BP101T COMPANION TEXT

Python for Pharmacists

Unit I · Introduction to Python Programming

BioDockify

Building the Future of AI-Driven Pharmaceutical Research

learn.biodockify.com

Course BP101T · B.Pharm Semester I

Chapters 1–5 · Six teaching hours

Theory-first · 58 code examples, every output machine-verified

Basics of Python Programming for Pharmaceutical Sciences

Contents

Chapter 1 — Python and Your Toolkit....1

1.1 What is a program?....1

1.2 How Python runs your code....2

1.3 Why should a pharmacy student learn Python?....2

1.4 Features of Python (syllabus point)....3

1.5 A short history of Python....4

1.6 Installing Python....5

1.7 Anaconda — the all-in-one option....5

1.8 IDEs — where you will type code....6

1.9 Why we never write Python in a plain text editor....7

1.10 Your first program....7

1.11 Troubleshooting your installation....8

Chapter 2 — Variables, Data Types and Type Casting....9

2.1 What is a variable?....9

2.2 Rules for naming variables....10

2.3 The four basic data types — theory....11

2.4 Checking a type: type()....14

2.5 The classic trap: 500 vs "500"....15

2.6 Type casting — converting on purpose....15

2.7 Comments — notes to the reader (and to yourself)....18

2.8 Worked example — paediatric syrup calculation....18

Chapter 3 — Operators, Input and Output....19

3.1 The seven arithmetic operators — theory....19

3.2 Operator precedence — who goes first?....20

3.3 Comparison operators — theory....21

3.4 Logical operators — and, or, not....22

3.5 Assignment shortcuts....22

3.6 input() — asking the user....23

3.7 print() — more than it looks....23
3.8 f-strings — printing like a human....24
3.9 Four complete pharmacy programs....24
Chapter 4 — String Operations and Manipulation....26
4.1 The bead-on-a-thread model — theory....26
4.2 Indexing — every bead has a number....27
4.3 Slicing — cutting out a piece....27
4.4 Immutability in action....28
4.5 Case and cleaning methods — upper, lower, title, strip....29
4.6 replace — swapping text....29
4.7 Searching — find, count, in....30
4.8 Taking apart and joining — split, join....30
4.9 Joining and repeating — +, *, f-strings....31
4.10 Worked programs....31
Chapter 5 — Libraries: Standard, Third-Party and pip....32
5.1 What is a library? — theory....33
5.2 Three ways to import....33
5.3 The math module....34
5.4 The datetime module — dates that calculate....35
5.5 Two more for the road....36
5.6 pip — install, list, uninstall (syllabus point)....36
5.7 Worked program — expiry report, complete....37
Unit I — Consolidation....37
Syllabus coverage map....37
Twenty-question exercise bank....38
Answers to selected exercises....41
Important points for examination....41

Right-click the table of contents and choose "Update Field" to refresh page numbers.

Unit hours: 6 · **Book chapters:** 1–5 · **Syllabus bullets covered:** installing Python and an IDE; variables and data types; type casting; operators; input and output; string operations; standard and third-party libraries.

Unit objectives. After studying this unit you should be able to: (1) install Python and a suitable IDE; (2) create and use variables of the four basic data types; (3) convert values between types; (4) apply arithmetic, comparison and logical operators; (5) read input from the user and produce formatted output; (6) manipulate strings; and (7) import, install and uninstall libraries.

Chapter 1 — Python and Your Toolkit

1.1 What is a program?

A **computer program** is a list of instructions that a computer follows, one after another. It is exactly like a recipe: ingredients go in (data), steps are followed in order (instructions), and a dish comes out (results). Everything a computer does — from printing your college hall ticket to analysing a clinical trial — is a program.

Python is the language we use to write those instructions. It is called a *high-level* language because it hides the electronics and lets you write steps that read almost like English: the same task usually takes fewer lines in Python than in older languages such as C++ or Java. Python is also *general-purpose* — the same language powers websites, hospital dashboards and drug-discovery research — so what you learn here travels anywhere.

1.2 How Python runs your code

Two facts explain almost everything you will see in this book.

1. **Python reads your program line by line, from top to bottom**, translating each line into action as it goes (this is what *interpreted* means). There is no separate "build" step: the moment you run the code, it runs. When an error exists on line 10, lines 1–9 have already executed — which is why partial output sometimes appears before an error message.
2. **Python can be used in two modes.** In *interactive mode* (the `>>>` prompt of a plain Python session, or a Jupyter cell) you type a line and see its result instantly — ideal for experimenting. In *script mode* you save a `.py` file and run the whole file at once — ideal for finished programs. Jupyter gives you both worlds: cells behave interactively, but the notebook is a saved document.

```
# script mode: nothing appears unless you print it
dose = 500
print(dose)
```

OUTPUT

```
500
```

1.3 Why should a pharmacy student learn Python?

Modern pharmacy runs on data. A single pharmacokinetic (PK) study produces many concentration readings per patient; pharmacovigilance centres receive thousands of adverse drug reaction (ADR) reports every month; every dissolution test adds batch after batch of time-point measurements. Spreadsheets struggle as this data grows. Python handles it comfortably — and never makes a copy-paste mistake.

Python is also:

- **Free and open** — no licence, no cost, runs on Windows, macOS and Linux.
- **The most popular programming language in the world** (first place on the TIOBE index for several years running), which means every question you will ever have already has an answer online.
- **The language of science** — the data-analysis tools used in pharmaceutical research (NumPy, Pandas, Matplotlib, RDKit) are all Python libraries, and Units III–V of this course are built on them.

1.4 Features of Python (syllabus point)

Seven properties explain why Python became the default language of science:

1. **Readable** — programs read close to everyday English, which shortens the learning curve for newcomers.
2. **Interpreted** — instructions take effect immediately, one line at a time, so mistakes surface early.
3. **Free and open** — anybody may use, share and study it; a large community maintains it.
4. **Portable** — one program behaves the same across Windows, macOS and Linux.
5. **Batteries included** — a rich standard library plus a vast ecosystem of community packages (NumPy, Pandas, Matplotlib among them).
6. **Dynamic types** — a variable's type follows from its value, with no declarations.
7. **Paradigm-flexible** — procedural, object-oriented and functional styles are all available.

Where Python already works in this field: analysing pharmacokinetic datasets (Cmax, Tmax, AUC), automating dose and dilution calculations, plotting dissolution profiles and concentration–time curves, managing large drug and ADR databases, machine learning for drug discovery and QSAR, and even simple hospital billing or stock software. Everything on this list is within reach of this course.

1.5 A short history of Python

Year	Event
Late 1980s	Guido van Rossum begins Python at CWI, Netherlands, as a successor to the ABC language
1991	Version 0.9.0 released — classes, functions, lists, dictionaries already present
1994	Version 1.0 adds functional tools (`lambda`, `map`, `filter`)
2000	Version 2.0 — served science for 20 years; retired in January 2020
2008	Version 3.0 — the modern, cleaned-up Python (this book uses Python 3)
Today	Python 3.13+ — still the world's most used language

Fun fact. Python is not named after the snake. Guido van Rossum was a fan of the British comedy group *Monty Python's Flying Circus*, and wanted a name that was short, unique and slightly mysterious.

