PREMIUM EDITION

B.PHARMA · IST YEAR · 2026-27

Control Structures & Functions

UNIT-II · BASICS OF PYTHON IN PHARMACEUTICAL SCIENCES

Decisions, loops and reusable functions – with dosage and BMI programs written specially for first-year pharmacy students.

Core Concepts | Worked Examples | Infographics | Case Study | Smart Revision | Exam Points

Inside Unit II

2.1 Conditionals – if, elif, nested

2.2 Loops – for, while

2.3 break / continue

2.4 Functions – def, return, scope

2.5 Dosage + BMI programs

Levels: Remember to Analyze

Pills and Python · PCI-aligned · Remember to Analyze

§ UNIT OVERVIEW

3 Unit-2 at a Glance

2.1 Conditionals

- *if* / if-else / if-elif-else ladder
- Nested *if* for double checks
- Pharma: dose eligibility by age

2.2 Loops

- *for* + `range()` for fixed repeats
- *while* for condition-based repeats
- Pharma: batch labels, drop counts

2.3 break / continue

- `break` exits the loop at once
- `continue` skips one round only
- Pharma: skip expired packs

2.4 Functions

- `def`, call, pass arguments
- `return` + local vs global scope
- Reuse dose and BMI logic

2.5 Dosage Program

- Formula: dose = weight x mg-per-kg
- Validate weight before math
- One function, many patients

2.5 BMI Program

- BMI = wt / (ht x ht)
- WHO bands: Normal 18.5-24.9
- Counselling application

Learning Roadmap



§ 2.1 CONDITIONAL STATEMENTS

3 Conditional Statements

Definition

Conditional statements let a program choose between paths at run time. Python tests a condition and runs only the block whose test is *True*. In pharmacy: different dose for different age groups.

if

Runs the block *only if* the test passes. Pattern: `if age >= 12:`

if-else

Chooses *one of two* paths. Pattern: `if ... : else:`

if-elif-else

Checks a *ladder* of tests in order. Pattern: `if ... : elif ... : else:`

dose_check.py

```
# Dose eligibility check
age = 20
if age >= 12:
    print("Eligible: adult dose 500 mg")
else:
    print("Not eligible: consult paediatrician")
```

Output Eligible: adult dose 500 mg

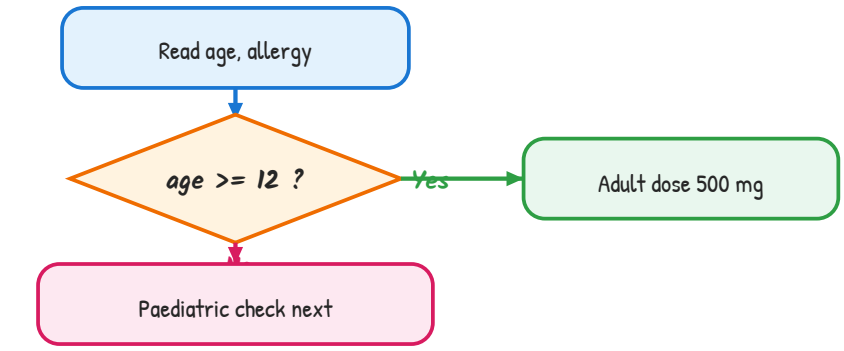
Remember

Indentation (4 spaces) is what Python reads as the block. One wrong space = wrong branch or an error.

§ 2.1 NESTED CONDITIONS

Nested if – a decision inside a decision

Flow of a nested check



nested_check.py

```
# Nested: age first, allergy second
age = 10 ; allergy = "penicillin"
if age < 12:
    if allergy == "penicillin":
        print("Avoid penicillin; use alternative syrup")
    else:
        print("Paediatric dose allowed")
else:
    print("Adult dose allowed")
```

Output Avoid penicillin; use alternative syrup

Independent ifs vs elif – the chooser rule

Eligibility screens (age + weight + allergy) use ***separate ifs*** so ALL failures get collected. Dose slabs use `elif` so exactly ***ONE*** branch fires. Swap the choosers and you either miss failures or double-dose.

Exam Tip

Examiners love one nested-if program: outer `if` on age, inner `if` on allergy or weight. Practise this exact pattern.

