accessed by, one example each.

## Indexing and slicing lists

Indexing picks one element. Counting starts at 0, and negative numbers count from the end:

```

drugs = ["Aspirin", "Paracetamol", "Ibuprofen", "Amoxicillin"]
drugs[0]    # 'Aspirin'      (first item)
drugs[-1]   # 'Amoxicillin' (last item)

```

Slicing picks a range with **list[start:stop]** — the start is included, the stop is excluded:

```

drugs[0:2]   # ['Aspirin', 'Paracetamol']
drugs[1:]    # everything from index 1 onwards
drugs[:2]    # everything before index 2
drugs[:]     # a copy of the whole list

```

Trap alert: **drugs[0:2]** gives items 0 and 1 — the stop index is never included. Off-by-one errors here are a classic exam trap.

## Operations on lists and dictionaries

```
# Lists
stock = [120, 200, 150]
len(stock)          # 3 - number of items
stock.append(90)     # add one item at the end
stock.remove(200)    # remove by value
stock.sort()         # sort in place: [90, 120, 150]
"Aspirin" in drugs   # True - membership check

# Dictionaries
patient.keys()       # all keys
patient.values()     # all values
patient.items()      # key-value pairs
patient.get("allergy", "None") # safe lookup - returns "None" if key is missing
```

Remember: **remove()** deletes by value, indexing deletes by position (**del stock[0]**). And **dict.get(key, default)** never crashes on a missing key — use it instead of **patient["allergy"]** when you are not sure the key exists.

## String manipulation in data work

You learned string methods in Unit I; here they clean messy real-world text — for example drug names typed inconsistently in a dataset:

```
raw = " ASPIRIN 75mg "
clean = raw.strip().lower()          # 'aspirin 75mg'
clean.replace("75mg", "150mg")      # 'aspirin 150mg'
"aspirin,paracetamol".split(",")    # ['aspirin', 'paracetamol']

# Cleaning a whole list of drug names
raw_names = [" Aspirin ", "PARACETAMOL", "ibuprofen "]
clean_names = [n.strip().lower() for n in raw_names]
print(clean_names)                  # ['aspirin', 'paracetamol', 'ibuprofen']
```

Exam tip: The likely framing is "Write a program to clean a list of drug names (remove spaces, convert to lowercase)." The last three lines above are the answer.

## Introduction to NumPy arrays

NumPy is the standard third-party library for fast numerical work in Python. Its core object is the array — like a list, but every element must be the same type, and maths happens on all elements at once, with no loops:

```
import numpy as np    # the universal alias - always import it this way

conc = np.array([0, 5, 12, 20, 15, 8]) # drug concentrations
print(conc * 2)          # [ 0 10 24 40 30 16] - element-wise, no loop needed
print(conc.mean())       # 10.0
print(conc.max())        # 20
```

## Array creation

| Function                    | Result                                                          |
|-----------------------------|-----------------------------------------------------------------|
| <b>np.array([1, 2, 3])</b>  | Array from a list                                               |
| <b>np.zeros(5)</b>          | <b>[0. 0. 0. 0. 0.]</b>                                         |
| <b>np.ones(4)</b>           | <b>[1. 1. 1. 1.]</b>                                            |
| <b>np.arange(0, 10, 2)</b>  | <b>[0 2 4 6 8]</b> — like <b>range()</b> , but returns an array |
| <b>np.linspace(0, 1, 5)</b> | <b>[0. 0.25 0.5 0.75 1.]</b> — evenly spaced points             |

## Arithmetic and attributes

Arithmetic is element-wise: **a + b**, **a \* 2**, **a \*\* 2** apply to every element at once — perfect for scaling a whole column of concentrations. Useful attributes and methods:

```
arr = np.array([2, 4, 6, 8])
arr.shape    # (4,) - dimensions of the array
arr.dtype    # dtype('int64') - data type of elements
arr.size     # 4 - total number of elements
arr.mean()   # 5.0
arr.sum()    # 20
arr.min()    # 2
```

## NumPy arrays vs Python lists

| NumPy array                                     | Python list                           |
|-------------------------------------------------|---------------------------------------|
| Faster for large numerical data                 | Slower for numerical work             |
| Uses less memory                                | Uses more memory                      |
| Element-wise maths directly ( <b>conc * 2</b> ) | Needs a loop for the same calculation |
| All elements must be the same type              | Can mix types freely                  |

Exam tip: "Why is NumPy used in pharmaceutical data analysis?" — because concentration-time data, dissolution readings and assay values are large numerical datasets, and NumPy processes them element-wise without loops, faster and with less memory than lists.

