Lab 04 — Making Decisions

Conditions: `if`, `elif`, `else`

DURATION ~3 hours COVERS Lectures 5 & 6 TASKS 11 files CHECKER `check.py`

How this lab works

- Work top to bottom — each section builds on the one before it.
- **Warm-ups** (W1, W2, ...) are quick drills done in the Thonny **Shell** or on this sheet. Nobody collects them; they load the ideas you need for the task that follows.
- **Tasks** (TASK 1–11) are the deliverable. Each one is its own file — `task01.py` to `task11.py` — inside your `lab04` folder.
- **The badges are a ladder.** EASY tasks drill one idea exactly as the lecture showed it. MEDIUM tasks reuse a familiar idea in a new setting. HARD tasks combine several ideas at once. Exam questions are built up the same ladder — and the bottom rungs of the exam are precisely what this lab rehearses, so climb all of them.
- **Trace before you run.** Walk each snippet by hand with the given values and write the output down first. A program with branches has more than one story; tracing is how you learn to read all of them.
- **Type the code yourself.** No copy-paste. Your fingers are learning a language — including the four spaces.
- Run `check.py` whenever you like. Your goal before leaving: **all 11 tasks PASS**. Show the final report to your instructor.
- This lab assumes Lab 03: `input()` returns a string, casting with `int()` and `float()`, f-strings, and the six comparison operators.

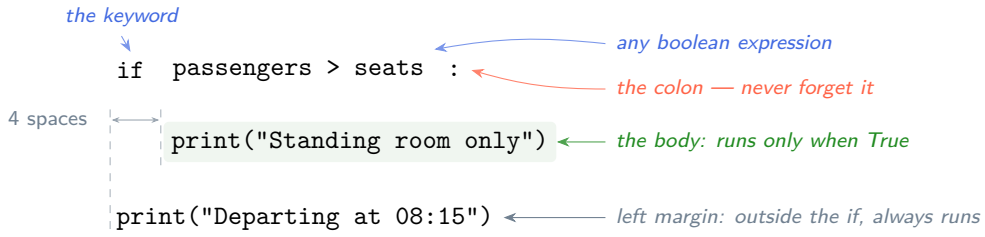
Before you start: get the files

Download the `lab04` folder (from the course LMS page) onto your machine. It contains `task01.py` ... `task11.py` and `check.py`. Keep everything in that **one folder**, and do not rename any file — the checker finds your work by its file name.

1 One-Way Decisions: the `if` Statement (~25 min)

Ref: Lecture 5 — Conditions

Until today every line of your programs ran, every time, in order. An `if` statement breaks that promise — on purpose. It guards a block of code with a boolean question, and the block runs *only* when the answer is `True`.



W1. EASY Set `marks = 60` in the Shell. For each condition below, decide *in writing* whether the body of `if <condition>:` would run — then type the condition alone into the Shell (`True` means it runs).

Condition	Runs or skipped?	Shell says
`marks > 60`	.....	.....
`marks >= 60`	.....	.....
`marks != 60`	.....	.....
`marks == 60`	.....	.....
`not marks < 60`	.....	.....

Two of these five behave identically on `marks = 60`. Which two — and would they still agree if `marks` were 75?

W2. EASY These two scripts differ by **four spaces**. Type both, run both:

```
stock = 0
if stock == 0:
    print("Out of stock")
    print("Order more")
```

```
stock = 0
if stock == 0:
    print("Out of stock")
print("Order more")
```

Same output — so is the difference imaginary? Now change `stock` to `5` in both and run again: left: ..... right: .....
One test case agreed; the other exposed the bug. Remember that when you test today's tasks.

W3. MEDIUM Write the condition, not the answer. Fill each blank so the `if` line says exactly what the English says:

In English	The condition
the tank is empty (variable `litres`)	`if` ..... `:`
the ticket costs over Rs 500 (`price`)	`if` ..... `:`
the seat number is even (`seat`)	`if` ..... `:`
the user left the name blank (`name`)	`if` ..... `:`

The last one needs no comparison operator at all — an empty string is already *falsy*. `if not name:` reads like English.

TASK 1 Standing Room Only

EASY `task01.py` ~8 min

The campus shuttle prints a notice before it leaves. `task01.py` binds `passengers` and `seats`. Print the warning `Standing room only` **exactly when** there are more passengers than seats, and always print `Departing at 08:15`:

```
Standing room only
```

```
Departing at 08:15
```

One `if`, one indented line, one line at the left margin — the diagram above, typed out. Before you run it, change `passengers` to `40` and predict what disappears. Change it back before you save.

