Lab 03 — Talking to Python

Strings, Input & Output, and Booleans

DURATION ~3.5 hours COVERS Lectures 3 & 4 TASKS 13 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 1–13) are the deliverable. Each one is its own file — `task01.py` to `task13.py` — inside your `lab03` 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.
- **Predict before you run.** Write your prediction on paper first; only then let the computer judge it. This is where the learning happens.
- **Type the code yourself.** No copy-paste. Your fingers are learning a language.
- Run `check.py` whenever you like. Your goal before leaving: **all 13 tasks PASS**. Show the final report to your instructor.
- This lab assumes Lab 02: binding variables to descriptive names, and casting with `int()`, `float()`, `str()`.
- **Done early?** Part B on the last pages is for you. It is optional and unmarked — and harder than anything above.

Before you start: get the files

Download the `lab03` folder (from the course LMS page) onto your machine. It contains `task01.py` ... `task13.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 Strings: Glue, Repeat, Measure (~20 min)

Ref: Lecture 3 — Strings and I/O

A string is a sequence of case-sensitive characters between quotes. The operators `+` and `*` work on strings too — but there they mean *glue* and *repeat*, not arithmetic.

W1. EASY In the Shell, set `dot = "."` and `dash = "-"`, then predict each value *in writing* before checking:

Expression	Your prediction	Shell says
`"key" + "board"`	.....	.....
`dot + 3 * dash`	.....	.....
`"7" + "1"`	.....	.....
`7 + 1`	.....	.....

The last two rows look like twins. Why aren't they?

W2. EASY Predict `len("selfie")`, `len("game over")`, and `len("")`, then verify. Does `len` count the quotes? Does it count the space?

W3. MEDIUM With `tag = "yo"` and `times = "4"`: predict `tag * int(times)`, then try it. Now try `tag * times` — read the last line of the error message out loud. What did Python refuse to do?

TASK 1 String Machines — EASY `task01.py` ~8 min

Open `task01.py`. It lists five string expressions. Print the value of each, **one per line, in the given order** — and write the *expression* inside `print( )`, never the answer itself. The first line is already done. **Predict all five results here first** — exactly as Python will print them (is `"5" + "5"` ten, or something else?):

....

Only then run it.

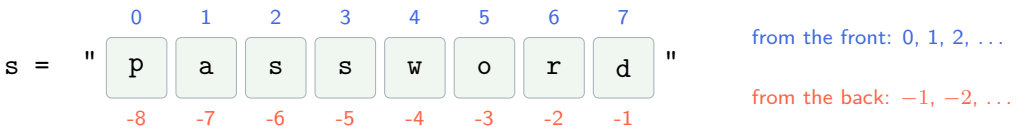
Your checker learnt a new trick

Open `check.py` and press **Run** — same report as Lab 02, now with thirteen rows. New this week: four tasks ask the user questions with `input()`. The checker answers them *automatically*, typing the exact sample values shown in each task (like `Hamza` and `3`). So your output must match the task's sample run — nothing more, nothing less.

2 Pick a Letter, Slice a Word (~30 min)

Ref: Lecture 3 — Strings and I/O

Every character in a string has an **index**: count from 0 at the front, or from −1 at the back.



W4. EASY Using `s = "password"` (diagram above), predict, then verify in the Shell:

Expression	Your prediction	Shell says
`s[0]`	.....	.....
`s[4]`	.....	.....
`s[-1]`	.....	.....
`s[-8]`	.....	.....
`s[8]`	.....	.....

One of these rows is a trap — which one, and what does Python say?

W5. MEDIUM Slicing: `s[start:stop]` takes characters from `start` up to *but not including* `stop`. With `s = "dark mode"` (count the space!), predict, then verify:

Expression	Your prediction	Shell says
`s[0:4]`	.....	.....
`s[5:8]`	.....	.....
`s[5:]`	.....	.....
`s[::-1]`	.....	.....
`s[8:5]`	.....	.....

W6. MEDIUM In the Shell, set `s = "bug"` and try `s[0] = "h"`. Read the error: strings are **immutable** — you can read any character, but never overwrite one. Now build `hug` *from* `s` in one line, using `+` and a slice. Write your line here: .....