1.6 Installing Python

1. Open **python.org/downloads**. The prominent yellow button names the current release for your OS (for example, *Download Python 3.14.8*).
2. Run the downloaded file. On the **first screen of the installer, tick the box "Add python.exe to PATH"** before clicking *Install Now*. This single checkbox lets Windows find Python from any terminal — forgetting it is the most common installation mistake.
3. Verify the installation. Press `Win + R`, type `cmd`, press Enter, and in the black window type:

```
python --version
```

```
OUTPUT
Python 3.14.8
```

Any `Python 3.x` answer means success. The companion command `pip --version` checks **pip**, Python's package installer, which arrives together with Python.

1.7 Anaconda — the all-in-one option

Anaconda (anaconda.com/download) installs Python *plus* about 250 data-science tools in one download — including Jupyter Notebook, Pandas, NumPy and Matplotlib, exactly the toolkit for

Units III–V. It is a large download (about 1 GB), but it saves many individual installations later. During setup choose *Install for: Just Me* and leave all other options at their defaults.

Two habits make Anaconda painless: open **Anaconda Prompt** (from the Start menu) instead of a plain terminal — inside it, `python`, `pip` and `jupyter` always work — and use **Anaconda Navigator** when you prefer clicking tiles to typing commands.

1.8 IDEs — where you will type code

An **IDE (Integrated Development Environment)** is a smart editor for code. The three we use in this course:

- **Jupyter Notebook** — runs in your web browser; code, notes, results and charts live together on one page. This is our classroom tool, and the natural home for data work.
- **VS Code** (code.visualstudio.com) — a light, free editor from Microsoft for every language; add the *Python* extension and it becomes a full Python environment.
- **PyCharm** (jetbrains.com/pycharm) — a professional Python IDE; the unified product's core is free forever, and the Pro edition is free for students with a college email.
- **IDLE** — Python's own built-in basic IDE; it arrives with every Python install and is fine for first experiments, though it lacks the comforts above.

IDE	Nature	Best used for	Cost
Jupyter Notebook	Browser-based, cell-by-cell	Data analysis, learning, step-by-step PK work	Free (part of Anaconda)
PyCharm	Full professional IDE	Large projects, application development	Core free; Pro paid (free for students)
VS Code	Lightweight, customisable editor + extensions	General coding and scripting	Free
IDLE	Minimal built-in IDE	Simple scripts, quick tests	Free (comes with Python)

To start Jupyter from Anaconda Prompt:

```
jupyter notebook
```

A browser tab opens at `localhost:8888` showing your folders (the *dashboard*). Choose **New → Python 3 (ipykernel)** and an empty notebook appears — its empty box is called a **cell**.

1.9 Why we never write Python in a plain text editor

You *can* write Python in Notepad, but nobody does. An IDE adds:

Plain text editor	IDE (Jupyter / VS Code / PyCharm)

IDE = smart editor + runner + error helper, in one place.

1.10 Your first program

In a fresh Jupyter cell, type:

```
print("Hello, Pharmacy!")
```

```
OUTPUT
Hello, Pharmacy!
```

Press **Shift + Enter**. That is the whole rhythm of Jupyter: *type → Shift+Enter → see the answer below the cell.* Note the quotation marks — they mark text (a **string**). Without them, Python would look for a variable called `Hello` and complain.

A second cell — a small taste of what this unit builds:

```
dose_mg = 500
weight_kg = 70
print(dose_mg / weight_kg, "mg per kg")
```

```
OUTPUT
7.142857142857143 mg per kg
```

1.11 Troubleshooting your installation

Problem	Cause and cure
`'python' is not recognized…`	The PATH box was not ticked. Re-run the installer and choose *Modify*, or reinstall with the box ticked.
Microsoft Store opens when you type `python`	Settings → Apps → Advanced app settings → App execution aliases → turn **off** both `python.exe` aliases.
`'jupyter' is not recognized`	Run it from Anaconda Prompt, or (plain-Python users) install it: `pip install notebook`.
An older Python answers	The Store copy is winning; apply the aliases fix above. Use `where python` to see all installed copies.

Chapter summary. A program is a list of instructions; Python executes them line by line, interactively or from a script. Python is free, dominant in science, and the natural tool for pharmaceutical data. Install it from python.org (tick *Add to PATH*) or take the all-in-one Anaconda

route. Work in Jupyter. The IDE colours your code, runs it with one keystroke and points at errors — that is why we never use plain text editors.

Quick checks 1. (a) Explain the difference between interactive and script mode. (b) What does the "Add python.exe to PATH" checkbox do? (c) Give three advantages of an IDE over Notepad. (d) What key combination runs a Jupyter cell?

Chapter 2 — Variables, Data Types and Type Casting

2.1 What is a variable?

A **variable** is a labelled box in the computer's memory. You store a value in it and use the label whenever you need the value. Formally, the assignment `name = value` does three things: Python creates the value (if needed), reserves the label `name`, and **binds** the label to the value. Nothing more — no type declaration, no memory paperwork. That is why creating a variable takes a single line:

```
drug = "Paracetamol"
dose_mg = 500
weight_kg = 70
print(drug, dose_mg, "mg")
```

OUTPUT

```
Paracetamol 500 mg
```

The = sign does **not** mean "equals". It means *"put the right side into the box on the left"*. Read `dose_mg = 500` aloud as "dose_mg **becomes** 500".

Several consequences follow from the binding idea:

Reassignment replaces. A variable holds one value at a time; assigning again simply re-binds the label, and the old value is gone.

```
tablets = 100
tablets = tablets - 10
print(tablets)
```

OUTPUT

```
90
```

Line 2 looks odd in arithmetic; in Python it means "compute the right side using the current value (100 − 10), then store the result back."

Dynamic typing. The *value* has a type; the *name* does not. The same label may hold a number today and text tomorrow (`dose = 500`, later `dose = "500 mg"`). Convenient — but it means *you* must keep track of what is in the box, which is what `type()` (§2.4) is for.

Multiple assignment. Python can fill several boxes in one line — including the famous swap:

```
a, b = 1, 2
a, b = b, a
print(a, b)
```

OUTPUT

```
2 1
```

Why variables matter. They let you write a value once and reuse it everywhere; update it in one place and every result follows; give calculations meaningful names (`dose_mg / weight_kg` reads like the formula itself); and store results for later use. This is exactly what a spreadsheet's cell references do — and programs go further, because a variable can also hold entire datasets (Unit III) or plots (Unit V).

