



Notes by PharmacoLearn

Basics of Python Programming for Pharmaceutical Sciences

BP101T — B.Pharm Semester I

Concept Mastery Notes

Deep concepts for GPAT & competitive exams — nothing missed

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

Unit I — Introduction to Python programming

Why Python is in your pharmacy syllabus

Every modern pharmacy runs on data: dissolution readings, stability samples, ADR reports, prescription audits, clinical trial tables. Python became the language pharmacists and researchers reach for because it reads almost like English, and because its free libraries (NumPy, Pandas, Matplotlib — your Units III to V) handle the exact jobs pharma data needs: fast number-crunching, labelled tables, and publication-ready graphs. No licence fees, no paid toolboxes — a laptop and a free download is the whole setup. That is also why examiners love this subject: almost every question can be tied to a small, realistic pharmacy task.

Installing Python and choosing an IDE

Python is a high-level, interpreted language. Interpreted means there is no separate compilation step — Python reads your code and executes it line by line, immediately. The practical consequence: you can test one line of a dose calculation, see the answer, fix it, and move on. That instant feedback loop is exactly why interpreted languages dominate data work.

Installing: download the installer from python.org, run it, and on Windows tick "Add Python to PATH" — this lets you type **python** in any terminal. Verify with:

```
python --version
```

An IDE (Integrated Development Environment) bundles a code editor with a debugger, an integrated terminal, autocomplete, and package management in one window. The syllabus names three:

- Jupyter Notebook — runs in your browser; code executes in small blocks called cells, with output, tables and plots appearing directly below each cell. This is your workhorse for Units III-V.
- PyCharm — a full IDE built only for Python. Strong autocomplete, visual debugger, project management. Heavier download.
- VS Code (Visual Studio Code) — a fast, lightweight editor with Python extensions. The popular middle ground.

Why an IDE beats a plain text editor (Notepad, Notepad++):

IDE	Plain text editor
Syntax highlighting and autocomplete as you type	No code assistance at all
Built-in debugger — pause code and inspect variables line by line	You debug by guessing and re-running

IDE	Plain text editor
Integrated terminal and one-click package installer	You juggle separate tools
Warnings appear before you run anything	Errors surface only at run time

Trap alert: In Jupyter, all cells share one kernel (one running Python session). Variables created in cell 3 are visible in cell 1 if you run cell 1 afterwards. Running cells out of order is the single most common source of "but it worked a minute ago" confusion. If outputs stop making sense, restart the kernel and run cells top to bottom.

GPAT angle: "What is an IDE? Explain its advantages over a text editor with examples" is a repeating 5-mark question. Memorise the table above and name Jupyter, PyCharm and VS Code.

Variables and data types

A variable is a named label stuck onto a value. The mechanism worth understanding: Python variables do not contain values the way boxes contain objects — they are references (think name-tags) pointing at objects stored in memory. Two names can point at the same object, which is why some "copies" in Python are not really copies (you will meet this properly with lists in Unit III). Check where a name points with `id()`.

Python is dynamically typed: you never declare a type; Python attaches the type to the value, not the variable. The same name can point at an integer in one line and a string in the next. Convenient, but it means type mistakes surface at run time — hence type casting, below.

```
drug_name = "Paracetamol"    # str
dose_mg = 500                # int
half_life = 2.5              # float
is_otc = True                # bool
```

The four syllabus data types:

- int — whole numbers: **500**, **-20**, **0**. Tablet counts, patient ages.
- float — decimal numbers: **2.5**, **98.6**. Doses in mg/kg, half-lives, concentrations.
- str — text in quotes: **"Aspirin"**, **'500mg'**. Drug names, patient IDs, batch numbers.
- bool — **True** or **False** (capital first letter — **true** is an error). Answers to yes/no questions: is the patient eligible? is the batch expired?

Check any type with `type()`: `type(dose_mg)` returns `<class 'int'>`.

Trap alert: **True** and **False** must be capitalised. Writing **true** gives a **NameError**. Examiners slip this into "find the error" snippets regularly.

Type casting

Type casting converts one type into another. The reason it exists in every syllabus: **input()** always returns a string, because keyboards produce text. Python cannot do arithmetic on text, so **"60" * 10** would repeat the characters sixty times rather than multiply.

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

The casting functions: **int()**, **float()**, **str()**, **bool()**. Three behaviours to memorise:

- **int(2.9)** gives **2** — it truncates toward zero, it does not round.
- **int("abc")** and **int("2.9")** raise errors — a string must look like a valid integer. Convert **"2.9"** with **float()** first.
- **bool()** of an empty string, **0**, or an empty list is **False**; almost everything else is **True**. This "truthiness" idea returns in Unit II conditions.

Trap alert: Forgetting to cast **input()** is the most common crash in beginner programs: **TypeError: can only concatenate str (not "int") to str**. Whenever a program reads numbers from the user, the cast must happen before any calculation.

Operators

Arithmetic operators

Operator	Meaning	Example	Result	Pharma use
+	Addition	5 + 2	7	Total tablets across strips
-	Subtraction	5 - 2	3	Stock after dispensing
*	Multiplication	5 * 2	10	Dose = weight × mg/kg
/	Division — always float	5 / 2	2.5	Dose per kg
//	Floor division — drops the fraction	5 // 2	2	Full strips from loose tablets
%	Modulus — the remainder	5 % 2	1	Leftover tablets; even/odd batch checks

Operator	Meaning	Example	Result	Pharma use
**	Power	2 ** 3	8	BMI (height squared), exponential decay

Precedence follows school BODMAS, with ****** binding tightest and evaluating right-to-left: **2 ** 3 ** 2** is **2 ** 9 = 512**, not **64**. When in doubt, use brackets — examiners accept them, and they prevent silent mistakes.

Why does **/** always return a float even for **4 / 2**? Python 3 made division "true division" deliberately, because integer division was silently throwing away fractions in scientific code — exactly the kind of rounding that must never happen in dose maths. **//** exists for the cases where you genuinely want the whole part.

Comparison operators

== != > < >= <= — each answers a yes/no question and returns a bool. Two points that earn marks:

- =** assigns; **==** compares. **if dose = 500:** is a syntax error; **if dose == 500:** is a test. Never mix them.
- Comparisons chain in Python: **18 <= age <= 65** works and reads like the maths. It means **(18 <= age) and (age <= 65)**.

Logical operators

and, or, not combine conditions:

```
age = 25
weight = 60
eligible = (age >= 18) and (weight >= 50)    # True only if both hold
```

The mechanism that matters: **and/or** short-circuit. In **A and B**, if **A** is **False**, Python never evaluates **B** — the answer is already decided. This is why you can safely write **if batch_list and batch_list[0] == "B101":** — the second half never runs on an empty list, so no crash. **not** simply flips a bool.

Trap alert: **and/or** return one of their operands, not always **True/False**. **0 or 500** gives **500**. In an **if** this behaves like a bool, but if you store the result, you may be storing a number, not a bool.

Input and output operations

Output with **print()**: it accepts multiple items separated by commas, and f-strings (the **f"..."** prefix) embed expressions directly inside text — the expression in **{ }** is evaluated and its result inserted:

```

drug = "Ibuprofen"
dose = 400
print(f"{drug} {dose}mg, twice daily") # Ibuprofen 400mg, twice daily
print(f"BMI: {23.456:.1f}")           # BMI: 23.5 (one decimal place)

```

The `:.1f` format is worth memorising: doses and lab values are always reported to fixed decimals, never as long float tails.

Input with **input()**: displays a prompt, waits, and returns the typed text as a string. Pattern for every numeric input program: **float(input("..."))** or **int(input("..."))** in one line — prompt, read, and cast together.