TASK 2 Pick and Chop

EASY `task02.py` ~10 min

Open `task02.py`. It binds `word = "fundamentals"` and asks for five printed lines: the first character, the last character (negative index!), the slice that says `dam`, the first three letters, and the whole word reversed. **Every line must be built from `word`** — if a letter of the answer appears between quotes in your code, you are doing the chopping that Python should do.
Before you code, count along `fundamentals` and fill in the numbers each line needs (the first is done):
`word[0]` last: `word[.....]` dam: `word[.....:.....]` first three: `word[.....:.....]` reversed: `word[.....]`

TASK 3 Word Surgery

MEDIUM `task03.py` ~10 min

Strings can't be edited — but new ones can be built from old ones. `task03.py` binds `word = "python"`. Using only `word`, slices, and `+`, print: `Jython` (first letter swapped), `ytho` (first and last letters dropped), and `python-python` (the word, a hyphen, the word again). The letters `y t h o n` must never appear inside quotes in your file — that is the whole game.
Before you code, write each expression here. Every one is a slice of `word`, possibly glued to something with `+`:
Jython ..... ytho ..... python-python .....

3 Printing, Three Ways (~30 min)

Ref: Lecture 3 — Strings and I/O

Mixing words and numbers in one line of output is Week 2's bread and butter. Python gives you three tools: commas (which add spaces), `+` (which glues strings only), and f-strings (which evaluate whatever sits inside `{ }`).

W7. EASY Type these four lines into the **Code Editor** as a script and run it:

```
print("ping")
print(4 * 15)
4 * 15
"ping" * 2
```

Which lines actually appear in the output window? Cross out the silent ones above. (Lab 02 said it: *scripts must print.*)

W8. EASY With `a = "Level"`, `b = 4`, `c = "unlocked"`, predict the *exact* output of each line — spaces included — then run both:

Statement	Exact output
`print(a, b, c)`	.....
`print(a + str(b) + c)`	.....

Why does the second line need `str(b)` at all?

W9. MEDIUM The f-string trap the lecture warned you about — new numbers. With `lives = 3`, predict what each line prints, then run all three:

```
print(f"Lives doubled:2*lives lives")
print("Lives squared:", lives**2, "lives")
print(f"Lives tripled:{lives*3} lives")
```

....

One of them is wrong on purpose. What is missing?

TASK 4 Top-Up Lines

EASY `task04.py` ~10 min

Your mobile network wants tidy top-up receipts. `task04.py` binds `unit`, `price`, and `gigs`. Print the same data bundle three ways: line 1 with **commas** (`Bundle:  4 GB`), line 2 with **`+` and `str()`** (`Total:  Rs 300`), line 3 as **one f-string** (`4 GB costs Rs 300`). Let Python compute the 300 — never type it yourself.

Before you code, write lines 2 and 3 in full here — every `str()`, every brace:

line 2 .....

line 3 .....

TASK 5 The Broken Scoreboard

MEDIUM `task05.py` ~10 min

`task05.py` is a game's end-of-match screen, and it **crashes**. Run it once and read the error calmly — it is the same `TypeError` you met in W3. Then fix line A using `str()`, and fix line B by rewriting it as an f-string, so the program prints:

```
Wins: 4
XP earned: 120
Play again?
```

The last line was never broken — leave it alone.

Before you run it, read the two marked lines and predict: which one crashes first, A or B? ..... Which error name will the last line of the message show? .....

Does `Play again?` get printed before the crash? ..... — now run it and see.

Every piece you glue must be a string

`"XP earned:  " + 120` crashes: `+` between a string and a number is neither maths nor gluing, so Python refuses. Either cast — `"XP earned:  " + str(120)` — or skip the problem entirely with an f-string: `f"XP earned:  {120}"`.

4 Programs that Listen (~35 min)

Ref: Lecture 3 — Strings and I/O

So far your programs only spoke. `input()` lets them listen: it prints a prompt, waits for the user to type something and press Enter, and hands your program what was typed.

input() always returns a string

Whatever the user types — even `42` — arrives as the *string* `"42"`. If you want to do maths with it, cast it first: `int(...)` or `float(...)`. Forgetting this is the single most common bug of Week 2.

W10. EASY Type this two-line script in the Code Editor and run it **twice** — first answer `sleepy`, then answer `5`:

```
mood = input("Your mood today: ")
print(2 * mood)
```

Output for `sleepy`: ..... for `5`: .....
Why `55` and not `10`? One phrase: .....