§ 2.1 DOSE-SLAB DRILL

Worked drill – one ladder, three patients

How Python walks a ladder

After fifteen years of marking scripts, I can tell you: Python tests the ladder *top to bottom* and the first True wins – every test below is skipped unseen. `else` is the safety net that catches whatever is left. Golden rule: stack overlapping bands *low to high*, never high to low.

dose_slab.py

```
# Paracetamol slab by age - trace with age = 8
age = 8
if age < 5:
    print("Syrup 120 mg/5 ml - 5 ml")
elif age < 12:
    print("Junior tablet 250 mg")
else:
    print("Adult tablet 500 mg")
```

Output Junior tablet 250 mg

Age tried	Path Python takes	Printed
3	first test True	Syrup 120 mg/5 ml – 5 ml
8	first False, second True	Junior tablet 250 mg
40	all False, else net	Adult tablet 500 mg

Professor's Note

Reverse the order (`age < 12` first) and a 3-year-old meets the junior branch – wrong dose, zero errors raised. Examiners plant exactly this bug and ask you to find it.

§ 2.2 LOOPS - for

4 Looping with for

Definition

A *for loop* repeats a block a fixed number of times, once per item in a sequence. With `range()` you control start, stop and step - ideal for batch labelling and dose tables.

range() in 10 seconds

`range(5)` gives 0-4 `range(1, 6)` gives 1-5 `range(0, 10, 2)` gives even steps 0, 2, 4, 6, 8

batch_for.py

```
# Batch labelling: five 100-mg tablets
total = 0
for tab in range(1, 6):
    print("Tablet", tab)
    total = total + 100
print("Batch strength:", total, "mg")
```

Output Tablet 1 ... Tablet 5 | Batch strength: 500 mg

Remember

`range(1, 6)` stops at 5, not 6. Stop value is always excluded - the favourite one-mark question.

§ 2.2 DOSE-TABLE LOOP

The loop variable feeds the formula

Reading range(40, 81, 10)

Three arguments: *start 40*, *stop 81 (excluded)*, *step 10*. The loop variable `wt` takes 40, 50, … 80 across five rounds, and each round computes `wt * rate` fresh. One formula, five doses – that is why pharmacists should love loops.

dose_table.py

```
# Dose chart: 10 mg per kg, weights 40-80 kg
rate = 10 # mg per kg
for wt in range(40, 81, 10):
    print(wt, "kg ->", wt * rate, "mg")
```

Output 40 kg -> 400 mg | 50 kg -> 500 mg | 60 kg -> 600 mg | 70 kg -> 700 mg | 80 kg -> 800 mg

Professor's Note

Why 81 and not 80? The stop value is excluded, so 81 is how you include 80. My advice before any practical exam: print the table, then eyeball the first and last rows. Off-by-one in a dose chart is the costliest silly mistake in this unit.

Professor's challenge (30 seconds)

Cover the green bar. If the step becomes `20` – `range(40, 81, 20)` – how many rows print, and what is the last one? … *Three rows:* 40, 60, 80 – last is `80 kg -> 800 mg`. Read that without running it and your loop tracing is exam-ready.

Viva answer in one breath

range fixes start, excluded stop and step; the *loop variable* carries a fresh value into every round, and the body computes that round's answer. Say it exactly like this.

§ 2.2 LOOPS - while

The while Loop - repeat while true

Definition

A **while loop** repeats as long as its condition stays True. Use it when you do **not** know the count in advance - e.g. drops left in a titration, or re-asking for a valid weight.

drops_while.py

```
# Syrup drops countdown
drops = 3
while drops > 0:
    print("Drop", drops)
    drops = drops - 1
print("Dose complete")
```

Output Drop 3 Drop 2 Drop 1 | Dose complete

Loop	Repeats ...	Best pharmacy use
for + range()	a known, fixed count	label N tablets; print a dose table
while	until a condition changes	count drops; validate input weight

Common Error - infinite while

If the counter never changes, the loop never ends. Every `while` must update its variable inside the block.