## Reading and writing CSV files

A CSV (Comma-Separated Values) file is the simplest way to store tabular data — each line is a row, commas separate the columns. Excel opens CSVs, and every dataset in Units III–V uses this format.

```
import csv

# Writing a CSV file
with open("patients.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["PatientID", "Age", "Dose_mg"])
    writer.writerow(["P001", 45, 500])
    writer.writerow(["P002", 60, 250])

# Reading a CSV file
with open("patients.csv", "r") as f:
    reader = csv.DictReader(f)      # each row becomes a dictionary
    for row in reader:
        print(row["PatientID"], row["Dose_mg"])
```

Four points examiners expect you to know:

- 1 **with open(...)** closes the file automatically — no need for **f.close()**.
- 2 **newline=""** in write mode prevents blank lines on Windows.
- 3 **csv.DictReader** lets you access columns by name (**row["Dose\_mg"]**) instead of by index — far more readable.
- 4 **"w"** overwrites the file; **"r"** reads it; **"a"** would append.

## Understanding structured healthcare datasets

A structured dataset is data organised in rows and columns:

- Each row = one record (one patient, one tablet batch, one PK blood sample).
- Each column = one variable (age, dose, plasma concentration).

Example — a tiny pharmacokinetic (PK) dataset:

| Time_hr | Conc_ug_mL | Subject |
|---------|------------|---------|
| 0       | 0          | S01     |
| 1       | 5.2        | S01     |
| 2       | 12.1       | S01     |

Before analysing any dataset, check three things: what each column means, what the units are, and whether there are missing or impossible values (a negative age, a blank dose). That habit is exactly what "understanding structured healthcare datasets" means in the exam.

## Importing small pharmaceutical datasets — access and manipulation

Putting the unit together: load a small dataset and run core access-and-filter tasks on it.

```
import csv

with open("dissolution.csv", "r") as f:
    rows = list(csv.DictReader(f))

print("Batches:", len(rows))                # how many records
for r in rows:
    if float(r["Dissolution_%"]) < 80:      # core filtering step
        print("Below spec:", r["BatchID"], r["Dissolution_%"])
```

Note the **float(...)** — CSV values always arrive as strings, so cast before comparing, exactly as you learned in Unit I.

Exam tip: "Write a program to read a CSV file of dissolution data and display batches below 80%" — the code above is the complete answer. Swap the filename and column names and it fits any similar question.

### Very short questions (2 marks)

- 1 Which brackets are used for a list, a tuple and a dictionary?
- 2 What does **drugs[1:3]** return? Explain the slicing rule.
- 3 Why is a tuple preferred over a list for a drug's (name, strength) pair?
- 4 Write the NumPy code to create an array **[0, 2, 4, 6, 8]**.
- 5 What does **csv.DictReader** return for each row?

### Short questions (5 marks)

- 1 Differentiate between list, tuple and dictionary with suitable examples.
- 2 Explain array creation and arithmetic operations in NumPy with examples.
- 3 Write a Python program to read a CSV file and display records that meet a condition (e.g. dissolution below 80%).

### Long questions (10 marks)

- 1 Explain indexing and slicing in Python with examples. Describe the common operations performed on lists and dictionaries.
- 2 What is NumPy? Explain array creation, arithmetic operations and attributes of arrays. Compare NumPy arrays with Python lists.

### How to approach these answers

- Very-short Q1-Q5: one line each — brackets, slicing rule, immutability reason, one-liner code, "a dictionary per row".
- Short Q1: the comparison table (brackets, changeable, accessed by) + one example each.
- Short Q2: array-creation table + element-wise arithmetic example + one attribute line.
- Short Q3: write the full read-and-filter program; add one line on why **float()** is needed.
- Long Q1: structure — indexing (with negative indexes), slicing rule with diagram-like examples, list operations, dictionary operations.
- Long Q2: structure — what NumPy is and why, creation table, arithmetic examples, attributes, comparison table with lists, conclusion.