2.2 Rules for naming variables

1. **Letters, digits and underscore only**: `dose_mg` ✓, `price2` ✓.
2. **Never start with a digit**: `dose2` ✓, `2dose` ✗.
3. **No spaces or hyphens**: `total dose` ✗ (splits in two), `total-dose` ✗ (reads as subtraction) — use `total_dose`.
4. **Case-sensitive**: `Dose`, `dose` and `DOSE` are three different variables. Using one you never created raises a `NameError`.
5. **Keywords are reserved**: `if`, `for`, `class`, `True`, … belong to Python's grammar and cannot be reused. (In a code editor they change colour — if a name you invented turns green or bold, it is taken.)

Name	Valid?	Reason
`dose_mg`	✓	letters + underscore
`_temp`	✓	may start with underscore
`batch_no_2026`	✓	digits allowed after the first letter
`2dose`	✗	starts with a digit
`total-dose`	✗	hyphen means minus
`for`	✗	keyword

Style. Legal is not enough — names should explain themselves. Python's convention is **snake_case**: lowercase words joined by underscores. Compare `r = d / x` with `mg_per_kg = dose_mg / weight_kg`; the second needs no comment. Future-you, reading notebooks at midnight before exams, is the audience.

2.3 The four basic data types — theory

Every value in Python has a **type**, and the type decides what the value can do: numbers take part in arithmetic, text can be sliced and searched, Booleans drive decisions. Python decides the type from *how you write the value* — quotes make text, a decimal point makes a float, bare digits make an integer, and `True`/`False` spelled exactly so make a Boolean.

(a) int — whole numbers. Integers are exact: there is no size limit in Python (a strength inherited from its design — `10 ** 100` computes without complaint), and integer arithmetic never loses anything. Rules of the road: no decimal point (`70` is an int, `70.0` is not), no thousand separators (`1,00,000` is invalid — write `100000`), negatives are fine (`-12`).

Five examples:

```
tablets = 500
print(tablets + 100)          # addition of ints stays int
age = 21
print(age * 2)
deficit = -12
print(abs(deficit))           # abs() drops the sign
big = 1000000                 # commas are not allowed
print(big * 1000)
print(17 // 5, 17 % 5)        # integer results from integer inputs
```

OUTPUT

```
600
42
12
1000000000
3 2
```

Pharmacy integers: tablet counts, patient ages, pack strengths (a 500 mg tablet is whole milligrams), stock units.

(b) float — decimal numbers. Floats represent measurements — quantities that come from the real world and are never exact. Internally they are stored in *binary* scientific notation (the IEEE 754 standard), which has a beautiful consequence you must meet early: fractions that are tidy in decimal (like 0.1) have no exact binary equivalent, so tiny rounding wigs appear:

```
print(0.1 + 0.2)
```

OUTPUT

```
0.30000000000000004
```

This is normal, not an error — every language with binary floats does it — and Unit III shows the practical cure (`round`). Other rules: the dot *makes* the float (`70` int, `70.0` float); the comma is never a decimal separator; very large/small values may use scientific notation (`1.5e3` is 1500.0).

Five examples:

```
temp = 37.5
print(temp + 0.7)
print(7 / 2)                  # / ALWAYS produces a float, even for ints
```

```
print(0.1 + 0.2)              # the binary-storage curiosity
print(1.5e3)                  # scientific notation
price = 32.50
print(price * 3)
```

OUTPUT

```
38.2
3.5
0.30000000000000004
1500.0
97.5
```

Pharmacy floats: weights, temperatures, volumes, concentrations, half-lives, prices.

(c) str — text (strings). A string is an *ordered sequence of characters* — beads on a thread — enclosed in single or double quotes (both are equal; pick one style). Inside the quotes anything goes: letters, digits, spaces, symbols — which is why `"500 mg"` and even `"123"` are text. Useful escape sequences let one character line stand for another:

Escape	Means
`\n`	new line
`\t`	tab
`\\`	a real backslash
`\'` / `\"`	a quote inside quotes

Long text can span lines with triple quotes. Strings are **immutable** — more in Chapter 4.

Five examples:

```
print("Dose: 5 mg")           # double quotes
print("It's a generic")       # apostrophe inside double quotes
print("Line 1\nLine 2")       # \n starts a new line
label = """Shake
well"""
print(label)
print(len("500 mg"))          # spaces count: 6 characters
```

OUTPUT

```
Dose: 5 mg
It's a generic
Line 1
Line 2
Shake
well
6
```

Pharmacy strings: drug names, batch IDs, blood groups, unit labels, barcodes.

(d) bool — truth values. A Boolean has exactly two values, `True` and `False` — capital first letter, no quotes (quoted `"True"` would be text!). Booleans are *produced* by comparisons and logical operators and *consumed* by decisions (`if`, Unit II). Two theoretical notes worth knowing: `bool` is actually a subtype of `int` (`True` behaves as 1, `False` as 0 — `True + True` is `2`); and every value in Python has a truthiness — zero numbers, empty text and empty containers count as `False`, everything else as `True`.

Five examples:

```
print(10 > 5)                 # comparisons PRODUCE bools
print(3 != 3)
expired = False
print(not expired)            # logic flips a bool
print("A" == "A")
print(bool(0), bool(7))       # truthiness of numbers
```

OUTPUT

```
True
False
True
True
False True
```

Pharmacy Booleans: `expired`, `in_stock`, `prescription_ok` — every checklist question is one.

(These four basic types are Unit I's business. Python also has a numeric type `complex` — numbers with a real and an imaginary part — and the collection types `list`, `tuple`, `dict` and `set`, which store many values inside one variable. They belong to Unit III and are only name-dropped here so that the family photograph is complete.)

2.4 Checking a type: type()

`type()` is the stethoscope of this course — whenever a result looks wrong, check the type first. It works on values and on variables alike, because variables simply hold values:

```
print(type(500))
print(type(37.5))
print(type("A+"))
print(type(True))
x = 98.6
print(type(x))
```

OUTPUT

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

```
<class 'float'>
<class 'str'>
<class 'bool'>
<class 'float'>
```

2.5 The classic trap: `500` vs `"500"`

```
print(500 / 70)
print("500" + 100)
```

OUTPUT

```
7.142857142857143
TypeError: can only concatenate str (not "int") to str
```

Same digits, different types. Quotes are not decoration — they change what a value **is**. A number stored as text cannot take part in arithmetic until it is converted, which is the business of the next section.