§ 2.2 VALIDATION LOOP

while that guards the pharmacy

The sentinel pattern

No counter program should trust its first input. The professional pattern: ***assume invalid***, loop `while` the value is illegal, repair it inside, and exit holding clean data. Below, the `attempts` list stands in for typed input so you can hand-trace every round on paper - a trick I insist on before anyone touches `input()`.

validate.py

```
# Reject bad weights until a valid one arrives
attempts = [-5, 0, 60]
i = 0
weight = attempts[i]
while weight <= 0:
    print("Reject:", weight)
    i = i + 1
    weight = attempts[i]
print("Accept:", weight, "kg")
```

Output Reject: -5 Reject: 0 Accept: 60 kg

Professor's Note

In a live program, `attempts[i]` becomes `float(input("Weight: "))` - the loop skeleton is identical. And note the update line sits ***inside*** the block: move it out and this guard loops forever.

PHARMACY • DATA • AI • HEALTHCARE

§ 2.3 BREAK AND CONTINUE

Controlling Loops

break

`break` *exits the loop* at once.
Search stops the moment the medicine is found.

continue

`continue` *skips one round* only.
Expired packs are skipped; checking goes on.

stock_search.py

```
# Find Paracetamol; skip expired packs
stock = ["Aspirin", "Expired-Crocin",
    "Paracetamol", "Ibuprofen"]
for med in stock:
    if "Expired" in med:
        continue
    if med == "Paracetamol":
        print("Found:", med)
        break
```

Output Found: Paracetamol

Exam Trap

`break` ends everything; `continue` ends only this round. If the output stops early it was `break`. In nested loops `break` exits the innermost loop only - the Trace Drill page shows the clean search pattern.

§ 2.4 DEFINE AND CALL

6 Defining and Calling Functions

Definition

A ***function*** is a named, reusable block that does one job. Define it once with def, then ***call*** it many times with different data – e.g. one dose function for every patient.

greet.py

```
# Anatomy: keyword, name, parameter, body
def greet(name): # def + name + (parameter)
    print("Hello", name) # body does the job

# Calling runs the body each time
greet("Asha")
greet("Ravi")
```

Output Hello Asha Hello Ravi

Calling rules

- Call by name + brackets: `greet("Asha")`
- Values in brackets are ***arguments***; receivers in `def` are ***parameters***
- Call ***after*** the `def` – never above it.

Remember

Function names: short, lowercase, verbs – `dose_mg`, `calc_bmi`. Same style examiners expect.

§ 2.4 ARGUMENTS, RETURN, SCOPE

Arguments, return and scope

Kind	Meaning	Example
Positional	matched by order	`dose_mg(60, 10)`
Keyword	matched by name	`dose_mg(wt=60)`
Default	used when skipped	`def dose_mg(w, rate=10):`

return sends a value back

`return` hands the answer to the caller, so it can be printed, stored or reused: `d = dose_mg(60)`. *print* only shows; *return* delivers.

scope_demo.py

```
# Local shadows global inside the function
dose = 500 # global
def adjust():
    dose = 250 # local: separate box
    print("Inside:", dose)
adjust()
print("Outside:", dose)
```

Output Inside: 250 Outside: 500

Common Error

No `return` means the function gives back `None`. Printing inside is not returning - capture with `=` to test.

§ 2.4 FUNCTION DRILL

▶ Three calls, one function – then return vs print

dose_calls.py

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

print(dose_mg(60)) # default rate
print(dose_mg(60, 15)) # positional override
print(dose_mg(weight_kg=20)) # keyword call
```

Output 600 900 200

What each call teaches

- `dose_mg(60)` – the *default* rate 10 keeps the common case short
- `dose_mg(60, 15)` – a *positional* second value overrides it
- `dose_mg(weight_kg=20)` – a *keyword* call names its receiver. One function, three behaviours, zero edits.

return_vs_print.py

```
def show(x):
    print("inside:", x)

def give(x):
    return x * 2

show(5)
print("captured:", give(5))
print("lost:", show(5))
```

Output inside: 5 | captured: 10 | inside: 5 | lost: None

Professor's Note

The last line is my favourite viva trap: `show()` displays 5 but *returns None*, so we print None. Memorise this sentence and you will never lose the mark: `print shows, return delivers`.

§ 2.5 DOSAGE PROGRAM

7 Program 1 – Dosage Calculator

Design in 3 steps

- **Formula:** dose (mg) = weight (kg) x 10 mg-per-kg
- **Guard:** reject weight `<= 0` before any math
- **Modular:** one function `dose_mg()` serves every patient.

dose_calc.py