## Unit IV — Data Handling with Pandas

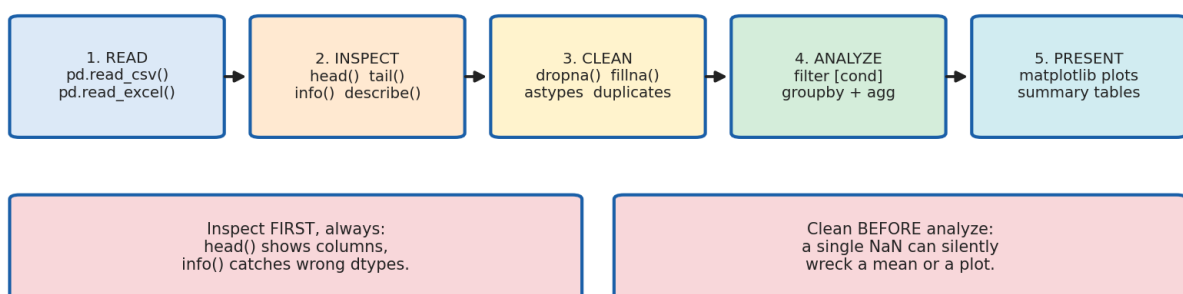
### Introduction to the Pandas library

Pandas is the most important Python library for working with tabular data — patient records, PK study results, ADR (Adverse Drug Reaction) reports. Where NumPy handles raw numbers, Pandas handles labelled tables: it knows your column names, tolerates missing values gracefully, and turns filtering and grouping into one-liners.

```
import pandas as pd  # the universal alias - always import it this way
```

Install with **pip install pandas**. Reading Excel files needs one extra package: **pip install openpyxl**.

#### Standard Pandas workflow: from raw file to result



Standard Pandas workflow: read, inspect, clean, analyze, present

Exam tip: If a question asks for "the steps in analysing a dataset with Pandas", the five-step workflow in the figure — read → inspect → clean → analyze → present — is a ready-made 5-mark answer.

## Pandas Series and DataFrame structures

Pandas has two core structures:

Series — a one-dimensional labelled array (think: one column of data):

```
doses = pd.Series([500, 250, 500, 750], index=["P001", "P002", "P003", "P004"])
print(doses["P002"]) # 250 - accessed by LABEL, not position
```

DataFrame — a two-dimensional table of rows and columns (the structure you will use almost all the time):

```
data = {"PatientID": ["P001", "P002", "P003"],
        "Age": [45, 60, 32],
        "Dose_mg": [500, 250, 500]}
df = pd.DataFrame(data)
print(df)
```

Remember: A DataFrame is a table; each column of a DataFrame is a Series. One line, and the 2-mark "differentiate Series and DataFrame" question is answered.

## Reading CSV and Excel files — PK study datasets and ADR reports

```
pk = pd.read_csv("pk_study.csv") # pharmacokinetic study data
adr = pd.read_excel("adr_reports.xlsx") # adverse drug reaction reports
```

Both functions return a DataFrame, ready for inspection. Two practical points examiners like to test:

- 1 The file path must be correct — keep the dataset in the same folder as your notebook to avoid path errors.
- 2 **read\_excel()** fails without the openpyxl library installed — a common error with a one-line fix (**pip install openpyxl**).

## Inspecting datasets — `head()`, `tail()`, `info()`, `describe()`

The first thing you do with any new dataset is look at it. These four functions are the standard inspection toolkit:

| Function                   | What it shows                                                                      |
|----------------------------|------------------------------------------------------------------------------------|
| <code>df.head()</code>     | First 5 rows — what do the columns look like? ( <code>df.head(10)</code> shows 10) |
| <code>df.tail()</code>     | Last 5 rows — anything odd at the end?                                             |
| <code>df.info()</code>     | Column names, data types, and non-missing counts — this is how you spot problems   |
| <code>df.describe()</code> | Summary statistics: count, mean, std, min, max (numeric columns only, by default)  |

- A column that should hold numbers but shows as **object** (text) in **info()** almost always means dirty data — fix it before analysis.
- **describe()** skipping text columns is normal behaviour, not an error.