TASK 2 **Two Broken Lines** EASY `task02.py` ~8 min

`task02.py` crashes before printing anything. It should print:

```
Part-time student
Registration complete
```

There are exactly **two** bugs, and both are on Lecture 5's pitfall list. Fix them one at a time: run the file, read only the *last* line of the error, fix that, run again. `Registration complete` prints for every student, so it stays at the left margin.

The two ways Python refuses an `if`

No colon — `if units < 4` ⇒ `SyntaxError: expected ':'`. Python read the whole line and never found where the condition ended.

No indent — a body flush against the left margin ⇒ `IndentationError: expected an indented block`. The colon promised a block and nothing was offered. Use **four spaces**, and never mix tabs with spaces — they look identical and Python treats them as different.

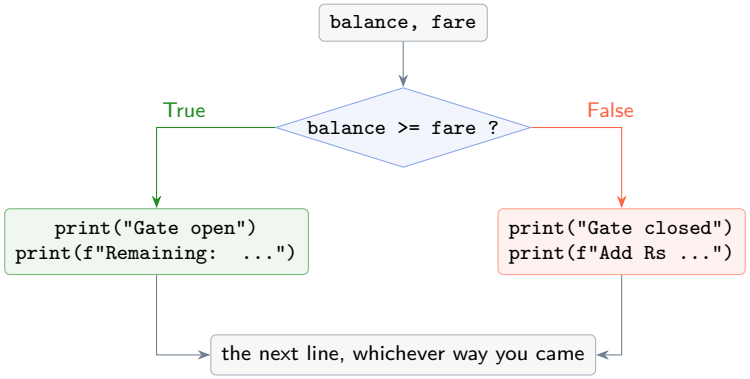
Your checker learnt a new trick

Open `check.py` and press **Run** — same report as Lab 03, eleven rows again. New this week: your programs now take *different paths* depending on their input, and the checker only types **one** set of answers. A PASS therefore proves that *one* branch is right. The other branches are your job: run the file yourself and type answers that send it down the road the checker never travelled. That is exactly what W2 just showed you.

2 Two-Way Decisions: `if/else` (~30 min)

Ref: Lecture 5 — Conditions

An `if` on its own has one exit for `True` and no plan for `False`. Adding `else` gives the false case somewhere to go. The guarantee is worth memorising: **exactly one body runs — never both, never neither.**



W4. EASY Trace this on paper — do not run it yet. Write the output line by line, then check yourself in Thonny:

```
fuel = 8
if fuel >= 10:
    print("Ready to go")
    print("Doors closing")
else:
    print("Refuel first")
print("Log entry saved")
```

Output: ....
How many of the four `print` lines belong to a branch, and how many run whatever `fuel` holds?

W5. MEDIUM `=` binds, `==` asks. Label each line **I** (an instruction that stores something), **Q** (a question with a `True`/`False` answer), or **X** (illegal where a condition belongs):

	`count = 5`	`count == 5`	`count != 5`	`5 == count`	`if count = 5:`
I / Q / X	.....	.....	.....	.....	.....

Type the last one into the Shell. Python does not answer `False` — it refuses to run the line at all. Which error? .....

W6. MEDIUM The Week 3 crash. Type this, run it, and answer `20` at the prompt:

```
age = input("Age: ")
if age >= 18:
    print("Adult")
```

Read the last line of the error out loud. Python is comparing a ..... with an ....., and it will not guess which you meant. Fix the line so it works — the fix is one word long and you learnt it in Lab 03: .....

TASK 3 Exact Change

EASY `task03.py` ~10 min

The canteen cashier needs a two-way decision. `task03.py` reads the amount paid and the bill total (the checker types `500` and `340` — **do not edit the `input` lines**). Print **one** line:

```
Amount paid: 500
Bill total: 340
Change: Rs 160
```

and, when the customer is short, `Short by Rs <amount>` instead. Both amounts come from an f-string — and watch the direction: the two branches subtract the opposite way round.

TASK 4 Metro Gate

MEDIUM `task04.py` ~12 min

The Orange Line gate decides whether to open. `task04.py` reads a card balance and a fare (the checker types `80` and `50`). Each branch prints **two** lines — and both lines must sit inside the same branch, indented alike:

```
Card balance: 80
Fare: 50
Gate open
Remaining: Rs 30
```

When the balance falls short, print `Gate closed` and then `Add Rs <shortfall>`. *A balance of*

exactly the fare still opens the gate — so which of `>` *and* `>=` *do you want?* Run it yourself with a balance of `50`, then `20`, and watch both branches work.