2.6 Type casting — converting on purpose

Type casting means converting a value from one type to another. Python distinguishes two directions of conversion:

- **Implicit conversion** — Python widens automatically when no information would be lost. Mixing an `int` with a `float` in arithmetic yields the wider type, silently:

```
result = 100 + 0.5      # int + float
print(result, type(result))
```

OUTPUT

```
100.5 <class 'float'>
```

- **Explicit conversion (casting)** — *you* order the change with the four converter functions. They are called *constructors*: `int("500")` reads the text, sees a whole number, and builds the matching integer. Because Python refuses most silent conversions (text never becomes a number by accident), explicit casting is a statement of intent — one of the language's safety features.

int() — to whole numbers (five examples):

```
print(int("500"))      # numeric text -> whole number
print(int(98.6))       # CUTS toward zero (never rounds)
print(int(-3.9))       # -3, not -4
print(int(True))       # True counts as 1
# int("4.5")           # ValueError - the dot makes it non-integer text
```

```
OUTPUT
500
98
-3
1
```

float() — to decimals (four examples):

```
print(float("70.5"))   # numeric text with a dot
print(float(70))       # int gains a dot
print(float("500"))    # whole-number text is fine too
print(float(True))
```

```
OUTPUT
70.5
70.0
500.0
1.0
```

str() — to text (four examples):

```
print(str(500))
print(str(37.5))
print(str(True))
print("Dose: " + str(500) + " mg")   # the pre-f-string way to build messages
```

```
OUTPUT
500
37.5
True
Dose: 500 mg
```

bool() — to truth values (two examples, one per outcome):

```
print(bool(0), bool(0.0), bool(""))    # falsy: zero and empty
print(bool(7), bool(0.5), bool("tablet"))   # everything else is True
```

```
OUTPUT
False False False
True True True
```

Why casting matters daily. Everything a user types arrives as text — even numbers:

```
weight = input("Weight in kg: ")   # user types 70
type(weight)
```

```
OUTPUT
<class 'str'>
```

So the everyday idiom casts at the door, in one line:

```
weight = float(input("Weight in kg: "))
```

Three warnings.

1. `int()` **cuts**, never rounds: `int(98.6)` → `98`. (When you truly want rounding, `round(98.6)` → `99`.)
2. The text must *look like* the target type: `int("500")` ✓, but `int("Paracetamol")` and `int("4.5")` both raise `ValueError` — the dot makes `"4.5"` non-integer text; convert via `float("4.5")` first.
3. `float()` always decorates: `float(70)` → `70.0`. Harmless — and it explains the mystery `.0` that appears in your answers.

2.7 Comments — notes to the reader (and to yourself)

A **comment** is a note inside the code that Python completely ignores at run time. Comments exist for humans: they explain *why* something is written, mark section headings, or temporarily switch a line off during testing.

Two forms:

```
# Single-line comment: everything after the hash is ignored
dose_mg = 500          # trailing comments explain one line

"""
Multi-line note (often used at the top of a program):
This program calculates a weight-based paediatric dose.
"""
print(dose_mg)
```

```
OUTPUT
500
```

Style rule for this book: comment the *unit* of every pharmacy number (`dose_mg = 500    # mg per tablet`) — future-you reads the notebook at revision time and needs no decoder ring.

2.8 Worked example — paediatric syrup calculation

```
# Paracetamol syrup for a child: 15 mg/kg, syrup has 24 mg per mL
weight_kg = 22
```

```
mg_per_kg = 15
mg_per_ml = 24

total_mg = weight_kg * mg_per_kg
ml_needed = total_mg / mg_per_ml
print("Give", ml_needed, "mL")
```

OUTPUT

```
Give 13.75 mL
```

Change `weight_kg` to 30 and re-run: `18.75 mL`. The formula lives in code; only the values change.

Chapter summary. Variables are labels bound to values by `name = value`; names follow five rules and snake_case style. Values carry one of four basic types — `int` (exact wholes), `float` (binary decimals with the 0.1 quirk), `str` (immutable character sequences), `bool` (truth values, secretly numbers) — inspectable with `type()`. Quotes and decimal points decide the type, and the constructors `int()/float()/str()/bool()` convert between them. Since `input()` always delivers text, casting is not optional — it is the door every number passes through.

Quick checks 2. (a) Why is `2dose` invalid? (b) What does *dynamic typing* mean? (c) `type("98.6")`? (d) `int(9.9)` vs `round(9.9)`? (e) Write one line that asks for a patient's age in years and stores it as an `int`.

Chapter 3 — Operators, Input and Output

3.1 The seven arithmetic operators — theory

An **operator** is a symbol that combines values (its *operands*) into a new value. Python's seven arithmetic operators look like mathematics, with two pharmacy-friendly additions. The essential theory:

- Operands steer the result: **integer op integer stays integer** — with the single exception of `/`, which *always* returns a float, because real division so often needs decimals.
- `//` (floor division) gives the whole-number part of a division; `%` (modulo) gives what is left over. Together they split any count into *wholes and remainders*.
- `**` is exponentiation and works with fractional powers (`0.5 ** 3`), which is exactly what pharmacokinetics needs.
- Operator precedence follows mathematics (`**` binds tighter than `*`, which binds tighter than `+`), and brackets override everything.

Operator	Meaning	Example	Result
`+`	add	`500 + 50`	`550`
`-`	subtract	`70 - 5`	`65`
`*`	multiply	`10 * 30`	`300`
`/`	divide (always float)	`7 / 2`	`3.5`
`//`	floor division (whole part)	`43 // 10`	`4`
`%`	remainder	`43 % 10`	`3`
`**`	power	`2 ** 3`	`8`

Five examples with pharmacy meaning:

```
print(500 + 50)        # total after adding a half-strip
print(140 - 55)        # the Cockcroft-Gault numerator (140 - age)
print(22 * 15)         # mg for a 22 kg child at 15 mg/kg
print(330 / 24)        # mL of a 24 mg/mL syrup
print(0.5 ** 3)        # three half-lives -> one-eighth of the dose remains
```

OUTPUT

```
550
85
330
13.75
0.125
```

The pharmacy pair `//` and `%` deserves its own worked example. A patient needs 43 tablets; strips hold 10:

```
total = 43
strips = total // 10      # 4 full strips
loose = total % 10        # 3 loose tablets
print(strips, "strips +", loose, "loose")
```

OUTPUT

```
4 strips + 3 loose
```

Check: 4 × 10 + 3 = 43. The same pair splits minutes into hours (`95 // 60` → 1, `95 % 60` → 35) and packs into cases (`100 // 24` → 4 full cases, `100 % 24` → 4 short).

3.2 Operator precedence — who goes first?

Python evaluates in the order **** → `* / // %` → `+ -`**, then comparisons, then `not`, `and`, `or`. Equal-priority operators evaluate left to right. When in doubt, use brackets — they never lie and they document intent:

```
print(2 + 3 * 4)        # 14 - multiplication first
print((2 + 3) * 4)      # 20 - brackets first
```

OUTPUT

```
14
20
```

The Mosteller body-surface-area formula in Chapter 5 is written `math.sqrt(height_cm * weight / 3600)` — brackets (and precedence) make the fraction explicit.

3.3 Comparison operators — theory

Six operators compare two values and answer a **Boolean** — they are questions, not arithmetic. Theory worth knowing: comparisons *chain* naturally in Python (`0 < stock < 10` means both must hold, as in mathematics); equality (`==`) compares *values*, not appearance (`3 == 3.0` is `True` — same value, different type); and text comparisons use dictionary order (`"A" < "B"` is `True`), capital letters sorting before lowercase.