Exam tip: "Explain any four functions used to inspect a dataset in Pandas" is a direct past-pattern question. The table above — one line per function — is the complete answer.

## Data cleaning techniques and handling missing values

Real datasets have gaps — a missed blood sample, an unrecorded age. Missing values appear as **NaN** (Not a Number) in Pandas, and you must handle them before analysis or your means and plots will be silently wrong.

```
df.isnull().sum()           # count missing values in each column
df.dropna()                 # remove rows containing any missing value
df.fillna(0)                # replace missing values with 0
df["Age"].fillna(df["Age"].mean()) # replace with the column's mean
df.drop_duplicates()        # remove duplicate rows
df["Dose_mg"] = df["Dose_mg"].astype(int) # fix a wrong data type
df["Drug"] = df["Drug"].str.strip().str.lower() # standardise text in a column
```

When to drop and when to fill:

- **dropna()** — when the missing rows are few and unimportant.
- **fillna()** — when you cannot afford to lose rows. In a small clinical dataset, dropping 3 of 20 patients hurts; filling with the mean is the textbook approach for numeric columns.

Other cleaning steps in the same family: fixing wrong data types with **astype()**, removing duplicates with **drop\_duplicates()**, and standardising text with **.str.lower()** / **.str.strip()**, which work on a whole column at once.

Trap alert: Never compute a mean or draw a plot before handling **NaN**. Pandas quietly propagates **NaN** through calculations — one missing value can turn a whole column's mean into **NaN** without any error message.

## Filtering and selecting data based on conditions

Filtering picks out rows that meet a condition, using boolean indexing:

```
adults = df[df["Age"] >= 18] # rows where Age >= 18
high_dose = df[(df["Dose_mg"] > 500) & (df["Age"] < 60)] # AND: &
elderly_aspirin = df[(df["Drug"] == "Aspirin") | (df["Age"] > 65)] # OR: |
```

Two rules that cause most student errors:

- 1 Parentheses around every condition — (**df["Age"] >= 18**) — are compulsory.
- 2 Use **&** for AND and **|** for OR — the words **and** / **or** do not work inside Pandas filtering.

Selecting picks columns or specific cells:

```
df["Dose_mg"] # one column (a Series)
df[["Drug", "Dose_mg"]] # several columns (a DataFrame)
df.loc[0, "Drug"] # row labelled 0, column "Drug"
df.iloc[0:3, 0:2] # first 3 rows, first 2 columns - by POSITION
```

|            | loc                       | iloc                     |
|------------|---------------------------|--------------------------|
| Selects by | labels (row/column names) | integer positions        |
| Example    | <b>df.loc[0, "Drug"]</b>  | <b>df.iloc[0:3, 0:2]</b> |

Exam tip: "Differentiate between loc and iloc" — labels vs positions, one example each. Two lines, two marks.

## Grouping data and performing aggregation functions

Grouping splits the data by a category, then computes a summary for each group — e.g. average dose per drug, or total ADRs per drug:

```
df.groupby("Drug")["Dose_mg"].mean() # mean dose for each drug
df.groupby("Drug")["ADR_Reported"].sum() # total ADRs reported per drug

df.groupby("Drug").agg({"Dose_mg": "mean", # several summaries at once
                       "Age": "max",
                       "PatientID": "count"})
```

Common aggregation functions: **mean()**, **sum()**, **count()**, **max()**, **min()**, **std()**.

Exam tip: "An ADR dataset has columns Drug and ADR\_Reported. Write Pandas code to find the total ADRs for each drug." The one-line answer:

**df.groupby("Drug")["ADR\_Reported"].sum()**. Group-then-aggregate is the standard route to every "per drug / per group" question.

### Very short questions (2 marks)

- 1 Differentiate between a Pandas Series and a DataFrame in one line.
- 2 Name any four functions used to inspect a dataset in Pandas.
- 3 What does **df.isnull().sum()** tell you?
- 4 Differentiate between **loc** and **iloc**.
- 5 Write the Pandas code to find the mean dose for each drug in a DataFrame **df** with columns **Drug** and **Dose\_mg**.