3 Many Doors: the `elif` Chain (~30 min)

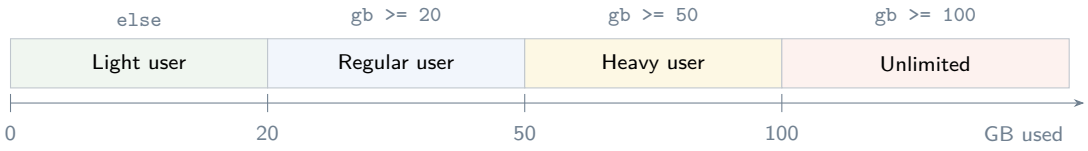
Ref: Lecture 5 — Conditions

Some decisions have four answers, not two. Stacking separate `if`s does not work: Python tests every one of them independently, so several bodies run and the last one quietly overwrites the rest. `elif` chains the tests together instead.

First true wins — then the chain stops

Python walks an `if/elif/else` chain from the top, stops at the **first** condition that is `True`, runs that one body, and skips every remaining test *without even looking at it.* Exactly one body runs — or none at all, if there is no `else`.

A chain is really a set of bands laid out on a number line, and each band is only correct because the ones above it already claimed their share:



W7. EASY With `gb = 45`, trace the chain that matches the picture — *most demanding band first*:

```
if gb >= 100:
    print("Unlimited")
elif gb >= 50:
    print("Heavy user")
elif gb >= 20:
    print("Regular user")
else:
    print("Light user")
```

Output: .....
Now count carefully: how many of the three conditions did Python actually evaluate before it printed? ..... And how many did it never look at? .....

W8. MEDIUM Same four bands, written bottom-up by a careless programmer:

```
if gb >= 20:
    print("Regular user")
elif gb >= 50:
    print("Heavy user")
elif gb >= 100:
    print("Unlimited")
```

With `gb = 250` this prints ....., which is wrong.
Worse: two of these branches can *never* run, for any value of `gb` at all. Which two, and why?
.....
The rule in one line: **most demanding condition first.**

W9. MEDIUM A chain with no `else`. Predict what happens, then run it:

```
score = 12
if score >= 90:
    grade = "A"
elif score >= 50:
    grade = "P"
print(grade)
```

It does not print a blank line, and it does not print `None` — it crashes with a .....
Explain why in one sentence: .....
A name that is only bound inside a branch does not exist when that branch is skipped.

TASK 5 Postage Bands — MEDIUM — task05.py — ~12 min

The campus post office charges by weight: up to 250 g costs Rs 80, up to 500 g Rs 140, up to 1000 g Rs 220, anything heavier Rs 350 ("up to 250" includes exactly 250). `task05.py` reads a weight in grams — the checker types `640`:

```
Parcel weight in grams: 640
Postage: Rs 220
```

Use **one** `if/elif/elif/else` chain that binds `postage`, then **one** `print` after the chain — not four `print` statements inside it. These bands run smallest-first, so "most demanding first" means the *tightest* test leads. Test all four bands yourself: try 250, 500, 1000, 1001.

TASK 6 Speed Trap — MEDIUM — task06.py — ~12 min

A speed camera hands a reckless driver a Rs 500 warning. `task06.py` runs without crashing, has the right numbers in the right order, and is still wrong — the checker types `95` and the verdict must be:

```
Speed in km/h: 95
Reckless: Rs 5000
```

Trace it on paper first with `speed = 95` and write down what `verdict` holds after each of the three `if`s: ....
Now you can see the bug. The fix is four characters long, in two places. *While you are there: which `if` does that lonely `else` currently belong to?*

4 Decisions Inside Decisions (~25 min)

Ref: Lecture 5 — Conditions

A body may hold anything — including another `if`. The inner decision is only ever reached when the outer one has already been answered, which is exactly what you want when the second question makes no sense until the first is settled.

W10. MEDIUM Trace by hand, with the values as written:

```
weight = 30
size = "large"
if weight <= 25:
    if size == "large":
        print("Cabin bag")
    else:
```

```
        print("Cabin bag, small")
else:
    print("Check-in required")
print("Tag printed")
```

Output: ....
How many times was `size` looked at? .....
`print("Tag printed")` sits at the left margin. Which `if` owns it? .....