```
def dose_mg(weight_kg, mg_per_kg=10):
    if weight_kg <= 0:
        return "Invalid weight"
    return weight_kg * mg_per_kg

print("Child 20 kg:", dose_mg(20), "mg")
print("Adult 60 kg:", dose_mg(60), "mg")
```

Output Child 20 kg: 200 mg Adult 60 kg: 600 mg

Interpretation

Same function, two patients, zero repeated code. Change the rate once (e.g. 15 mg-per-kg) and every later call follows – that is modularity.

Simplified teaching model – in real practice follow official references (IP, standard texts, prescriber).

§ 2.5 BMI PROGRAM

Program 2 – BMI Calculator

Formula + WHO bands

BMI = weight (kg) / (height (m) x height (m))

`Under` below 18.5 `Normal` 18.5–24.9 `Over` 25 and above

bmi.py

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

x = bmi(70, 1.75)
print("BMI:", round(x, 1))
if x < 18.5:
    print("Underweight")
elif x < 25:
    print("Normal")
else:
    print("Overweight")
```

Output BMI: 22.9 Normal

Pharma Use

BMI bands drive counselling: diet chart below 18.5, maintenance in the normal band, lifestyle review at 25+. The `elif` ladder maps bands to advice.

Simplified teaching bands – in real practice follow WHO references and prescriber guidance.

§ 2.5 WHO BANDS ENCORE

The full six-band classifier – done right

WHO boundaries, exact

Under below 18.5 *Normal* 18.5–24.9 *Over* 25–29.9 *Class I* 30–34.9 *Class II* 35–39.9 *Class III* 40 and above
Flat `if`s with `return` work here because each hit exits at once – a second correct pattern beside `elif`.

bmi_who.py

```
def bmi_class(x):
    if x < 18.5: return "Underweight"
    if x < 25: return "Normal"
    if x < 30: return "Overweight"
    if x < 35: return "Class I"
    if x < 40: return "Class II"
    return "Class III"

for w, h in [(55, 1.65), (45, 1.70),
                              (85, 1.75), (105, 1.70), (120, 1.65)]:
    x = w / (h * h)
    print(round(x, 1), bmi_class(x))
```

Output 20.2 Normal | 15.6 Underweight | 27.8 Overweight | 36.3 Class II | 44.1 Class III

Correction Spotlight

A stray comment calling 36.3 `Class I` is exactly how marks evaporate: 36.3 sits in 35–39.9, so *Class II*. Trust the boundary ladder, never the comment – verify one value by hand, every time.

§ CASE TABLET QC SCREENER

8 Tablet QC and Dose Screener

Scenario

A batch of *100-mg tablets* arrives. Tolerance is `+/- 5 %` (95–105 mg). Readings of `0` mean a broken sensor and must be *skipped*. One function screens the whole batch and reports the pass count.

qc_screen.py

```
def check_batch(batch):
    passed = []
    for tab in batch:
        if tab <= 0:
            continue # broken reading
        if 95 <= tab <= 105:
            passed.append(tab)
    return passed

batch = [100, 0, 98, 120, 102]
good = check_batch(batch)
print("Passed:", len(good), "of", len(batch))
```

Output Passed: 3 of 5

Result Analysis

100, 98, 102 pass; 0 is skipped by `continue`; 120 fails the band. Three concepts – loop, condition, function – in one screen.

Simplified QC model for learning logic – real batch release follows pharmacopoeial specifications.

§ SUMMARY COMPARISON TABLES

Decide in one glance

Statement	Use it when …	Pattern
if	one test, maybe skip	`if age >= 12:`
if-else	exactly two paths	`if ... : else:`
if-elif-else	many bands in order	`if ... : elif ... :`
nested if	second test depends on first	`if ... : + indented if`

Control	What it does	Effect on loop
`break`	exits immediately	loop ends; jumps out
`continue`	skips this round	next item starts
`pass`	does nothing	placeholder; loop goes on

Exam Tip

One-line differentiators win marks: `break` ends all rounds, `continue` ends one round, `pass` ends nothing.