### Short questions (5 marks)

- 1 Explain the functions **head()**, **tail()**, **info()** and **describe()** used for inspecting datasets.
- 2 How are missing values handled in Pandas? Explain with examples.
- 3 Explain filtering and selecting data in Pandas. What is the difference between **&** and the word **and** here?

### Long questions (10 marks)

- 1 Describe the standard workflow of handling a pharmaceutical dataset (e.g. an ADR report) in Pandas: reading the file, inspection, cleaning, filtering and grouping, with code for each step.
- 2 Explain data cleaning techniques in Pandas. When should missing values be dropped and when should they be filled? Illustrate with examples.

### How to approach these answers

- Very-short Q1-Q5: one line each — Series vs DataFrame, four function names, "missing-value counts per column", labels vs positions, the groupby one-liner.
- Short Q1: the four-function table; add what **info()** reveals about dtypes.
- Short Q2: find with **isnull().sum()**, then **dropna()** vs **fillna()** with the when-to-use rule.
- Short Q3: boolean-indexing example + the two error rules (parentheses, **&/|** not **and/or**); **loc** vs **iloc** table.
- Long Q1: structure — five subheads matching the workflow figure, one code snippet per step, close with the ADR-per-drug groupby line.
- Long Q2: structure — what missing values are (**NaN**), detection, drop vs fill with the small-dataset reasoning, other cleaning (**astype**, duplicates, **.str** methods), conclusion.

# Unit V — Data Visualization with Matplotlib

## Introduction to Matplotlib

Matplotlib is Python's core plotting library — it turns analysed data into graphs. In pharmacy, graphs are not decoration: a concentration–time curve is the result of a PK study, and a dissolution profile is how you compare formulations.

```
import matplotlib.pyplot as plt # the standard alias
```

The pattern behind every Matplotlib figure is the same three steps:

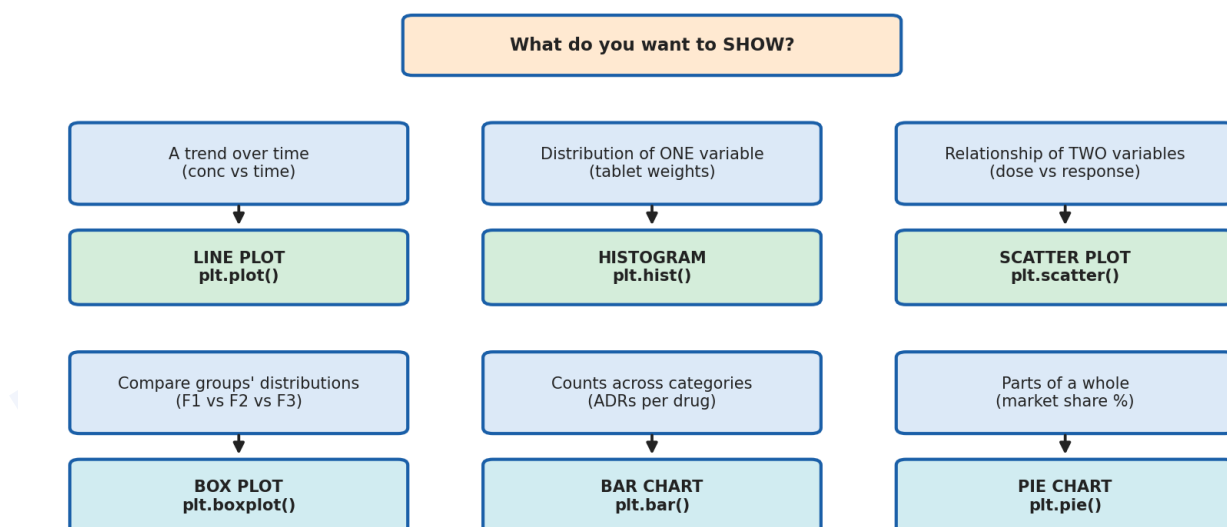
```
plt.plot([0, 1, 2, 3], [0, 5, 12, 8]) # 1. create the plot
plt.show()                          # 2. display the figure
```

In Jupyter Notebook, plots appear directly below the cell, so **plt.show()** is optional but harmless.

Remember: Every Matplotlib answer follows plot → label → show. Forget the labels and you lose marks even if the code is right.