Operator	Question	Example	Result
`==`	equal to?	`5 == 5`	`True`
`!=`	not equal to?	`5 != 3`	`True`
`>`	greater than?	`37.5 > 37`	`True`
`<`	less than?	`stock < 50`	depends

Operator	Question	Example	Result
`>=`	greater or equal?	`dose >= 500`	`True` for 500
`<=`	less or equal?	`age <= 12`	child dose

Five examples:

```
temp = 37.5
print(temp > 37.5)       # > EXCLUDES the boundary
print(temp >= 37.5)      # >= INCLUDES it
stock = 8
print(stock < 10)
print("A" != "B")
print(0 < stock < 10)    # chained: both must hold
```

OUTPUT

```
False
True
True
True
True
```

Two cautions. **`=` stores, `==` asks** — writing `=` inside a condition is a syntax error. And **boundaries are policy**: a pass mark of exactly 50 passes under `>=` and fails under `>`; a fever cut-off of 37.5 °C is "not yet fever" under `> 37.5`. Choose the operator deliberately.

3.4 Logical operators — and, or, not

Logical operators combine Booleans into one verdict. Theory: `and` needs **every** side True; `or` needs **at least one**; `not` flips. Python also *short-circuits*: in `A and B`, if `A` is already False, `B` is never even evaluated — useful when the second check would be unsafe (Unit II uses this).

Five examples — a dispensing checklist in pieces:

```
in_stock = True
rx_ok = True
expired = False
print(in_stock and rx_ok)          # both must hold
print(in_stock and not expired)    # condition + flip
print(expired or not rx_ok)        # either red flag
print(not expired)                 # plain flip
print(not in_stock or rx_ok)       # mixed: at least one side True
```

OUTPUT

```
True
True
False
```

```
True
True
```

Read the whole line `in_stock and rx_ok and not expired` aloud — it *is* the pharmacy's dispensing rule, written in code.

3.5 Assignment shortcuts

Augmented assignment folds “operate on yourself” into one operator — `x += 1` is exactly `x = x + 1`. Five examples:

```
tablets = 90
tablets += 10            # -> 100
stock = 20
stock -= 1               # -> 19
price = 32.50
price *= 1.05            # -> 34.125
dose = 500
dose /= 2                # -> 250.0  (note the float!)
shelf = 24
shelf //= 2              # -> 12
print(tablets, stock, price, dose, shelf)
```

OUTPUT

```
100 19 34.125 250.0 12
```

These forms appear constantly inside loops (Unit II), where a counter grows or stock shrinks once per pass.

3.6 input() — asking the user

`input()` pauses the program, shows a **prompt**, and returns whatever the user typed — **always as a string**, because keyboards produce text. The prompt is your question; the cast belongs in the same line:

```
name = input("Patient name: ")
weight = float(input("Weight in kg: "))
```

The program waits at each line until Enter is pressed. (Unit II turns this into whole question-and-answer programs; for now, one-line reads suffice.)

3.7 print() — more than it looks

`print()` converts each of its arguments to text, joins them, and adds a new line at the end. Two optional dials: `sep` (what goes *between* values — default one space) and `end` (what goes *after* — default a new line).

```
print("Dose", 500, "mg")            # several values, space-separated
print("A", "B", "C", sep="-")       # sep = what goes BETWEEN
print("next: ", end="")             # end="" -> no newline
print("room 3")                    # continues the same line
```

OUTPUT

```
Dose 500 mg
A-B-C
next: room 3
```

3.8 f-strings — printing like a human

Put an `f` before the quotes and the string becomes a *template*: inside curly braces `{ }`, Python evaluates an expression and inserts its result. Everything after a colon inside the braces is a **format specification** — a dressing instruction for display only:

Spec	Effect	Example	Shows
`:.1f`	one decimal	`f"{24.22:.1f}"`	`24.2`
`:.2f`	two decimals	`f"{7.142857:.2f}"`	`7.14`
`:6`	width 6, right-align	`f"\`	{500:6}\
`:<12`	width 12, left-align	`f"[{drug:<12}]"`	`[Paracetamol ]`

Five examples:

```
drug = "Paracetamol"
dose = 500
print(f"{drug} {dose} mg")
print(f"Double: {dose * 2}")           # expressions allowed
print(f"Per kg: {dose / 70:.2f} mg")
print(f"|{dose:6}|")                   # column alignment
print(f"[{drug:<12}]")                 # label alignment
```

OUTPUT

```
Paracetamol 500 mg
Double: 1000
Per kg: 7.14 mg
|   500|
[Paracetamol ]
```

The dressing changes only what is *printed* — the stored value is untouched (`round(x, 2)` is the separate function that rounds the value itself). Forgetting the `f` prints the braces literally: `print("{dose} mg")` → `{dose} mg`.

3.9 Four complete pharmacy programs

Program 1 — BMI calculator.

```
weight = float(input("Weight in kg: "))
height_m = float(input("Height in m: "))
bmi = weight / (height_m ** 2)
print(f"Your BMI is {bmi:.1f}")
```

OUTPUT

```
Weight in kg: 70
Height in m: 1.7
Your BMI is 24.2
```

Program 2 — dose per kilogram.

```
dose = float(input("Dose in mg: "))
weight = float(input("Weight in kg: "))
print(f"Dose per kg: {dose / weight:.2f} mg")
```

OUTPUT

```
Dose in mg: 500
Weight in kg: 70
Dose per kg: 7.14 mg
```

Program 3 — creatinine clearance (Cockcroft–Gault). CrCl (mL/min) = (140 − age) × weight / (72 × SCr); multiply by 0.85 for female patients.

```
age = int(input("Age in years: "))
weight = float(input("Weight in kg: "))
scr = float(input("Serum creatinine (mg/dL): "))
sex = input("Sex (M/F): ").strip().upper()

factor = 1.0
if sex == "F":
    factor = 0.85

crcl = (140 - age) * weight / (72 * scr) * factor
print(f"Creatinine clearance: {crcl:.1f} mL/min")
```

OUTPUT

```
Age in years: 55
Weight in kg: 70
```

```
Serum creatinine (mg/dL): 1.0
Sex (M/F): M
Creatinine clearance: 82.6 mL/min
```

(The `if` is a small preview of Unit II.)

Program 4 — child dose, two ways. Clark's rule uses weight; Young's rule uses age:

```
age = int(input("Child age in years: "))
weight = float(input("Child weight in kg: "))
adult_dose = float(input("Adult dose in mg: "))

clark = (weight / 70) * adult_dose
young = (age / (age + 12)) * adult_dose
print(f"Clark (weight): {clark:.1f} mg")
print(f"Young (age)  : {young:.1f} mg")
```

OUTPUT

```
Child age in years: 6
Child weight in kg: 20
Adult dose in mg: 500
Clark (weight): 142.9 mg
Young (age)  : 166.7 mg
```

Two formulas, one shape — *fraction × adult dose*. Notice the brackets in `age / (age + 12)`: without them the answer would be wrong. *In practice, mg/kg dosing is preferred; these rules give quick estimates.*

Chapter summary. Seven arithmetic operators (operands steer the result; `/` always floats; `//` and `%` split counts into wholes and remainders), precedence with brackets as the honest fix, six comparisons that manufacture Booleans (chaining allowed; boundaries are policy), and `and`/`or`/`not` with short-circuit evaluation. `input()` asks (returning text — cast immediately); `print()` reports with `sep`/`end` dials; f-strings evaluate expressions in braces and dress numbers with `:.1f`/`:.2f`/width specs. The four programs of §3.9 are worth retyping until they feel natural — they are the skeleton of every calculation you will ever automate.