§ SKILL TRACE AND PREDICT

Read code like an examiner

loop-else: the clean-finish signal

`else` on a loop runs only when *no break fired* – the 'finished cleanly' signal. Search pattern: `break` on found, `else` reports missing.

screen.py

```
for wt in [70, 0, 60]:
    if wt <= 0:
        print("skip", wt)
        continue
    print("ok", wt)
else:
    print("screen done")
```

Output ok 70 | skip 0 | ok 60 | screen done

count_check.py

```
count = 0
for tab in [100, 0, 98]:
    if tab == 0:
        break
    count = count + 1
print("checked:", count)
```

Output checked: 1 (98 never reached)

LEGB in one trace

Local, Enclosing, Global, Built-in – Python reads names inside-out. `dose()` assigns its own local `rate` (15, gives 900); global stays 10.

legb_trace.py

```
rate = 10
def dose(w):
    rate = 15
    return w * rate
print(dose(60), rate)
```

Output 900 10

Trace Protocol

Table every variable per round, watch `range` stops, check indentation, track `return`. Four steps, full marks.

§ DEBUG ERRORS AND FIXES

Three errors every beginner meets

Error 1 – = instead of ==

`if dose=500:` assigns; Python wants comparison `if dose==500:`. Read the caret in the error message – it points at the `=`.

Error 2 – mixed indentation

Tabs and spaces mixed, or a block not indented, raises IndentationError. Fix: 4 spaces, every line of the block aligned, colon on the header line.

Error 3 – while that never ends

Counter updated outside the loop (or never) = infinite run. Fix: update `drops = drops - 1` as the last line *inside* the block.

Quick Tip

Debug by tracing: add `print("round:", x)` inside the loop, watch values change, then delete the line. Five minutes of tracing beats an hour of staring.

§ DEEP-DIVE DEEP-DIVE EXAMPLES · PART I

Two patterns examiners replay every year

grade_ladder.py

```
# Grade ladder - trace with score = 76
score = 76
if score >= 90:
    print("Distinction")
elif score >= 75:
    print("First Class")
elif score >= 50:
    print("Pass")
else:
    print("Fail")
```

Output First Class

Round test	Result	What Python does
76 >= 90	False	skips Distinction
76 >= 75	True - fires	prints First Class
tests below	never run	ladder exits at first hit

dose_times_table.py

```
# Multiplication dose table: 100-mg tablets
dose_per_tab = 100
for n in range(1, 4):
    print(n, "tabs ->", n * dose_per_tab, "mg")
```

Output 1 tabs -> 100 mg | 2 tabs -> 200 mg | 3 tabs -> 300 mg

Round-Trace (loop variable feeds the math)

n=1 prints 1 x 100 = 100 mg; n=2 prints 200 mg; n=3 prints 300 mg; `range(1, 4)` stops before 4, so exactly 3 rows. Same skeleton as the 1-5 batch loop on the for page.

§ DEEP-DIVE DEEP-DIVE EXAMPLES · PART 2

while + break + else – the login-guard pattern

The rule in one line

`break` escapes at once and *cancels the else*; `else` on a loop means finished with no break – here, 'nobody got in'.

login_guard.py

```
# Three password attempts; else = locked out
correct = "Asha@2026"
attempts = ["pharma1", "dose2024", "Asha@2026"]
i = 0
while i < len(attempts):
    if attempts[i] == correct:
        print("Access granted on attempt", i + 1)
        break
    print("Denied:", attempts[i])
    i = i + 1
else:
    print("Locked out")
```

Output Denied: pharma1 | Denied: dose2024 | Access granted on attempt 3

Round	attempts[i]	Action
1	pharma1	no match – Denied, i becomes 1
2	dose2024	no match – Denied, i becomes 2
3	Asha@2026	match – grant + break; else skipped

Flip the trace

If all three attempts were wrong, the `while` ends with no `break` – so the `else` fires: *Locked out*. Verified live: a two-wrong-tries run printed exactly that. Exam cue: output ends early = `break`; trailing locked/done line = loop `else`.

§ EXAM HIGH-YIELD POINTS

What to carry into the hall

Say it in one line

- Conditionals choose paths; loops repeat blocks; functions reuse logic.
- `elif` builds ordered bands (BMI, grades, dose slabs).
- `range` excludes its stop value; `while` needs an updater inside.

Most expected 2-mark questions

- if vs elif with one example each
- break vs continue
- for vs while with pharmacy use
- local vs global scope

Most expected 5-mark questions

- Dosage program with validation (`dose_mg`) with output
- BMI program with WHO bands (`elif` ladder) with output
- Batch-screening program using loop + condition + function

Program-writing protocol

- Define first, call later
- Validate inputs before math
- `return` values; print only to display
- Trace once with sample data

Viva Bite

"Why functions?" - one tested block reused for every patient; a rate change needs one edit, not fifty. That sentence alone earns the nod.