## Creating line plots, histograms, scatter plots and box plots

### Which Matplotlib plot should I use? — decision chart



Always add: xlabel + ylabel (with units), title, legend, grid.

Which Matplotlib plot should I use? — decision chart

## Line plot — trends over a continuous variable

Use it when the x-axis is ordered and continuous, usually time. The workhorse of pharma graphs.

```
time = [0, 1, 2, 4, 8, 12]
conc = [0, 5, 12, 18, 10, 4]
plt.plot(time, conc)
plt.show()
```

## Histogram — distribution of one variable

Use it to see how values spread out. Example: the weights of 100 tablets — you want them clustered tightly around the target weight.

```
weights = [498, 500, 501, 499, 502, 500, 497, 501, 500, 499]
plt.hist(weights, bins=5)
plt.show()
```

**bins** controls how many bars the range is split into: too few hides detail, too many turns the picture into noise.

## Scatter plot — relationship between two variables

Use it for one dot per observation to reveal a relationship. Example: dose vs percentage pain relief.

```
dose = [100, 200, 300, 400, 500]
relief = [20, 35, 55, 70, 85]
plt.scatter(dose, relief)
plt.show()
```

Dots trending upward mean the variables are positively related — the visual basis of dose-response.

## Box plot — comparing distributions across groups

Use it to compare one variable across groups in a single view. Example: dissolution percentages of three formulations side by side.

```
f1 = [82, 85, 88, 84, 86]
f2 = [70, 72, 68, 75, 71]
f3 = [90, 92, 89, 91, 93]
plt.boxplot([f1, f2, f3], labels=["F1", "F2", "F3"])
plt.show()
```

How to read a box plot:

- The line inside the box = the median.
- The box = the middle 50% of values (interquartile range).

- The whiskers = the overall spread.
- Dots beyond the whiskers = possible outliers worth investigating.

Exam tip: "Differentiate between a histogram and a bar chart" — a histogram shows the distribution of one continuous variable split into bins; a bar chart compares values across categories. They look similar but answer different questions.

## Labelling axes, titles and legends

An unlabeled graph earns no marks — in the exam or in a real report. The checklist:

```
plt.plot(time, conc, label="Formulation A") # label= names the line
plt.plot(time, conc2, label="Formulation B")
plt.xlabel("Time (hours)") # x-axis WITH units
plt.ylabel("Plasma conc. (µg/mL)") # y-axis WITH units
plt.title("Dissolution profile comparison")
plt.legend() # displays the label= names
plt.grid(True) # light grid - easier to read values off
plt.show()
```

Finish-every-plot checklist: x-label with units, y-label with units, title, legend (when more than one series), grid.

## Visualising pharmaceutical datasets

### Concentration-time curves: oral vs IV administration

The syllabus explicitly asks for this — learn the code and the reading of the curves.

```
time = [0, 1, 2, 4, 8, 12]
conc_oral = [0, 5, 12, 18, 10, 4] # rises to a peak, then falls
conc_iv = [25, 18, 12, 7, 3, 1] # starts highest, falls immediately

plt.plot(time, conc_oral, marker="o", label="Oral")
plt.plot(time, conc_iv, marker="s", label="IV bolus")
plt.xlabel("Time (hours)")
plt.ylabel("Plasma concentration (µg/mL)")
plt.title("Concentration-time curves: oral vs IV")
plt.legend()
plt.grid(True)
plt.show()
```

How to read the two curves:

- The oral curve rises first (absorption phase) to a peak, then declines (elimination phase). The peak height is  $C_{max}$ ; the time it occurs is  $T_{max}$ .
- The IV bolus curve starts at its maximum at time zero — there is no absorption — and only declines.

- The area under either curve (AUC) relates to total drug exposure.

Mnemonic: Oral = up then down (absorb, then eliminate). IV = down only (no absorption to wait for). If you can sketch this from memory, the 5-mark question is half done.

## ADR reporting rates across drugs

When comparing counts across categories, a bar chart is the natural choice:

```
drugs = ["Aspirin", "Ibuprofen", "Paracetamol"]
adr_counts = [45, 30, 12]
plt.bar(drugs, adr_counts)
plt.xlabel("Drug")
plt.ylabel("Number of ADR reports")
plt.title("ADR reporting rates across drugs")
plt.show()
```

Exam tip: The syllabus lists line, histogram, scatter and box plots as the types to learn, but bar charts are the standard real-world choice for counts across categories like drugs. Know **plt.bar()** as well — examiners accept it for ADR-rate questions.