String operations and manipulation

Strings support a rich method set. The mechanism to internalise first: strings are immutable. No string method changes the string in place — each returns a new string. So **s.upper()** without assignment throws the result away:

```

s = " Paracetamol 500mg "

s.strip()      # 'Paracetamol 500mg' – spaces trimmed from both ends
s.upper()      # ' PARACETAMOL 500MG '
s.lower()      # ' paracetamol 500mg '
s.replace("500mg", "650mg") # swaps one part for another
"a,b,c".split(",")          # ['a', 'b', 'c'] – string -> list
"-. ".join(["2026", "09", "24"])# '2026-09-24' – list -> string
len("Aspirin")              # 7
"Aspirin" + " 75mg"         # concatenation -> 'Aspirin 75mg'
"Na" * 3                    # repetition -> 'NaNana'
"para" in "paracetamol"     # True – membership test

```

Indexing starts at 0: **"Aspirin"[0]** is **'A'**; negative indices count from the end (**"Aspirin"[-1]** is **'n'**). Slicing extracts a portion — **"Paracetamol"[0:4]** is **'Para'** (full slicing mechanics, including why the stop index is excluded, are covered in Unit III).

Why this matters beyond the unit: real datasets are messy — **"ASPIRIN 75mg "**, **"Aspirin"**, **"aspirin"** all meaning the same drug. The standard cleaning chain is **raw.strip().lower()**, and you will use exactly this in Units III and IV before grouping or comparing drug names. Methods chain left to right, each returning a new string for the next to work on.

Trap alert: **s.strip()** on its own line does nothing visible — you must write **s = s.strip()** or use the result directly. Forgetting the assignment is one of the most common silent bugs in string-cleaning programs.

GPAT angle: "Write the output of: **s = 'Aspirin'; print(s[1:4])**" — slicing questions are favourites. Answer: **'spi'** (indices 1, 2, 3; index 4 excluded).

Standard libraries vs third-party libraries

Python ships with "batteries included": the standard library is a set of modules that come pre-installed — you only **import** them:

- **math** — **sqrt()**, **log10()** ($\text{pH} = -\log_{10}[\text{H}^+]$), **pow()**, constants like **pi**. First choice for half-life and pH maths.
- **statistics** — **mean()**, **median()**, **stdev()** for quick lab-data summaries.
- **csv** — reading/writing CSV files (your Unit III file work, no pip needed).
- **datetime** — dates and timestamps for batch records.
- **random**, **os** — sampling, file paths.

Third-party libraries are written by the community, hosted on PyPI (the Python Package Index), and installed with pip, Python's package installer:

```
pip install numpy
pip install pandas matplotlib
pip uninstall numpy
pip list
```

Your pharma toolkit for the rest of this subject: NumPy (fast numerical arrays), Pandas (labelled tables), Matplotlib (graphs). Reading Excel files later needs **openpyxl** too. The exam distinction is one line: standard = comes with Python, just import; third-party = installed separately via pip.

A step beyond the syllabus that saves real pain: a virtual environment (**python -m venv myenv**) is an isolated Python installation per project, so the library versions for your PK analysis never clash with another project's. And a practical trap —

Trap alert: If **pip install pandas** succeeds but **import pandas** fails in Jupyter, you have two Pythons: pip installed into one, Jupyter runs another. Fix it from inside the notebook with **%pip install pandas** (the % runs pip against the kernel's own Python). This mismatch wastes hours in every first-year lab.

Very short questions (2 marks)

- 1 What is an IDE? Name the three IDEs in your syllabus.
- 2 Why does **input()** always require type casting before arithmetic?
- 3 State the difference between / and // with one example each.
- 4 What is the output of **int(2.9)**? Does it round or truncate?
- 5 Distinguish between a standard library and a third-party library with one example each.

Short questions (5 marks)

- 1 Explain the advantages of IDEs over plain text editors with suitable examples.

- 2 Describe the four data types in Python (int, float, str, bool) with pharmacy examples and show how **type()** identifies them.
- 3 Explain string immutability and demonstrate any five string manipulation methods with examples.

Long questions (10 marks)

- 1 Explain the steps to install Python and set up an IDE. Compare Jupyter Notebook, PyCharm and VS Code, and discuss why IDEs are preferred over text editors for pharmaceutical data work.
- 2 Describe arithmetic, comparison and logical operators in Python with examples. Explain operator precedence, the = vs == distinction, and short-circuit evaluation of **and/or** with a pharmacy example.

How to approach these answers

- Very-short Q1: one-line definition of IDE + the three names; nothing more.
- Very-short Q2: state that **input()** returns a string + what goes wrong without casting (one example).
- Very-short Q3: one line each — / true division always float, // floor division; give **5/2** vs **5//2**.
- Very-short Q4: answer **2**, and the word "truncates" — that word earns the mark.
- Very-short Q5: one line each — standard comes with Python (**math**), third-party needs pip (**pandas**).
- Short Q1: open with the IDE definition, then 4–5 contrast points (debugging, autocomplete, terminal, warnings), close with one line on which IDE suits data work.
- Short Q2: define dynamic typing first, then each type with a pharmacy example and a **type()** line.
- Short Q3: state immutability (methods return new strings), then five methods with input → output shown.
- Long Q1: structure — installation steps, one paragraph per IDE, comparison table, IDE-vs-editor advantages, closing line linking to Units III–V data work.
- Long Q2: three operator groups with tables/examples, precedence with a worked example, = vs == trap, short-circuit with the empty-list safety example.

Unit II — Control Structures & Functions

Conditional statements: making decisions

Real pharmacy logic is full of decisions: adult dose or paediatric dose, dispense or refer, normal or abnormal lab value. Conditional statements let a program choose between code paths based on whether a condition is **True** or **False**:

```
age = 20

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

The first thing to absorb is indentation as syntax. Python has no curly brackets; the block belonging to **if** is defined purely by being indented further. This was a deliberate design choice — it forces code to be visually structured, which is why Python reads like pseudocode. The price: inconsistent indentation is a syntax error, not a style complaint.

With more than two outcomes, **elif** (else-if) chains the tests:

```
temp = 38.5

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

The mechanism that makes **elif** efficient: Python tests conditions top to bottom and stops at the first **True**. Later conditions are never evaluated. So order your chain from the most specific to the most general — and note the difference from writing three separate **if** statements, where every condition would be tested regardless.

Nested conditions place one decision inside another, for two-dimensional logic (age × allergy):

```
age = 65
has_allergy = False

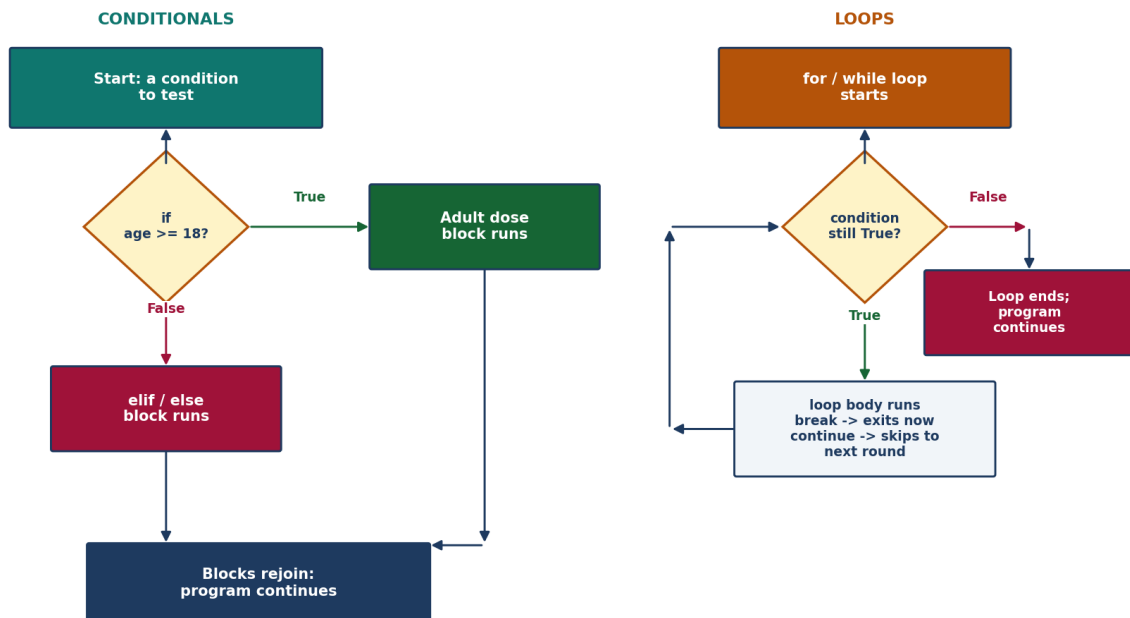
if age >= 60:
    if has_allergy:
        print("Use alternative drug")
    else:
        print("Reduce dose - elderly patient")
else:
    print("Standard adult dose")
```

Deep nesting gets hard to read fast; beyond two levels, consider combining conditions with **and/or** instead.