Quick checks 3. (a) `43 // 10` and `43 % 10`? (b) Why does `7 / 2` give `3.5` but `7 // 2` gives `3`? (c) `50 >= 50` vs `50 > 50`? (d) What do `sep=` and `end=` control? (e) Produce `Per kg: 7.14` from `dose = 500` in one f-string.

Chapter 4 — String Operations and Manipulation

4.1 The bead-on-a-thread model — theory

A string is not one object; it is an **ordered sequence** of characters — beads on a thread. Each bead has a position number, positions start at **zero**, and every string-processing skill in this chapter (indexing, slicing, searching) is just different ways of addressing those beads. Two consequences: `len()` counts the beads (spaces included); and the sequence cannot be edited in place — strings are **immutable**, because Python shares text everywhere (labels, files, the internet) and protects it by never changing it after creation. Every "change" you will ever make is really the creation of a *new* string.

```
drug = "Paracetamol"
print(len(drug))
print(len("A+"))
print(len("500 mg"))     # the space counts
```

OUTPUT

```
11
2
6
```

4.2 Indexing — every bead has a number

Square brackets address a single character. Python counts **from zero**:

```
P   a   r   a   c   e   t   a   m   o   l
0   1   2   3   4   5   6   7   8   9   10
```

Negative indexes count from the end — a genuine convenience when you do not know the length. *Five examples:*

```
drug = "Paracetamol"
print(drug[0], drug[4])          # P c
print(drug[-1], drug[-2])        # l o
print(len(drug) - 1)             # the last valid index
print(drug[len(drug) - 1])       # the last character, computed
# drug[11]                        # IndexError - only 0..10 exist
```

OUTPUT

```
P c
l o
10
l
```

Reading past the last bead raises `IndexError` — the address exists, the bead does not.

4.3 Slicing — cutting out a piece

`[start : stop]` takes characters from `start` up to **but not including** `stop` — the same exclusion rule as `range()`. Omit `start` to begin at 0; omit `stop` to run to the end. Unlike indexing, an over-generous slice never crashes — it simply stops where the string stops.

Five examples — one batch ID, decoded:

```
batch = "MRP2026-114"
print(batch[0:3])       # 'MRP'   plant code
print(batch[3:7])       # '2026'  year (middle of the string!)
print(batch[8:])        # '114'   sequence
print(batch[:3])        # 'MRP'   missing start = from 0
print(batch[-4:])       # '-114'  from the 4th-last to the end
```

OUTPUT

```
MRP
2026
114
MRP
-114
```

Counting on fingers: `[3:7]` delivers positions 3, 4, 5, 6. One batch string, three facts — slicing *decodes* it, exactly the way barcodes and traceability codes work.

4.4 Immutability in action

```
drug = "Paracetamol"
drug[0] = "p"
```

OUTPUT

```
TypeError: 'str' object does not support item assignment
```

Strings are sealed. The workaround *is* the normal way — build a new string from pieces:

```
drug = "Paracetamol"
drug = "p" + drug[1:]
print(drug)
```

OUTPUT

```
paracetamol
```

This also explains the rule of the next section: **every string method returns a new string** and leaves the original untouched. Write `clean = raw.strip()`, not `raw.strip()` alone — the bare call computes and throws away.

4.5 Case and cleaning methods — upper, lower, title, strip

Theory: case methods map every letter through the Unicode case rules (digits and symbols pass through); `strip()` removes *whitespace* (spaces, tabs, newlines) from both ends — never from the middle; all of them return new strings.

Five examples:

```
name = "  pARACETamol  "
print(name.upper())              # shouty labels
print(name.lower())              # search keys
print(name.strip().title())      # clean AND case-fixed
print("vitamin b12".title())     # Vitamin B12
raw = "  Paracetamol  "
print(len(raw), len(raw.strip()), len(raw.lstrip()), len(raw.rstrip()))
```

```
OUTPUT
  PARACETAMOL
  paracetamol
Paracetamol
Vitamin B12
15 11 13 13
```

`strip()` pairs naturally with `input()`, which delivers stray spaces: `input("Drug: ").strip().title()` cleans a whole name in one line.

4.6 replace — swapping text

Theory: `replace(old, new)` substitutes **every** occurrence, left to right; an optional third argument limits the count; matching is case-sensitive and exact.

Four examples (the fourth is the safety lesson):

```
print("500 mg".replace("mg", "mcg"))       # a 1000-fold swap - handle with care
print("BC-2026-114".replace("-", ""))      # strip separators
print("aaa".replace("a", "b", 2))          # first 2 only
print("MG".replace("mg", "mcg"))           # case-sensitive: no match, no change
```

```
OUTPUT
500 mcg
BC2026114
bba
MG
```

A real mg→mcg swap is precisely how dispensing errors happen — the computer obeys exactly, so the pharmacist must choose the right operation.

4.7 Searching — find, count, in

Theory: `find()` returns the position of the **first** match (searching from the left) or **–1** when absent — never a crash; `count()` totals the matches; `in` is the plain-English membership test returning a Boolean, ready for Unit II's `if`.

Five examples:

```
drug = "Paracetamol"
print(drug.find("cetamol"))        # 4  - first match position
print(drug.find("xyz"))            # -1 - absent, no crash
print(drug.count("a"))             # 3  - positions 1, 3 and 7
print("cetamol" in drug)           # substring test
print("statin" in "Atorvastatin")  # drug-class suffix found
```

OUTPUT

```
4
-1
3
True
True
```

Drug-class suffixes (`-caine`, `-statin`, `-pril`) checked with `in` — cheminformatics in miniature.

4.8 Taking apart and joining — split, join

Theory: `split()` cuts a string into a **list** of pieces — at whitespace by default, at a separator you name otherwise; `join()` is its mirror, gluing a list of strings with one separator *between* them (the separator belongs to the string in front of `.join`). This pair is the mechanism behind CSV files (Unit III) and every "text → data" conversion.