W11. MEDIUM Same rule, two shapes. Both accept a traveller only when the passport *and* the ticket are in order:

```
if passport == "yes":
    if ticket == "yes":
        print("Board")
    else:
        print("No ticket")
else:
    print("No passport")
```

```
if passport == "yes" and ticket == "yes":
    print("Board")
else:
    print("Turned away")
```

For a traveller who may board, do the two versions agree? .....
A traveller is turned away. Which version can tell them *what* to go and fetch?
.....
Nest when each failure deserves its own answer; use `and` when one refusal covers them all.

TASK 7 Exam Hall Door — MEDIUM task07.py ~12 min

The invigilator runs two checks, and the second only makes sense once the first has passed: no ID card ⇒ `No entry without your ID card`; ID card but phone still in a pocket ⇒ `Deposit your phone first`; both in order ⇒ `Enter the hall`. Everyone who reaches the door is told when the exam starts. The checker types `yes` then `no`:

```
ID card? (yes/no): yes
Phone deposited? (yes/no): no
Deposit your phone first
Exam starts at 9:00
```

The phone question is only *asked* when there is an ID card, so its `input()` line belongs **inside** the first branch — a student with no card is never asked about a phone. Run the file yourself and answer `no` first: if you are still asked about a phone, your `input()` is at the wrong indentation.

5 Compound Conditions: `and`, `or`, `not` (~30 min)

Ref: Lectures 4 & 5 — Boolean Expressions, Conditions

One `if` can carry a whole sentence of requirements. `and` is strict (*both*), `or` is generous (*either*), and `not` flips whatever comes next.

The pecking order, once more

arithmetic → comparisons → `not` → `and` → `or`

So `not x > 5` means `not (x > 5)`, and `a or b and c` means `a or (b and c)`. Parentheses overrule everyone and cost nothing — bracket your meaning.

W12. EASY Two booleans, `has_id` and `paid_fee`. Write the condition for each rule:

The rule	The condition
both are in order	`if` ..... :
at least one is in order	`if` ..... :
the fee is unpaid	`if` ..... :
neither one is in order	`if` ..... :

The last row has two good answers — one built from `and`, one from `not` wrapped around an `or`. Write both if you can see them.

W13. MEDIUM Apply the pecking order. Bracket first on paper, then evaluate, then check in the Shell:

Expression	Your prediction	Shell says
`not 5 > 3 or 2 == 2`	.....	.....
`3 + 4 > 5 and not False`	.....	.....
`False and False or True`	.....	.....
`not (5 > 3 or 2 == 2)`	.....	.....

Rows one and four use the same three pieces and disagree. The brackets are the whole difference.

W14. MEDIUM Ranges. Predict each, then verify:

Expression	Your prediction	Shell says
`9 <= 9 <= 17`	.....	.....
`9 <= 20 <= 17`	.....	.....
`20 >= 9 and 20 <= 17`	.....	.....
`"Monday" != "Sunday"`	.....	.....

Rows two and three are the same question written two ways. The chained form `9 <= hour <= 17` is shorter *and* reads like the rule it encodes — prefer it.

W15. MEDIUM The trap that never fires an error. In the Shell, set `day = "Monday"` and evaluate:

```
day == "Saturday" or day == "Sunday"
day == "Saturday" or "Sunday"
```

....

The second line looks like English and is always `True` — for *every* day. Recall Lab 03: a non-empty string is truthy, so `or "Sunday"` is `or True`. Write the rule you will remember: each side of `or` must be a .....

Say the whole comparison, every time

`if score > 50 or 60:` and `if day == "Sat" or "Sun":` are not syntax errors — they are silent logic bugs that quietly make the condition always `True`. English lets you drop the repeated part; Python does not. Write `score > 50 or score > 60`, or use a chained comparison where one fits.

TASK 8 Lab Hours

MEDIUM task08.py ~12 min

The computer lab is open when the day is not Sunday **and** the hour is from 9 to 17 (both endpoints count). `task08.py` reads a day and an hour — the checker types `Sunday` and `11`:

```
Day: Sunday
Hour (0-23): 11
Lab closed
```

Two requirements, both non-negotiable: **one** `if` carrying **one** compound condition (an `or` would open the lab on Sunday), and the hour test written as the chained comparison `9 <= hour <= 17`. Then check the corners yourself: `Monday/9`, `Monday/18`, `Sunday/11`.