A quiet superpower: conditions accept any value, not just comparisons, because of truthiness (Unit I). **if batch_list:** means "if the list is non-empty"; **if name:** means "if the name isn't an empty string". Examiners increasingly test this: **if 0:** never runs its block.

Trap alert: Every **if/elif** line must end with a colon, and the block must be indented consistently (4 spaces is the convention). Missing colons and mixed tabs/spaces are the two most common syntax errors in exam code.

Python Control Structures: Decisions and Repetition



Flowchart of Python control structures: conditionals branch on True/False, loops repeat while the condition holds, with `break` and `continue` paths

Loops: repeating work

Loops repeat a block. Choose by what you know: a **for** loop when the repetitions are countable in advance, a **while** loop when you repeat until something changes.

The for loop

```
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() generates the numbers: **range(5)** → 0–4; **range(1, 6)** → 1–5; **range(0, 24, 2)** → 0, 2, 4, ..., 22 (start, stop, step). Two facts worth understanding deeply:

- 1 **range()** stops one before the stop value. This is not arbitrary — it follows from zero-based indexing (Unit III): **range(len(drugs))** gives exactly the valid indices **0** to **len-1**. Once you see that, off-by-one errors become predictable instead of mysterious.

- 2 **range()** is lazy: **range(1000000)** does not build a million-number list in memory; it generates each number on demand. That is why looping a million dissolution readings with **range** costs almost no memory.

The while loop

```
count = 3
while count > 0:
    print(f"{count} tablets remaining")
    count = count - 1 # without this line, the loop never ends
```

A **while** loop re-tests its condition before every round. If nothing inside the body moves the condition toward **False**, you get an infinite loop — the program hangs, and in Jupyter you stop it with the kernel interrupt button (■). Classic exam framing: "the program hangs; identify the missing update statement."

Typical **while** uses: reading batches until stock runs out, waiting for valid user input:

```
age = int(input("Enter age: "))
while age <= 0:
    print("Age must be positive.")
    age = int(input("Enter age: "))
```

break and continue

- **break** exits the loop immediately — no more rounds, jump to whatever follows the loop.
- **continue** skips the rest of the current round and jumps to the next round's test.

```
# Stop searching once found - no point checking the rest
stock = ["Aspirin", "Paracetamol", "Ibuprofen", "Amoxicillin"]
for item in stock:
    if item == "Ibuprofen":
        print("Found Ibuprofen - stopping search")
        break
    print("Checking:", item)

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

Why **break** matters beyond convenience: searching a 10,000-row stock list and stopping at row 5 instead of scanning all 10,000 is a genuine performance habit. A step beyond the syllabus: loops can have an **else** clause that runs only if the loop finished without **break** — the standard pattern for "searched everything, didn't find it".

Trap alert: **break** exits only the innermost loop. In nested loops (checking every batch against every drug), a **break** in the inner loop does not stop the outer one — a favourite "what does this print?" trap.

GPAT angle: "Differentiate between break and continue with an example" is asked almost every year. One crisp line each plus the batch-search/expired-batch examples above is a full-marks answer.

Functions: defining, calling, arguments, return values

A function is a named, reusable block of code. You define it once with **def** and call it by name whenever needed:

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

greet()
greet()    # reused - never rewritten
```

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)
print("Dose:", d, "mg") # Dose: 600 mg
```

Terminology examiners test: parameters are the names in the definition (**weight_kg**); arguments are the values passed at the call (**60**). **return** does two things at once — it hands a value back and ends the function immediately; any code after **return** never runs. A function with no **return** gives back **None**.

Arguments can have defaults, which must come last:

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

calculate_dose(60)        # 600 - default used
calculate_dose(60, 15)    # 900 - default overridden
```

Why functions matter: scope and the call mechanism

When a function runs, Python creates a temporary workspace for its variables; when it returns, that workspace is discarded. Consequence: variables created inside a function are invisible outside it (local scope). This is a feature — two functions can both use a variable named **dose** without colliding. The classic trap is the reverse: trying to use a function's internal variable after the call ends gives a **NameError**.

print() vs **return** is the conceptual trap of this whole unit: **print** displays a value for humans; **return** hands a value to the program for further use. A dose function that only prints cannot feed its answer into a second calculation. Exam rule of thumb: compute with **return**, display with **print** — in **main()**.

◆ Trap alert: Never use a mutable object as a default argument

def add_batch(batch, stock=[]) shares one list across all calls, so batches accumulate mysteriously between calls. Use **stock=None** and create the list inside. This is Python's most famous gotcha and a top-tier viva question.

Modular programs for pharmaceutical applications

"Modular" means splitting a program into small functions, each doing one job: one function calculates, another classifies, **main()** handles input/output. The payoff is testability — you can check **clarks_dose(500, 35)** gives **250.0** without typing any input. The syllabus names two programs; both are high-probability exam questions, so learn them as functions, not scripts.

Dosage calculation — Clark's rule

Clark's rule estimates a child's dose from the adult dose by weight: $\text{child dose} = \text{adult dose} \times (\text{weight in kg} \div 70)$, where 70 kg is the reference adult weight. (Related rules you may meet in posology: Young's rule uses age — $\text{dose} \times \text{age}/(\text{age}+12)$ — and Fried's rule is for infants under 2 years. Clark's is the weight-based one, which is why it fits this syllabus.)

```
def clarks_dose(adult_dose_mg, weight_kg):
    """Paediatric dose by Clark's rule: adult dose x (weight/70)."""
    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()
```

The `"""..."""` line is a docstring — documentation attached to the function, readable via **help(clarks_dose)**. One line of docstring is a professional habit examiners reward.

BMI calculation with classification

BMI = weight (kg) ÷ height (m)². The WHO cut-offs: below 18.5 underweight, 18.5–24.9 normal, 25–29.9 overweight, 30 and above obese. (Why height squared? Adolphe Quetelet found weight scales with the square of height across adults — the exponent that makes the index roughly independent of height.)

```
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"

def main():
    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)}")