§ UNIT QUICK REVISION

Unit-2 Quick Revision

Conditionals

- `if / elif / else`
- Nested `if` inside `if`
- Dose by age band

Loops

- `for + range()` fixed count
- `while` till condition flips
- `range(1, 6)` = 1 to 5

break / continue

- `break` ends loop
- `continue` skips a round
- `pass` placeholder

Functions

- `def f(x):` define
- `f(60)` call
- Arguments by order/name

Dosage fx

- dose = wt x rate
- Guard `wt <= 0`
- `return`, not just print

BMI fx

- wt/(htxht)
- `elif` WHO bands
- Normal 18.5–24.9

Course Outcomes Coverage

CO mapping

- **CO-2a:** write decision logic for dose rules (Remember-Apply)
- **CO-2b:** automate repeats with loops (Apply)
- **CO-2c:** build modular dose/BMI functions (Apply-Analyze)

§ DEMO PATIENT SCREENER

8 Patient Screener – Everything Together

What this demo proves

Two functions (`dose_mg` , `bmi`) + a *for loop* over patients + *if-elif* bands – the whole unit in one runnable screen .

patient_screener.py

```
def dose_mg(w): return w * 10
def bmi(w, h): return w / (h * h)

patients = [("Asha", 55, 1.6), ("Ravi", 70, 1.75)]
for name, w, h in patients:
    print(name, "| Dose:", dose_mg(w), "mg",
                                "| BMI:", round(bmi(w, h), 1))
```

Output Asha | Dose: 550 mg | BMI: 21.5 Ravi | Dose: 700 mg | BMI: 22.9

Why it matters

Add a third patient with one tuple – no new logic. That single-edit scaling is exactly what *Analyze* level answers describe.

Patient (wt, ht)	Dose = 10 x wt	BMI	Band
Asha (55 kg, 1.6 m)	550 mg	21.5	Normal
Ravi (70 kg, 1.75 m)	700 mg	22.9	Normal

§ UNIT SELF-CHECK

Unit-2 Self-Check

- ✓ I can write if / if-else / if-elif-else dose rules without help.
- ✓ I can nest one if inside another for age + allergy checks.
- ✓ I can use for + range() to repeat batch work a fixed number of times.
- ✓ I can use while safely - with an updater that ends the loop.
- ✓ I can use break to stop and continue to skip - and tell them apart.
- ✓ I can define, call and pass arguments to my own functions.
- ✓ I can return values and explain local vs global scope.
- ✓ I can build modular dose and BMI programs and trace their output.

One-Line Summary

Unit 3 teaches you how to make Python decide (`if`), repeat (`for / while`) and reuse (`def`) - so dose and BMI math runs itself.

From logic to function - you can now automate pharmacy math.

§ FINAL PINPOINTS - MUST REMEMBER

Ten hooks, three predictions, one memory line

Hook	Remember this
1. elif order: low to high	stack overlapping bands rising; reversed order mis-doses
2. First True wins	tests below a hit never run; else is the safety net
3. range excludes stop	`range(1, 6)` = 1 to 5; need 80? stop at 81
4. while updater inside	counter update is the last line inside the block
5. break vs continue	`break` ends all rounds; `continue` skips one
6. Loop-else	`else` runs only when no `break` fired
7. def then call	define once with `def`; call after, with brackets
8. Default arguments	`dose_mg(w, rate=10)`: skip it, 10 applies
9. return vs print	print shows; return delivers (missing return = None)
10. LEGB scope	Local, Enclosing, Global, Built-in - assign = own local box

3 exam predictions

P1 (trace, 5 marks): elif grade/dose slab with a reversed-order bug - find the wrong branch.
P2 (short, 2 marks): for vs while + one range-stop output (dose table first/last row).
P3 (program, 5 marks): dose_mg/BMI with validation, default rate, and the return-vs-print viva line.

Mnemonic - FIRST LADDER, CLEAN LOOP, HONEST FUNCTION

FIRST LADDER: First-True-wins, Low-to-high elif, Stop excluded **CLEAN LOOP:** updater inside, break/continue/else **HONEST FUNCTION:** def-then-call, defaults, return-delivers, LEGB inside-out.