Five examples:

```
print("500 mg".split())                   # at spaces
print("BC2026-114".split("-"))            # at a named separator
print("a,b,c".split(","))                 # CSV thinking
print(" + ".join(["Aspirin", "Paracetamol"]))   # glue with ' + '
print("\n".join(["Morning", "Night"]))     # glue with newlines
```

OUTPUT

```
['500', 'mg']
['BC2026', '114']
['a', 'b', 'c']
Aspirin + Paracetamol
Morning
```

```
Night
```

4.9 Joining and repeating — +, *, f-strings

The `+` operator glues strings (numbers must be dressed with `str()` first — or better, use an f-string); `*` repeats a string a whole number of times:

```
label = f"Dose: {500} mg"    # preferred - readable, computes inside { }
line = "-" * 30              # a 30-dash report divider
print(line)
```

OUTPUT

```
------------------------------
```

4.10 Worked programs

Program 1 — drug-name standardiser.

```
raw = input("Drug name: ")
clean = raw.strip().title()
print(f"Record : {clean}")
print(f"Label  : {clean.upper()}")
```

OUTPUT

```
Drug name:   aSPIRin
Record : Aspirin
Label  : ASPIRIN
```

Program 2 — dose-string parser.

```
dose_text = input("Dose (e.g. 500 mg): ")
parts = dose_text.split()
amount = float(parts[0])
unit = parts[1]
print(f"{amount / 70:.2f} {unit} per kg")
```

OUTPUT

```
Dose (e.g. 500 mg): 500 mg
7.14 mg per kg
```

Program 3 — batch-ID decoder.

```
batch = input("Batch ID: ")
print(f"Plant: {batch[:3]}")
print(f"Year : {batch[3:7]}")
```

```
print(f"Seq  : {batch[8:]}")
```

```
OUTPUT
Batch ID: MRP2026-114
Plant: MRP
Year : 2026
Seq  : 114
```

Chapter summary. Strings are immutable sequences: `len()` counts, `[i]` addresses (from zero; `[-1]` is last; `IndexError` past the end), `[a:b]` slices (stop excluded, over-runs forgiven) and decodes batch IDs. Methods return new strings: `upper/lower/title` re-case, `strip/lstrip/rstrip` trim edges, `replace` swaps (case-sensitive, all matches), `find/count/in` search without crashing, `split/join` take apart and reassemble — the bridge from text to data. `"500 mg"` becomes a number and a unit in two lines.

Quick checks 4. (a) `"Cetirizine"[:4]`? (b) `"Amoxicillin".count("i")`? (c) `"  Metformin ".strip()`? (d) Why does `drug[0] = "p"` fail, and what is the workaround? (e) `"BC2026-114".split("-")`?

Chapter 5 — Libraries: Standard, Third-Party and pip

5.1 What is a library? — theory

A **library** (often called a *module* when it is a single file) is a box of ready-made code that someone else wrote, tested and shared. `import` brings the box into your program and gives you access to everything inside through its name and a dot: `math.sqrt` means "the `sqrt` function that lives in the `math` module". This dot-addressing is called a *namespace* — it prevents name collisions, which is why your variable `pi` never disturbs `math.pi`.

Two kinds of library:

- **Standard library** — ships *with* Python; nothing to install, works offline. Over 200 modules: `math`, `datetime`, `random`, `statistics`, …
- **Third-party libraries** — built by the worldwide community and downloaded from **PyPI**, the Python Package Index hosting more than 400,000 packages. NumPy, Pandas, Matplotlib (Units III–V) and RDKit (drug discovery) all live here.

The dividing line matters for installation only — after import, both feel identical.

5.2 Three ways to import

```
import math                    # whole module: math.sqrt(16) -> 4.0
from math import sqrt          # just one tool: sqrt(16) -> 4.0
import numpy as np             # a nickname - the data-science convention
```

Style guidance: `import module` is clearest for beginners; `from module import name` saves typing for one or two heavily used functions; aliases (`as np`) shine when a module's name is used constantly — by Unit IV you will write `import pandas as pd` from muscle memory.

5.3 The math module

Theory: `math` holds the classical mathematical functions and constants, all operating on floats with full precision. Its two rounding cousins deserve special attention: `ceil` always rounds **up** (toward +∞) and `floor` always rounds **down** — neither *truncates toward zero* the way `int()` does on negatives.

Five examples:

```
import math
print(math.sqrt(16))            # 4.0 - always a float
print(math.sqrt(2))             # full precision
print(math.ceil(7.14), math.floor(7.8))
print(math.ceil(-7.14))         # -7, toward +infinity (int(-7.14) is -7 too
here; floor(-7.14) is -8)
```

```
print(round(math.pi, 4))
```

OUTPUT

```
4.0
1.4142135623730951
8 7
-7
3.1416
```

Pharmacy reading of `ceil`: when 7.14 packs of medicine are needed, you order **8** — never short the patient.

Worked program — BSA by the Mosteller formula (BSA in m² = √(height_cm × weight / 3600); BSA drives chemotherapy and paediatric dosing):

```
import math
height_cm = float(input("Height in cm: "))
weight = float(input("Weight in kg: "))
bsa = math.sqrt(height_cm * weight / 3600)
print(f"BSA = {bsa:.2f} m2")
```

OUTPUT

```
Height in cm: 170
Weight in kg: 70
BSA = 1.82 m2
```

(The formula needs height in **centimetres** — with metres the answer shrinks by a factor of ten and becomes clinically wrong. Units are policy.)

5.4 The datetime module — dates that calculate

Theory: a `date` object stores year, month and day as one value that knows the calendar. That is why dates *compare* with `>`, `<` like numbers and *subtract* into a duration whose `.days` you can read. `date.today()` is your program's connection to the actual current day.

Five examples:

```
from datetime import date
mfg = date(2026, 1, 15)
expiry = date(2027, 3, 31)
print(expiry.year, expiry.month, expiry.day)
print((expiry - mfg).days)          # shelf life in days
print(expiry > mfg)                 # dates compare like numbers
print(date.today())                 # the run date
print(mfg.weekday())                # 0 = Monday ... 3 = Thursday
```

OUTPUT

```
2027 3 31
440
True
2026-11-11
3
```

A near-expiry alert in two lines:

```
from datetime import date
expiry = date(2027, 3, 31)
today = date.today()

print(expiry > today)              # True - safe to dispense
print((expiry - today).days, "days left")
```

OUTPUT

```
True
140 days left
```

(The day count depends on the day you run it.) Real pharmacies run precisely this check on every batch.

5.5 Two more for the road

random — unpredictable numbers for sampling and simulation (it returns properly in Unit II and beyond):

```
import random
print(random.randint(1, 6))        # a die roll: one of 1..6
print(random.random())             # a float between 0 and 1
drugs = ["Aspirin", "Dolo", "Brufen"]
print(random.choice(drugs))        # one of the three, unpredictably
random.shuffle(drugs)              # mixes the list in place
print(drugs)
```

(Typical output: `3` · `0.721349...` · `Dolo` · `['Brufen', 'Aspirin', 'Dolo']` — different on every run.)

statistics — the descriptive trio, quietly previewing Unit IV where Pandas computes these for entire columns:

```
import statistics
print(statistics.mean([2, 4, 6]))
print(statistics.median([1, 3, 100]))          # 3 - the middle value, immune to
outliers
```

```
print(statistics.mode(["tab", "cap", "tab"]))  # the most frequent
print(statistics.stdev([2, 4, 6]))             # sample standard deviation
```

OUTPUT

```
4
3
tab
2.0
```

5.6 pip — install, list, uninstall (syllabus point)

`pip` arrived with Python (§1.6). From any terminal or Anaconda Prompt:

```
pip install pandas          # download + install from PyPI
pip install numpy pandas matplotlib   # several at once
pip list                    # every library you have, with versions
pip show pandas             # details of one package
pip install --upgrade pandas
pip install pandas==2.2.0   # pin an exact version
pip uninstall pandas        # asks y/n, then removes
```

Notes. Installing needs the internet and happens **once per computer**; after that `import` works forever, offline too. Never delete library files by hand — always `pip uninstall`. Anaconda users may also use `conda install` / `conda list` / `conda remove` (Anaconda's own store); in this course `pip` is the rule because it works in both setups.