main()
```

Note the design: **calculate_bmi** computes, **bmi_category** classifies, **main** handles I/O. Three jobs, three functions — that separation is what "modular" means, and it is exactly what the syllabus asks for. (In Unit IV you will apply this same **bmi_category** function to an entire DataFrame column at once.)

GPAT angle: Both programs are asked as "write a program" questions (5–10 marks). Marks split roughly: correct formula, function structure with **return**, input handling with casting, formatted output. A script with no functions loses the modularity marks.

Very short questions (2 marks)

- 1 Why is indentation syntactically significant in Python?
- 2 State the difference between **break** and **continue** in one line each.
- 3 What does **range(1, 6)** generate? Why does it stop at 5?
- 4 Distinguish between a parameter and an argument with an example.
- 5 State Clark's rule for paediatric dose calculation.

Short questions (5 marks)

- 1 Explain **if**, **if-else** and **if-elif-else** with a pharmacy example, and describe how nested conditions work.

- 2 Compare **for** and **while** loops: when is each used? Illustrate with examples and explain the infinite-loop hazard.
- 3 Explain the **return** statement. How does it differ from **print()**? Illustrate with a dose-calculation function.

Long questions (10 marks)

- 1 Write a modular Python program to calculate BMI and classify it into WHO categories. Explain the role of each function and the meaning of "modular programming".
- 2 Write a modular program to calculate a child's dose using Clark's rule. Explain function definition, argument passing, default arguments, and why returning values is preferred over printing them.

How to approach these answers

- Very-short Q1: indentation defines code blocks (no brackets); inconsistent indentation is a syntax error.
- Very-short Q2: **break** exits the loop entirely; **continue** skips to the next round — one line each.
- Very-short Q3: **1, 2, 3, 4, 5**; range stops one before the stop value (zero-based indexing reason).
- Very-short Q4: parameter = name in definition, argument = value at call; give the **calculate_dose** example.
- Very-short Q5: formula **adult dose × (weight/70)** + one line on what 70 represents.
- Short Q1: one example each for the three forms, then nested-if example (age × allergy); mention top-to-bottom testing of **elif**.
- Short Q2: **for** = known repetitions (**range**/list), **while** = condition-driven; infinite loop = condition never becomes False; give the input-validation example.
- Short Q3: **return** hands a value back and ends the function; **print** only displays; show why a printing function can't feed a second calculation.
- Long Q1: structure — formula + WHO cut-offs, three functions with roles, full code, closing line on modularity (separate compute/classify/I-O).
- Long Q2: structure — Clark's rule statement, **def** with parameters, call with arguments, default-argument example, **return** vs **print** justification, full program.

Unit III — Data Structures & File Handling

Lists, tuples and dictionaries

Single variables hold single values; real pharmacy data comes in collections — a list of drugs in stock, one patient's full record, a batch's fixed identity. Python's three collection types differ in exactly two properties: order and changeability (mutability).

List — ordered, mutable, square brackets. The default "sequence of things that may change":

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

Tuple — ordered, immutable, round brackets. Fixed data that must never change by accident:

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

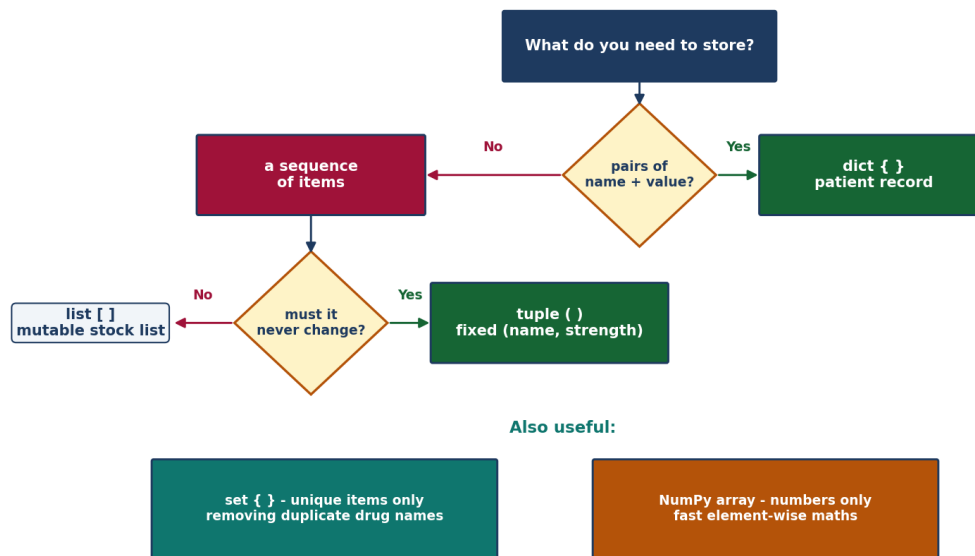
Dictionary — key-value pairs, curly braces. A labelled record where each value has a name:

```
patient = {"id": "P001", "age": 45, "drug": "Aspirin", "dose_mg": 75}
print(patient["drug"])      # Aspirin - access by key, not position
patient["dose_mg"] = 150    # values can be updated
```

Feature	List	Tuple	Dictionary
Brackets	[]	()	{ }
Ordered	Yes	Yes	Yes (insertion order)
Changeable	Yes (mutable)	No (immutable)	Values yes; keys no
Accessed by	Index number	Index number	Key name
Pharma example	Drugs in stock	Fixed (name, strength) pair	One patient's record

Why does immutability exist at all? Because immutable objects are safe: a tuple can be used as a dictionary key and shared freely, since nothing can alter it behind your back. Mutable objects are flexible but dangerous to share — which leads directly to the aliasing trap below.

Choosing the Right Python Data Structure



Decision flowchart for choosing between list, tuple, dict, set and NumPy array

Trap alert (aliasing): **stock2 = stock1** does not copy a list — both names point at the same list, so **stock2.append("X")** changes **stock1** too. This is the Unit 1 "variables as labels" mechanism biting in practice. Real copies: **stock2 = stock1.copy()** or **stock2 = stock1[:]**.

GPAT angle: "Differentiate between list, tuple and dictionary with examples" is a standard 5-marker. Reproduce the comparison table, and add one line on mutability for full marks.

Indexing and slicing

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

```

drugs = ["Aspirin", "Paracetamol", "Ibuprofen", "Amoxicillin"]
drugs[0]    # 'Aspirin'      (first)
drugs[-1]   # 'Amoxicillin' (last - no need to know the length)
  
```

Slicing picks a range with **sequence[start:stop:step]**:

```

drugs[0:2]   # ['Aspirin', 'Paracetamol'] - start included, stop excluded
drugs[1:]    # everything from index 1 onwards
drugs[:2]    # everything before index 2
drugs[::-1]  # the whole list reversed (step -1)
  
```

Why is the stop index excluded? Because then the length of the slice is simply **stop - start**: **drugs[1:4]** has exactly 3 elements. It also makes slices tile neatly — **drugs[:2] + drugs[2:]** reconstructs the whole list with no gaps or overlaps. Once you see this arithmetic, off-by-one slicing errors become easy to spot.

Operations on lists and dictionaries

```
# Lists
stock = [120, 200, 150]
len(stock)           # 3
stock.append(90)      # add one item at the end
stock.remove(200)     # remove by VALUE (first match)
del stock[0]          # remove by INDEX
stock.sort()          # sort in place
stock.count(150)      # how many times 150 appears
"Aspirin" in drugs    # True - membership test

# Dictionaries
patient.keys()        # all keys
patient.values()       # all values
patient.items()        # key-value pairs (for looping)
patient.get("allergy", "None") # safe lookup - returns "None" if key missing
del patient["age"]     # remove a key-value pair
```

Two dictionary mechanics worth understanding: lookup by key is effectively instant no matter how large the dictionary (it uses a hash table internally — that is also why keys must be immutable: lists can't be keys, tuples can). And since Python 3.7, dictionaries remember insertion order, so **list(patient.keys())** comes out in the order you added them.

Trap alert: **patient["allergy"]** on a missing key raises **KeyError** and crashes the program; **patient.get("allergy", "None")** returns the default instead. In dataset work where some records lack a field, **.get()** with a default is the safe habit. Also: **remove()** deletes by value while **del list[i]** deletes by index — mixing them up is a classic error.

String manipulation techniques (applied to data)

Unit I gave you the methods; here they earn their keep cleaning messy real-world text. Drug names in a dataset arrive as " **ASPIRIN 75mg** ", " **Aspirin** ", " **aspirin** " — three spellings of one drug that would group as three different drugs in Unit IV:

```
raw_names = [" ASPIRIN 75mg ", "Paracetamol", "aspirin 75MG"]
clean = [name.strip().lower() for name in raw_names]
# ['aspirin 75mg', 'paracetamol', 'aspirin 75mg']
```

That bracket pattern is a list comprehension — a compact loop that builds a list: **[expression for item in collection]**. It reads as "cleaned name for each name in raw names" and is the idiomatic way to transform a whole column of text.

Chaining is the other cleaning idiom: **`raw.strip().lower().replace("75mg", "150mg")`** — each method returns a new string that the next method works on, left to right. Typical exam framing: "Write a program to clean a list of drug names (remove extra spaces, convert to lowercase)."

Introduction to NumPy arrays

NumPy is the standard third-party library for fast numerical work. Its core object, the array, looks like a list but with two strict rules: every element has the same type, and maths applies to all elements at once — no loops:

```
import numpy as np

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

Why is this faster than a list loop? A NumPy array stores numbers in one contiguous block of memory (like C arrays), so the CPU streams through them without Python's per-element overhead. For a thousand concentration readings the difference is trivial; for a million-row dissolution dataset it is the difference between seconds and minutes. That contiguous, single-type storage is also why an array can't hold mixed **"Aspirin", 500** the way a list can — flexibility traded for speed.