TASK 9 Canteen Closing

MEDIUM task09.py ~10 min

The canteen shuts on Sundays, and it also shuts on exam days — either reason is enough on its own. `task09.py` reads the day and whether it is an exam day; the checker types `Monday` and `yes`:

```
Day: Monday
Exam day? (yes/no): yes
Canteen closed
```

Build it in two steps, in this order: bind **one** boolean variable named `closed` from **one** `or` expression, then write the decision as `if not closed:` with `Canteen open` in that branch. Yes, you could flip the branches and drop the `not` — do it this way anyway. Reading a condition that opens with `not` is a skill, and the quiz will ask for it.

6 Capstone Challenges (~35 min)

Ref: Lecture 6 — Problem Set (J1)

Two contest-style problems, in the shape Lecture 6 used: read the given values, compute, decide, print *exactly* what is asked — no decoration, no extra lines. Between them they use everything from today: an always-printed line, an `if/else` that guards arithmetic, an `elif` chain, and a nested tie-break. Finish them, run `check.py` one last time, and go show off your **11 of 11**.

TASK 10 Bench Shortage

HARD task10.py ~15 min

Convocation seating: a long bench seats **6**, a short bench seats **4**, and **50** guests are expected. `task10.py` reads the two bench counts (the checker types `5` and `3`) and prints two lines:

```
Long benches: 5
Short benches: 3
Seats: 42
Standing: 8
```

Line 1 always prints — it is not part of any branch. Line 2 is the decision: when the seats cover everyone, print `Everyone seated` instead.

- **The trap:** with 12 long benches there are 72 seats and 22 to spare. `Standing: -22` is not an answer. Your `if/else` is what prevents it — and the exact 50-seat case must land on `Everyone seated` too.

- **No magic numbers:** 6, 4 and 50 are already bound to named constants at the top of the file. Use the names.

Test three inputs before you leave: `5 3` (short), `7 2` (exactly 50), `12 0` (spare).

TASK 11 **Sports Day Verdict** HARD `task11.py` ~18 min

Ravi and Chenab finish the athletics meet. A gold is worth 5 points, a silver 3, a bronze 1. The winner is decided in this order:

1. more points wins;
2. if the points are level, more **golds** wins;
3. if the golds are level too, it is a `TIE`.

`task11.py` reads six numbers — Ravi's golds, silvers and bronzes, then Chenab's. The checker types `4 1 2 3 3 1`:

```
Ravi golds: 4
Ravi silvers: 1
Ravi bronzes: 2
Chenab golds: 3
Chenab silvers: 3
Chenab bronzes: 1
Ravi: 25
Chenab: 25
RAVI
```

Both houses scored 25, so rule 1 cannot settle it and rule 2 must: Ravi took four golds to Chenab's three.

- **Shape:** an outer `if/elif/else` on the points. The tie is not resolved until you are *inside* the branch where the points are level — that is where a second, nested chain belongs.
- **Rule:** never type `25`, and never type `RAVI` as a decision *you* made. Every number is computed and every verdict is chosen by a condition.

Sanity check before you call it done: swap the two houses' six numbers around and run it again. The verdict should read `CHENAB`. If it still says `RAVI`, one of your comparisons is pointing the wrong way.

Self-Assessment Checklist

Before you leave, honestly tick what you can now do:

☐ I can write an `if` statement with a colon and a body indented by four spaces.

☐ I can tell, by looking at the indentation alone, which lines belong to a branch and which run every time.

☐ I know `if/else` runs exactly one body — never both, never neither.

☐ I can put two or more statements in one branch and keep them aligned.

☐ I can read the two errors an `if` produces — `SyntaxError` for a missing colon, `IndentationError` for a missing indent — and fix each on sight.

☐ I know `=` binds a name and `==` asks a question, and that `if x = 5:` is not legal Python.

- ☐ I remember that `input()` hands me a string, and I cast it before comparing it with a number.
- ☐ I can write an `elif` chain and I know only the first true branch runs.
- ☐ I put the most demanding condition first, and I can spot a branch that can never be reached.
- ☐ I know that a chain with no `else` may run no branch at all — and that a name bound only inside a skipped branch does not exist.
- ☐ I can nest one `if` inside another, and I know when to nest rather than join with `and`.
- ☐ I can combine tests with `and`, `or` and `not`, and I bracket when the pecking order is not obvious.
- ☐ I write `day == "Sat" or day == "Sun"` in full, because `or "Sun"` is always `True`.
- ☐ I can write a range test as a chained comparison, e.g. `9 <= hour <= 17`.
- ☐ I test **every** branch of my own program, not just the one the checker types answers for.
- ☐ I can read a `check.py` report and fix a `FAIL` or a `CRASH` on my own.