## Dissolution profiles

A dissolution profile plots percentage of drug dissolved vs time, one line per formulation — the same labelled line-plot pattern as the concentration-time example, with formulations A and B distinguished by **label=** and a legend. This is how a test formulation is visually compared against a reference.

---

## Scientific interpretation of plots

Examiners want the code and the reading. Practise these four interpretations:

- Concentration-time curve: Point out C<sub>max</sub> and T<sub>max</sub> on the oral curve; explain why the IV curve has no absorption phase; state which route gives faster/higher exposure.
- Histogram (tablet weights): A narrow, symmetric histogram around the target weight means good content uniformity; a wide or skewed one signals a manufacturing problem.
- Scatter plot (dose vs response): An upward trend supports a dose-response relationship; scattered dots with no pattern mean no clear relationship.
- Box plot (formulations): Compare medians (which formulation dissolves most?), box heights (which is most consistent?), and outliers (which batch needs investigation?).

Exam tip: A strong interpretation is 2–3 sentences: what you see, what it means scientifically, what action it suggests. That structure fits every "interpret the plot" question in this unit.

### Very short questions (2 marks)

- 1 Which Matplotlib function draws a line plot? A histogram?
- 2 What do **plt.xlabel()** and **plt.ylabel()** do? What must they always include?
- 3 Define C<sub>max</sub> and T<sub>max</sub> with reference to a concentration-time curve.
- 4 In a box plot, what does the line inside the box represent?
- 5 Why does an IV bolus concentration-time curve have no absorption phase?

### Short questions (5 marks)

- 1 Explain the four plot types in Matplotlib (line, histogram, scatter, box) and state when each is used.
- 2 Write the code to plot concentration-time data for oral and IV administration with proper labels, title and legend.
- 3 How do you read and interpret a box plot? Explain with a formulation-comparison example.

### Long questions (10 marks)

- 1 Plot the given concentration-time data for oral and IV administration using Matplotlib, with labelled axes, title and legend. Interpret both curves scientifically (C<sub>max</sub>, T<sub>max</sub>, absorption, AUC).
- 2 Explain data visualization of pharmaceutical datasets with Matplotlib: concentration-time curves, ADR reporting rates and dissolution profiles. Include the labelling checklist for every plot.

### How to approach these answers

- Very-short Q1-Q5: one line each — function names, "axis labels with units", C<sub>max</sub>/T<sub>max</sub> definitions, "the median", "no absorption for IV".
- Short Q1: one paragraph per plot type — what it shows, the function, one pharma example.
- Short Q2: the full labelled code block; mention **label=** and **plt.legend()**.
- Short Q3: box-plot anatomy (median, box, whiskers, outliers) + what each tells you about formulations.
- Long Q1: structure — code block, labelled-curve sketch description, interpretation under three subheads (oral curve, IV curve, comparison), conclusion.
- Long Q2: structure — intro, three dataset types with code snippets, the labelling checklist, interpretation lines, conclusion.

---

---

## Quick revision checklist (all units)

- [ ] Unit I: IDE advantages, 4 data types, type casting, operator tables, string methods, pip install/uninstall
- [ ] Unit II: if/elif/else, for vs while, break vs continue, function with **return**, Clark's rule program, BMI program + cut-offs 18.5/25/30
- [ ] Unit III: list vs tuple vs dictionary table, indexing/slicing, NumPy array creation and element-wise maths, CSV read/write pattern
- [ ] Unit IV: Series vs DataFrame, **read\_csv/read\_excel**, **head/tail/info/describe**, **dropna/fillna**, boolean filtering, **groupby** + aggregation, **loc** vs **iloc**
- [ ] Unit V: four plot types and when to use each, labelling checklist, reading Cmax/Tmax, interpreting histograms, scatter plots and box plots

Write the code by hand at least once per unit — for this 30-mark paper, the programs (Clark's rule, BMI, CSV filtering, Pandas groupby, concentration-time plot) are where the marks live.