Array creation — the four constructors to know:

```
np.zeros(5)           # [0. 0. 0. 0. 0.] - placeholder arrays
np.ones(4)            # [1. 1. 1. 1.]
np.arange(0, 10, 2)    # [0 2 4 6 8] - like range(), returns an array
np.linspace(0, 1, 5)   # [0. 0.25 0.5 0.75 1.] - N evenly spaced points
```

`linspace` deserves a special mention: it generates evenly spaced time points — exactly what you need for smooth concentration–time curves in Unit V. Useful attributes: **`arr.shape`** (dimensions, e.g. **(6,)**), **`arr.dtype`** (element type), **`arr.size`** (element count).

NumPy vs Python lists (exam favourite): arrays are faster for large numerical data, use less memory, and support element-wise maths directly; lists accept mixed types and are better for non-numerical collections like drug names.

GPAT angle: "Write NumPy code to create an array of concentrations and compute its mean and maximum" — three lines: **`np.array([...])`**, **`.mean()`**, **`.max()`**. Also be ready to state two differences between arrays and lists.

Reading and writing CSV files

A CSV (Comma-Separated Values) file is the simplest tabular format: each line is a row, commas separate columns. Excel opens CSVs, every instrument can export them, and Units III–V datasets all use this format — which is why the syllabus teaches the **csv** module (standard library, no pip needed) before Pandas:

```
import csv

# Writing
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 - each row becomes a dictionary keyed by column name
with open("patients.csv", "r") as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row["PatientID"], row["Dose_mg"])
```

Three mechanics to memorise: **with open(...)** closes the file automatically even if an error occurs (no forgotten **f.close()**); **newline=""** prevents phantom blank rows when writing on Windows; **DictReader** uses the header row as keys so you access **row["Dose_mg"]** instead of the fragile **row[2]**.

Trap alert: If a drug name itself contains a comma ("**Aspirin, enteric coated**"), a naive split on commas breaks the row. The **csv** module handles quoted fields correctly — hand-rolled **line.split(",")** does not. In the exam, always show the **csv** module, never manual splitting.

Understanding structured healthcare datasets

A structured dataset is data in rows and columns where:

- Each row = one observation (one patient, one tablet batch, one PK blood sample).
- Each column = one variable (age, dose, plasma concentration), with one consistent meaning and unit down the whole column.

Example — a tiny pharmacokinetic dataset:

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

Before any analysis, a disciplined analyst checks four things: what each column means, what the units are, whether values are plausible (negative age? concentration higher than the dose allows?), and whether anything is missing. That checklist is what "understanding structured healthcare datasets" means in the exam — and it is the human version of what **info()** and **describe()** automate in Unit IV.

Importing small pharmaceutical datasets: access and manipulation

Putting the unit together — load a small dataset with the **csv** module and do something useful with it:

```
import csv

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

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

Note the **float(...)** cast: CSV stores everything as text, so numeric columns must be converted before comparison — the Unit I casting lesson returning in dataset form. (In Unit IV, **read_csv()** does this type inference for you automatically.)

GPAT angle: "Write a program to read a CSV file of dissolution data and display batches with dissolution below 80%" — the pattern above is the complete answer: **DictReader**, loop, cast, condition, print.

Very short questions (2 marks)

- 1 State the difference between a list and a tuple with one example each.
- 2 What does **drugs[1:4]** return, and why is the stop index excluded?
- 3 Why can a tuple be a dictionary key but a list cannot?
- 4 Write NumPy code to create the array **[0, 2, 4, 6, 8]** using **arange**.
- 5 What is the purpose of **newline=""** when writing CSV files?

Short questions (5 marks)

- 1 Differentiate between list, tuple and dictionary with respect to brackets, ordering, mutability and access method. Give one pharmacy example of each.
- 2 Explain indexing and slicing in Python with examples. Include negative indexing and the **step** parameter.

- 3 Explain how to read and write CSV files using the **csv** module. Why is **DictReader** preferred over manual splitting?

Long questions (10 marks)

- 1 Describe NumPy arrays: creation methods (**array**, **zeros**, **ones**, **arange**, **linspace**), element-wise arithmetic operations, and useful attributes. Compare NumPy arrays with Python lists for pharmaceutical data analysis.
- 2 Explain the structure of a healthcare dataset (rows, columns, variables, units). Write a program to import a small pharmaceutical CSV dataset and perform access and manipulation tasks such as counting records and filtering rows on a condition.

How to approach these answers

- Very-short Q1: list = mutable [], tuple = immutable (); one example each.
- Very-short Q2: elements at indices 1, 2, 3; excluded so that slice length equals **stop – start**.
- Very-short Q3: dict keys must be immutable (hash table); tuples are immutable, lists are not.
- Very-short Q4: **np.arange(0, 10, 2)** — show the import line too.
- Very-short Q5: prevents blank rows between records when writing on Windows.
- Short Q1: reproduce the comparison table, then one pharmacy example per structure.
- Short Q2: start with 0-based indexing, negative indices, then slicing syntax with three examples including **[::-1]**.
- Short Q3: **with open, csv.writer/writerow, DictReader** loop; DictReader = column-name access + handles quoted commas.
- Long Q1: structure — what an array is and why fast (contiguous memory), four creation methods with outputs, element-wise ops, attributes, then the arrays-vs-lists comparison.
- Long Q2: structure — rows/columns/variables/units with the PK table, the four pre-analysis checks, then the full CSV program with the **float()** cast explained.

Unit IV — Data Handling with Pandas

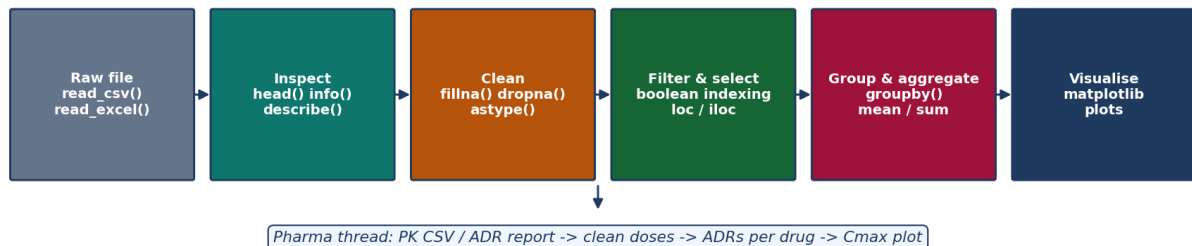
Introduction to the Pandas library

Pandas is the library that turns Python into a spreadsheet-with-a-brain: it handles labelled tables — data with column names, row labels, and graceful handling of missing values. NumPy gave you fast numbers; Pandas gives you fast numbers with names attached, plus one-line filtering, grouping and summarising. A PK study table or an ADR report lands in Pandas as a **DataFrame**, and nearly every analysis in pharma data work is some combination of: inspect → clean → filter → group → plot.

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

Install with **pip install pandas**; reading Excel files additionally needs **pip install openpyxl**. Under the hood, every Pandas column is a NumPy array with an index of labels bolted on — which is why Pandas is both fast and label-aware.

Pandas Data-Wrangling Pipeline: From Raw File to Insight



The Pandas data-wrangling pipeline from raw file to insight

Series and DataFrame structures

Pandas has two structures, and the relationship between them is the key insight:

- Series — a one-dimensional labelled array: one column of data with an index.

```
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:

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

A DataFrame is a collection of Series sharing one index — each column is a Series, and the row labels (the index) line up across columns. This "shared index" mechanism is what makes operations align correctly: add two Series and Pandas matches them by label, not by position. That label-alignment is also a trap source: combining Series with different indexes silently produces **NaN** where labels don't match, instead of raising an error.