5.7 Worked program — expiry report, complete

```
from datetime import date

drug = input("Drug: ").strip().title()
mfg = date(2026, 1, 15)
expiry = date(2027, 3, 31)
shelf_days = (expiry - mfg).days

print(f"{drug}: shelf life {shelf_days} days")
if expiry > date.today():
    print("Status: in date")
```

OUTPUT

```
Drug: paracetamol
Paracetamol: shelf life 440 days
Status: in date
```

Chapter summary. Libraries are borrowed expertise reached through namespaces: standard-library modules arrive with Python, third-party packages arrive by `pip` from PyPI. Three import styles serve three habits. `math` brings `sqrt/ceil/floor/pi` (Mosteller BSA in three lines; `ceil` is the pharmacy-grade rounding); `datetime` makes dates comparable and subtractable (expiry checking, shelf life, weekday); `random` and `statistics` wait in the wings. The pip lifecycle — install, list, show, upgrade, pin, uninstall — is the syllabus's final Unit I skill.

Quick checks 5. (a) `math.ceil(7.2)` and `math.floor(7.2)`? (b) Two differences between standard and third-party libraries. (c) The command to remove a library? (d) Write the import line that brings only `sqrt` into your program. (e) How would you print the number of days between two `date` objects?

Unit I — Consolidation

Syllabus coverage map

Syllabus bullet	Where
Installing Python and an IDE; IDE vs text editor	Ch 1 (§1.6–1.9)
Variables, data types, casting, operators, I/O	Ch 2–3
String operations and manipulation	Ch 4
Standard and third-party libraries; install/uninstall	Ch 5

Twenty-question exercise bank

Predict the output (1–8).

1. `print(43 // 10, 43 % 10)`
2. `s = "Paracetamol"; print(s[0], s[-1])`
3. `print(type("500"))`
4. `total = 5; total += 5; print(total)`
5. `print(f"{2 ** 3}")`
6. `print("A", "B", sep="+")`
7. `print(int(9.99))`
8. `print("Dolo".replace("Dolo", "Paracetamol"))`

Short answers (9–14).

9. State the five rules for naming variables, with one invalid example each.
10. Why does `input()` always require casting for arithmetic? Show the one-line idiom.
11. Explain `>=` vs `>` with a pass-mark example.
12. What is the difference between `int(9.9)`, `round(9.9)` and `math.ceil(9.9)`?
13. Why is `height_cm * weight / 3600` written with centimetres in the Mosteller program?
14. Give three advantages of an IDE over a text editor.

Write programs (15–20).

15. Ask two numbers; print sum, difference, product and quotient (2 decimals).
16. Ask a temperature in °C; print whether it is above 37.5 (preview of Unit II).
17. Batch decoder: read a batch ID like `HYD2027-042` and print plant, year, sequence.
18. Ask a drug name with messy spacing and case; print a clean title-case record and an UPPER-CASE label.

19. Days-supply: tablets in hand and per-day dose; print full days and leftovers (`//` and `%`).
20. Expiry checker: create `date(2028, 8, 31)` and print how many days remain from today, with an “in date / expired” line.

Answers to selected exercises

1. `4 3` — 4 strips, 3 loose. 2. `P l`. 3. `<class 'str'>`. 4. `10`. 5. `8`. 6. `A+B`. 7. `9` (cut, not rounded). 8. `Paracetamol`. 10. Because keyboard input arrives as text; `float(input("…"))`. 11. `>=` includes the boundary (50 passes); `>` excludes it. 12. `int` truncates to 9; `round` gives 10; `ceil` gives 10. 19. With 31 tablets and 2 per day: `31 // 2` → 15 days, `31 % 2` → 1 left.

Important points for examination

1. Guido van Rossum published the first public release in 1991. The language is *interpreted* (it runs line by line), *high-level* (it reads close to English) and *dynamically typed* (each value carries its own type). The name borrows from the Monty Python comedy troupe rather than the reptile.
2. While installing on Windows, tick **“Add python.exe to PATH”** on the first screen; afterwards `python --version` in a terminal confirms success.
3. An IDE bundles editor, debugger and one-click running — colour highlighting, name completion and instant error location that Notepad cannot offer. Jupyter suits stepwise data work; PyCharm suits big projects; VS Code is the light all-rounder; IDLE ships inside every Python install.
4. The Unit I quartet of types is int, float, str and bool — quotes make text, a decimal point makes a float. The containers (list, tuple, dict, set) wait until Unit III.
5. Name rules: letters, digits and underscore only; never digit-first; capitalisation matters; keywords are off-limits. snake_case is the house style.
6. Whatever the user types, `input()` hands back text — wrap it in `int()` or `float()` before any arithmetic. The converters are `int()`, `float()`, `str()` and `bool()`.
7. `int()` chops (98.6 → 98); `round()` rounds; `math.ceil()` rounds upward — when 7.14 packs are needed, order 8.
8. Operator families: arithmetic `+ - * / // % **` (true division always yields a float), comparisons `== != > < >= <=` (each yields a bool), logic `and or not`. Precedence runs `()` → `**` → `* / // %` → `+ -`, and brackets remain the final authority.
9. `>=` admits the boundary value; `>` rejects it — on pass marks and fever thresholds, that single character is a decision.
10. String positions begin at 0 (or at −1 counting from the rear); a slice `[start:end]` never includes `end`; and strings never change in place — methods hand back new strings.
11. The standard library needs only `import`; everything else arrives by `pip install` and departs by `pip uninstall`.

Sources for this unit: course decks Classes 1–5 and the author's classroom programs; topic coverage cross-checked against Noteskarts *Computer Applications in Pharmacy — Unit 1* notes, Yamuna Devi, Jeyabharathi & Vajpayee *Python Programming* (GCS) ch. 1–4, MCA R24 lecture notes Unit I, MRCET R17 notes Unit I, and the curated learn.biodockify.com catalog (trekhleb/learn-python, jackfrued/Python-100-Days, joaoventura/full-speed-python, gto76/python-cheatsheet, exercism/python). All prose and examples are original; code outputs verified by execution; originality audited against every consulted source (zero 7-word overlaps). Full credits: see the book's Sources and Acknowledgements page.