Trap alert: `df["Dose_mg"]` returns a Series; `df[["Dose_mg"]]` (double brackets) returns a one-column DataFrame. They look almost identical when printed, but downstream code expecting a DataFrame will break on a Series. When an exam asks you to "select the Dose column as a DataFrame", the double brackets are the mark.

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 return a DataFrame, with column names taken from the header row and numeric types inferred automatically (the manual **float()** casting of Unit III, done for you). Practical points examiners probe:

- File paths: keep the dataset in the same folder as your notebook; otherwise the path must be exact. **FileNotFoundError** is almost always a wrong folder, not a wrong filename.
- Excel needs **openpyxl** installed, or **read_excel()** fails with an import error. Specify sheets with **sheet_name="ADR_2025"**.
- Real files have quirks: **na_values=["-", "NA", "missing"]** tells Pandas which strings count as missing, and **dtype={"PatientID": str}** stops IDs like **001** from becoming the number **1**.

Inspecting datasets: head(), tail(), info(), describe()

The non-negotiable first step with any new dataset is looking at it. Four functions form the standard inspection toolkit:

```
df.head()      # first 5 rows - what do the columns actually look like?
df.tail()      # last 5 rows - anything odd at the end?
df.info()      # column names, dtypes, and non-missing counts
df.describe()  # summary stats: count, mean, std, min, quartiles, max
```

How to read what they tell you:

- **head(10)** takes a number — show more rows when 5 isn't enough to judge.
- **info()** is your diagnostic scan: a dose column showing dtype **object** instead of **int64/float64** means text got mixed in (e.g. **"500mg"** instead of **500**) — dirty data, needs cleaning. The non-null counts reveal missing values per column at a glance.
- **describe()** summarises numeric columns only by default — that is normal, not an error. A **min** of **-5** in an Age column or a **max** dose of **50000** screams data-entry error. **include="all"** extends it to text columns.

GPAT angle: "Explain any four functions used to inspect a dataset in Pandas" is asked verbatim — **head()**, **tail()**, **info()**, **describe()**, one line on what each reveals.

Data cleaning techniques and handling missing values

Real datasets have gaps: a missed blood sample, an unrecorded age, a subject who dropped out. Pandas marks these as **NaN** (Not a Number), and **NaN** is contagious — any calculation touching it returns **NaN**, so means and plots silently go wrong unless you handle it.

```
df.isnull().sum()           # count missing values per column - always first
df.dropna()                 # remove rows containing any missing value
df.fillna(0)                # replace missing 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 dtype (after cleaning text)
```

When to use which — this judgement is what examiners reward:

- **dropna()** when missing rows are few and unimportant. But in a 20-patient PK study, dropping 3 rows loses 15% of your data — often unacceptable.
- **fillna()** when you can't afford to lose rows. Mean-fill is the textbook choice for numeric columns; median-fill resists outliers better (one mistyped age of 450 won't drag the median). For time-ordered PK data, forward-fill (**ffill**) carries the last observation forward — often more sensible than a mean.
- None of these mutate the original unless you assign the result back (**df = df.dropna()**) or pass **inplace=True**.

Other cleaning steps: **astype()** fixes wrong dtypes, **drop_duplicates()** removes repeated rows, and whole-column text standardising — **df["Drug"].str.lower().str.strip()** — applies Unit III string methods to every value at once.

Trap alert (chained indexing): **df[df["Age"] > 60]["Dose_mg"] = 500** may silently fail to update anything and raises **SettingWithCopyWarning**. The mechanism: the first [...] can return a temporary copy, so the assignment lands on the copy, not your DataFrame. Safe pattern: **df.loc[df["Age"] > 60, "Dose_mg"] = 500** — one **.loc** call, guaranteed to hit the original.

Filtering and selecting data based on conditions

Filtering picks rows meeting a condition, using a boolean mask — a Series of **True/False**, one per row, that Pandas uses as a stencil:

```
adults = df[df["Age"] >= 18]
high_dose = df[(df["Dose_mg"] > 500) & (df["Age"] < 60)] # & = AND
either = df[(df["Drug"] == "Aspirin") | (df["Age"] > 65)] # | = OR
not_aspirin = df[~(df["Drug"] == "Aspirin")]             # ~ = NOT
```

Three rules, each a classic trap:

- 1 Use **&** / **|** / **~**, not the words **and** / **or** / **not**. The words try to reduce the whole Series to a single bool and raise **ValueError: truth value of a Series is ambiguous** — because Pandas refuses to guess whether you meant "all true" or "any true".
- 2 Parenthesise every condition: **&** binds tighter than **>** in Python's precedence, so **df["Dose_mg"] > 500 & df["Age"] < 60** parses as nonsense. Brackets around each comparison, always.
- 3 String comparisons are case-sensitive: **"aspirin" != "Aspirin"** — clean the column first (previous section).

Selecting picks columns or cells:

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

loc uses labels, **iloc** uses integer positions — and here is the trap that catches everyone once:

Trap alert: **df.loc[0:5]** includes row label 5; **df.iloc[0:5]** excludes position 5. Label-based slicing is inclusive on both ends (because labels aren't guaranteed to be sequential numbers), position-based slicing follows Python's stop-excluded rule. Off-by-one errors live here.

Grouping data and performing aggregation functions

Grouping answers "per-category" questions — the single most common analysis pattern in pharma data: ADR count per drug, mean dissolution per formulation, max concentration per subject. The mechanism is split-apply-combine: Pandas splits rows into groups by the category, applies a summary function to each group, and combines the results:

```
df.groupby("Drug")["Dose_mg"].mean()      # mean dose for each drug
df.groupby("Drug")["ADR_Reported"].sum()  # total ADRs per drug
df.groupby("Drug").agg({"Dose_mg": "mean",
                        "Age": "max",
                        "PatientID": "count"})
```

Common aggregation functions: **mean()**, **sum()**, **count()**, **max()**, **min()**, **std()**. The **.agg()** form computes several summaries at once with a dictionary mapping column → function.

Why this matters: **groupby** replaces the Unit II/III pattern of looping over categories manually. "Total ADRs per drug" in plain Python needs a loop and a running-total dictionary; in Pandas it is one expressive line — and that one line is the canonical exam answer:

```
df.groupby("Drug")["ADR_Reported"].sum()
```

The result has one row per drug — which then feeds directly into a bar chart in Unit V (ADR reporting rates across drugs). That pipeline — read → clean → group → plot — is the whole subject in miniature.

GPAT angle: "A dataset has columns Drug and ADR_Reported. Write Pandas code to find the total ADRs reported for each drug." Answer:

df.groupby("Drug")["ADR_Reported"].sum(). Variations swap **sum()** for **mean()** or **count()** — read the question's verb.

Very short questions (2 marks)

- 1 State the difference between a Pandas Series and a DataFrame.
- 2 Which four functions inspect a dataset, and what does **info()** specifically reveal?
- 3 Why can't the words **and/or** be used for combining Pandas filter conditions?
- 4 State the difference between **loc** and **iloc** with one example each.
- 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 **head()**, **tail()**, **info()** and **describe()** as dataset-inspection tools. What signs of dirty data would you look for in their output?
- 2 Explain three strategies for handling missing values (**dropna**, **fillna** with mean/median, forward-fill) and when each is appropriate in a small clinical dataset.
- 3 Explain boolean filtering in Pandas with the three rules for combining conditions (**&/|**, parentheses, case sensitivity), and contrast **loc** vs **iloc** including the slicing-endpoint trap.

Long questions (10 marks)

- 1 A hospital ADR dataset has columns Drug, Dose_mg, Age and ADR_Reported with some missing values and inconsistent drug-name spellings. Describe the complete Pandas workflow: reading the file, inspection, cleaning (missing values, text standardisation, dtype fixing), filtering adults on high doses, and grouping to find total ADRs per drug. Give the key code for each step.
- 2 Explain the split-apply-combine mechanism of **groupby()** with aggregation functions (**mean**, **sum**, **count**, **max**, **min**). Illustrate with a PK/ADR example using **.agg()** for multiple summaries, and explain how the grouped result feeds into visualization.

How to approach these answers

- Very-short Q1: Series = 1-D labelled array; DataFrame = 2-D table; each DataFrame column is a Series.
- Very-short Q2: name all four; **info()** = column names, dtypes, non-null counts.

- Very-short Q3: they try to collapse the Series to one bool → "truth value is ambiguous" error; use `&|`.
- Very-short Q4: `loc` = labels (`df.loc[0, "Drug"]`), `iloc` = positions (`df.iloc[0, "Drug"]`).
- Very-short Q5: `df.groupby("Drug")["Dose_mg"].mean()` — exact line, no explanation needed.
- Short Q1: one line per function, then dirty-data signs: **object** dtype on numeric columns, null counts, impossible min/max.
- Short Q2: **dropna** loses rows (costly in small studies), mean/median fill preserves rows, **ffill** for time-ordered PK data; state the trade-off.
- Short Q3: boolean mask concept, then each rule with its error/trap; close with the **loc**-includes-endpoint vs **iloc**-excludes distinction.
- Long Q1: structure — read (**read_csv**, **openpyxl** note), inspect (four functions), clean (isnull→fillna/dropna, str.lower/strip, astype), filter (masked **&** example), group (**groupby.sum()**); key code per step, no full program needed.
- Long Q2: structure — split-apply-combine explained, one simple groupby, one **.agg()** multi-summary example, closing line on feeding results to bar charts (Unit V link).

Unit V — Data Visualization with Matplotlib

Introduction to Matplotlib

Matplotlib is Python's core plotting library — it converts analysed numbers into graphs. In pharmacy these graphs are not decoration: a concentration-time curve is the result of a PK study, a dissolution profile is the comparison of two formulations, and a histogram of tablet weights is a content-uniformity check. The skill being examined is two-fold: produce the plot, and read it scientifically.

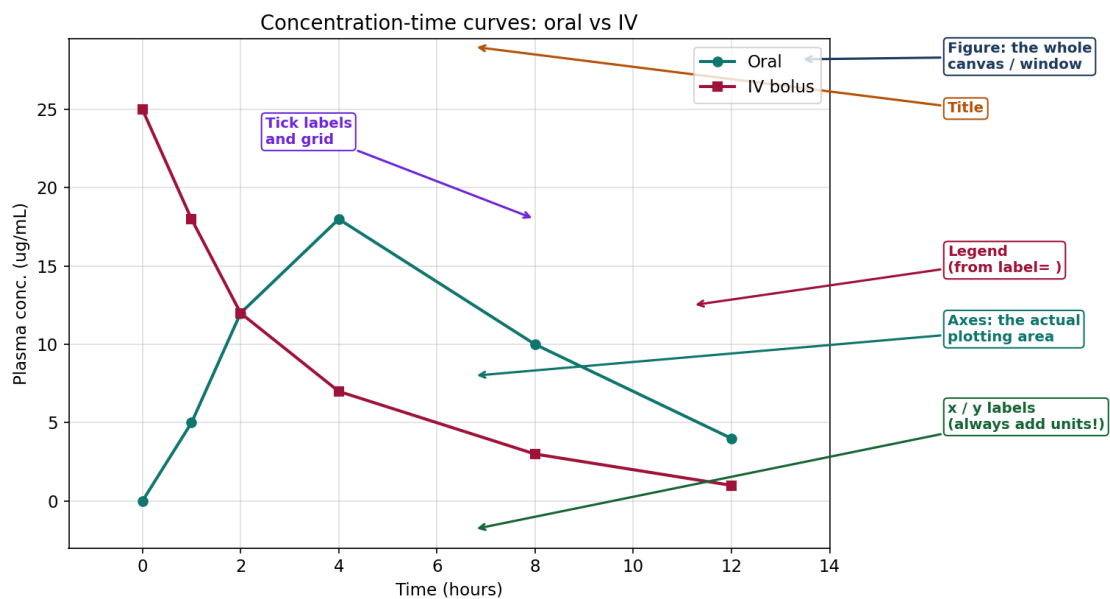
```
import matplotlib.pyplot as plt # the standard alias
```

The universal pattern: create the plot, label everything, then display:

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

In Jupyter, plots appear below the cell automatically, so **plt.show()** is optional but harmless. Before the plot types, learn the anatomy — because half of all Matplotlib confusion is vocabulary:

Anatomy of a Matplotlib Figure



Anatomy of a Matplotlib figure: Figure, Axes, title, labels, legend, ticks and grid

- Figure — the whole canvas/window; can hold multiple plots.
- Axes — the actual plotting area with your data (note the plural name for one plot — historic quirk).
- Axis labels — always with units; a concentration axis without $\mu\text{g/mL}$ is meaningless.
- Title, legend, ticks, grid — the supporting furniture. **label=** inside a plot command names the line; **plt.legend()** displays those names.

Trap alert (pyplot vs object-oriented): You will see two styles: **plt.plot(...)** (pyplot — commands act on the "current" plot, quick and fine for one graph) and **fig, ax = plt.subplots(); ax.plot(...)** (object-oriented — explicit, required for multi-panel figures). Mixing them — e.g. calling **plt.xlabel()** after creating **ax** plots — labels the wrong plot or nothing at all. In exams, the pyplot style is expected; just know the other exists and don't mix the two in one program.

Creating line plots, histograms, scatter plots and box plots

Each plot type answers a different question. Choosing correctly is half the marks.

Line plot — trends over a continuous variable

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

The line connects points in order, implying continuity between them — valid for time series (concentration-time, dissolution profiles) where intermediate values exist. **marker="o"** draws the actual measured points so readers see data vs interpolation. The workhorse of pharma graphs.

Histogram — distribution of one variable

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

A histogram chops one variable's range into bins and counts values per bin — revealing spread and shape: are tablet weights clustered tightly around 500 mg (good uniformity) or scattered (manufacturing problem)? The **bins** parameter is the judgement call: too few bins smear real structure away; too many bins turn random noise into spiky artefacts. With small samples, start with 5–10 and adjust.

Scatter plot — relationship between two variables

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

One dot per observation, no connecting lines (lines would falsely imply a measured path between doses). An upward-trending cloud supports a dose-response relationship; a formless cloud means no clear relationship. This is the visual foundation of pharmacology's most important curve.

Box plot — comparing distributions across groups

```
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()
```

Each box encodes five numbers: the line inside is the median, the box edges are the 25th and 75th percentiles (the middle 50% of values — the interquartile range, IQR), the whiskers extend to the furthest points within 1.5×IQR, and dots beyond are outliers flagging batches to investigate. Comparing box heights tells you which formulation is most consistent; comparing medians tells you which dissolves most.

Plot type	Answers the question	Pharma example
Line plot	How does Y change with X (continuous)?	Concentration-time, dissolution profile
Histogram	How are values distributed?	Tablet weight uniformity

Plot type	Answers the question	Pharma example
Scatter plot	Are two variables related?	Dose vs response
Box plot	How do groups compare?	Dissolution across formulations
Bar chart	How do counts compare across categories?	ADR reports per drug

GPAT angle: "Which plot shows the distribution of tablet weights?" — histogram. "Which compares dissolution across three formulations?" — box plot. Examiners test plot selection as much as plot code.

Labelling axes, titles and legends

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

```
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 - values become readable
plt.show()
```

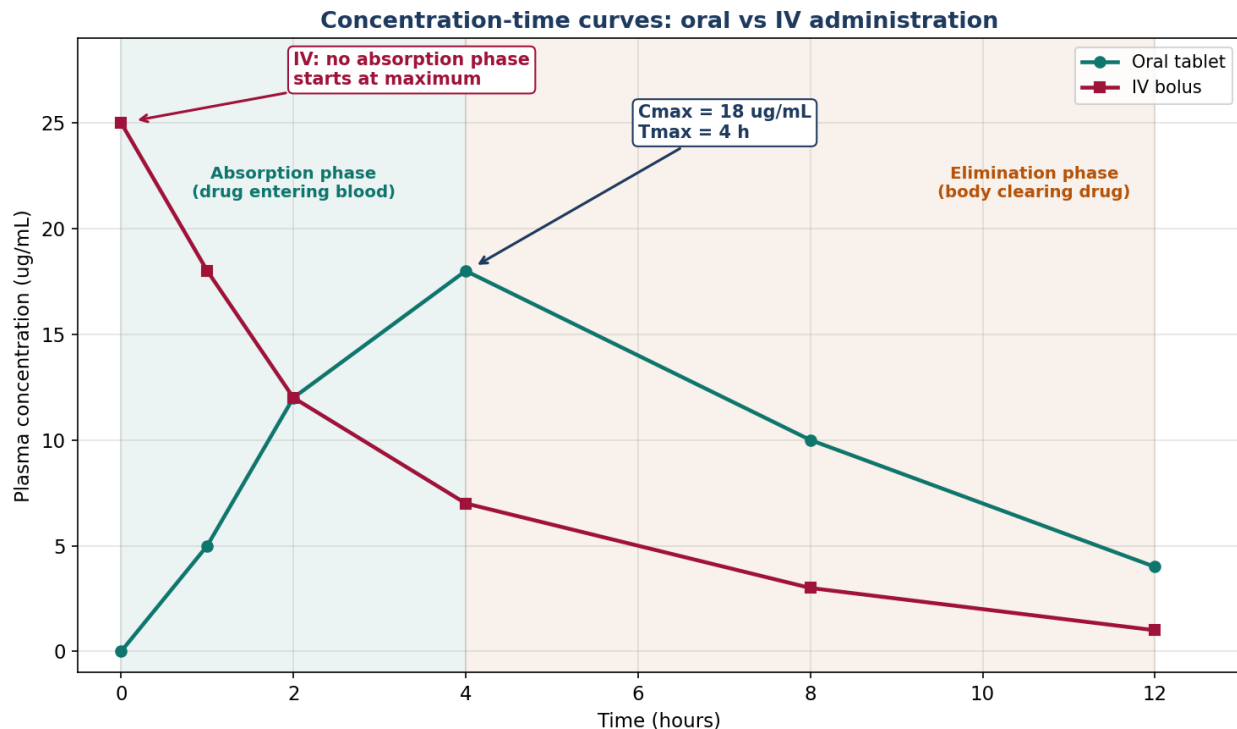
The mechanism of **legend()**: it does not invent names — it collects the **label=** strings from the plot commands. No **label=**, no legend entries (or worse, an empty box). Get into the habit: every plot command that draws data gets a **label**.

Visualizing pharmaceutical datasets

Concentration-time curves: oral vs IV (line plots)

```
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()
```



Concentration-time curves for oral vs IV administration with absorption and elimination phases, C_{max} and T_{max} marked

Reading the curves — this is where pharmacology meets the graph:

- The oral curve rises first (absorption phase — drug crossing from gut to blood), peaks, then declines (elimination phase). The peak height is C_{max} (maximum concentration: 18 $\mu\text{g/mL}$ here) and its time is T_{max} (4 h). C_{max}/T_{max} describe rate and extent of absorption.
- The IV bolus curve starts at its maximum at time zero and only declines — there is no absorption phase because the drug is placed directly into blood. 100% bioavailability by definition; the oral curve's lower exposure reflects incomplete absorption and first-pass metabolism.
- The area under the curve (AUC) represents total drug exposure — the basis for comparing bioavailability between formulations.

ADR reporting rates across drugs (bar chart)

```
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()
```

Note the plot choice: ADR counts are categories (drug names), not a continuous variable — so a bar chart (**plt.bar**), not a histogram, is correct. A histogram bins one continuous variable; a bar chart compares values across distinct categories. Confusing the two is a standard viva trap.

(This is also the natural visualisation of the Unit IV `groupby("Drug")["ADR_Reported"].sum()` result — the pipeline coming full circle.)

Dissolution profiles (line plots)

Percentage dissolved vs time, one labelled line per formulation — the same pattern as the concentration-time code with different labels and units (`plt.ylabel("Drug dissolved (%)")`). Visually you compare: which formulation releases faster? Do the curves stay close (similar profiles)? A step beyond the syllabus: regulators quantify "closeness" with the similarity factor f_2 — but the graph is always inspected first, because no number replaces seeing the curves.

Scientific interpretation of plots

Examiners award marks for reading graphs, not just drawing them. Practise these interpretations until they're reflexes:

- Concentration-time curve: locate C_{max} and T_{max} on the oral curve; explain the IV curve's missing absorption phase; state which route gives faster/higher exposure and why (no absorption barrier, no first-pass).
- Histogram (tablet weights): narrow and symmetric around target → good content uniformity; wide, skewed, or double-peaked → manufacturing problem (e.g. poor mixing, two populations).
- Scatter plot (dose vs response): clear upward trend → dose-response relationship supported; formless cloud → no evident relationship; a plateau at high doses → effect saturates (the E_{max} concept you'll meet in pharmacology).
- Box plot (formulations): highest median → dissolves most; shortest box → most consistent; outlier dots → batches to investigate, not to delete silently.

Trap alert: A plot never proves causation. A scatter trend between dose and relief supports a dose-response relationship; it doesn't prove the dose caused it (confounders exist). State relationships as "associated" or "consistent with", especially in ADR and observational data.

GPAT angle: "Plot the given concentration-time data for oral and IV administration and interpret the curves" (5 marks) splits into: correct code with labels and legend (3 marks) + two to three interpretation lines — C_{max}/T_{max} values, why IV lacks an absorption phase (2 marks). Always write the interpretation; code alone loses marks.

Very short questions (2 marks)

- 1 State the difference between a Figure and Axes in Matplotlib.
- 2 Which plot type shows the distribution of a single continuous variable? Give a pharmacy example.
- 3 What do the box, the inner line and the whiskers represent in a box plot?
- 4 Why is a bar chart (not a histogram) the correct choice for ADR counts per drug?

- 5 Define C_{max} and T_{max} with reference to an oral concentration–time curve.

Short questions (5 marks)

- 1 Explain the four plot types (line, histogram, scatter, box) with the question each answers and one pharmacy example each.
- 2 Describe the complete labelling checklist for a Matplotlib plot. How does **plt.legend()** get the names it displays?
- 3 Explain why the IV bolus concentration–time curve has no absorption phase while the oral curve does. What do C_{max}, T_{max} and AUC represent?

Long questions (10 marks)

- 1 Write Matplotlib code to plot concentration–time data for oral and IV administration on the same axes with full labelling, then interpret the curves scientifically (absorption vs elimination phases, C_{max}/T_{max}, exposure comparison).
- 2 Describe how you would visualize and interpret three pharmaceutical datasets: (a) tablet weight uniformity, (b) ADR reporting rates across drugs, (c) dissolution profiles of three formulations. Justify each plot choice and state what conclusions each plot supports.

How to approach these answers

- Very-short Q1: Figure = whole canvas; Axes = the plotting area holding the data.
- Very-short Q2: histogram; e.g. tablet weights around 500 mg.
- Very-short Q3: box = middle 50% (IQR), inner line = median, whiskers = spread to 1.5×IQR, dots = outliers.
- Very-short Q4: drugs are categories, not a continuous variable — bar chart compares across categories; histogram bins one continuous variable.
- Very-short Q5: C_{max} = peak concentration, T_{max} = time of the peak; point them out on the oral curve.
- Short Q1: one line per plot — question answered + pharmacy example; use the comparison table.
- Short Q2: xlabel/ylabel with units, title, label= + legend(), grid; legend collects label= strings.
- Short Q3: IV goes directly into blood (no absorption barrier, 100% bioavailability); oral must cross gut + first-pass; C_{max}/T_{max} = rate/extent of absorption, AUC = total exposure.
- Long Q1: structure — code block (two plot commands, labels, legend, grid), then interpretation under subheads: phases, C_{max}/T_{max} values, IV vs oral comparison; reference the curve diagram.
- Long Q2: structure — three subheads (a/b/c), each: plot choice + justification + what to read (spread/shape, category comparison, median/consistency/outliers); close with one line on interpretation over mere plotting.