W11. MEDIUM Now rename the variable to `x` and suppose the user enters `6`. Predict each line, then extend your script and verify:

Statement	Your prediction	Python says
`print(x + x)`	.....	.....
`print(int(x) + int(x))`	.....	.....
`print(float(x) * 2)`	.....	.....

TASK 6 Sign-Up Screen

MEDIUM `task06.py` ~12 min

TuneBox, a music-streaming app, sells Premium by the month. `task06.py` already asks the two sign-up questions — **do not edit the `input` lines**; the checker types `Hamza` and `3` at exactly those prompts. Premium is Rs 350 per month. Cast, compute the price, and print so a run looks like:

```
Your name: Hamza
Months of Premium: 3
Welcome to TuneBox, Hamza!
3 months of Premium: Rs 1050
```

Use f-strings for both printed lines — after Task 4 you have earned them.
Before you code: right after the second `input()` line runs, does `months` hold `3` or `"3"`? ..... And if Iman signs up for 5 months, write the last line your program should print: .....

TASK 7 The 777 Trap

MEDIUM `task07.py` ~10 min

The lecture's trap, with your own numbers. `task07.py` asks how many hours you slept last night (the checker types `7`). **Write your three predictions here first:**
....
Then make the file print `x * 3`, then `int(x) * 3`, then the f-string line `Squared:49` (the square, computed inside the braces — note: no space after the colon). Let `check.py` settle any argument

between you and Python.

5 Booleans: Yes/No Questions (~30 min)

Ref: Lecture 4 — Boolean Expressions

A boolean expression asks a question, and Python answers with one of exactly two objects: `True` or `False`. You met the `bool` type in Lab 02's Type Detective — now you get to generate them yourself with the six comparison operators `==` `!=` `<` `>` `<=` `>=`.

= binds, == asks

`x = 5` is an *instruction*: bind 5 to the name `x`. `x == 5` is a *question*: is the value of `x` equal to 5? Mixing them up is the classic Week 2 bug — and Python will not always catch it for you.

W12. EASY Predict `True` or `False`, then verify in the Shell:

Expression	Your prediction	Shell says
`4 < 9`	.....	.....
`7 <= 7`	.....	.....
`8 != 8`	.....	.....
`"lol" == "LOL"`	.....	.....

The *"or equal to"* in `<=` matters — and so do capital letters.

W13. MEDIUM Both sides of a comparison are full expressions — Python evaluates each side first, *then* compares. Predict, then verify:

Expression	Your prediction	Shell says
`(4 * 3) == 12`	.....	.....
`len("selfie") > 6`	.....	.....
`max(2, 9) == min(9, 14)`	.....	.....
`"meme" < "movie"`	.....	.....
`"Zebra" < "apple"`	.....	.....

Strings compare like dictionary entries — but every uppercase letter comes before every lowercase one.

W14. MEDIUM Predict `0.1 + 0.7 == 0.8`, then check it. Surprised? Now type just `0.1 + 0.7` in the Shell and look closely at the answer.

Floats are never exactly equal

`0.1 + 0.7` comes out as `0.7999999999999999` — computers store decimals approximately. Never ask two floats whether they are `==`; ask whether they are *close*: `abs(a - b) < 0.001`. Task 13 does exactly this.

W15. MEDIUM Real comparisons rarely involve two literals — they ask about *variables*. You are mid-game: in the Shell bind `score = 80`, `lives = 4`, `player = "Amna"`, then predict each line *before* you press Enter:

Expression	Your prediction	Shell says
`score * lives > 300`	.....	.....
`lives == 4.0`	.....	.....
`player == "amna"`	.....	.....
`score - lives * 20 == 0`	.....	.....
`len(player) < lives`	.....	.....
`score / lives == 20`	.....	.....

Two of these land *exactly* on a boundary. Boundaries are where `<` and `<=` part ways — and where most exam marks on booleans are lost.

W16. HARD Now the other direction. Each sentence is a yes/no question about the same three variables. Write it as **one comparison** (no `and`/`or` yet). No Shell until all six are written; then run each and check that Python's answer is the one the sentence deserves.

Question	Python
score times lives is at least 500	.....
the player is not called Bilal	.....
the player's name has more than three letters	.....
fewer than five lives are left	.....
the score is a whole number of tens	.....
the player's name starts with a capital A	.....

At least is `>=`, *more than* is `>`, *fewer than* is `<`, *at most* is `<=`. Get the words right and the operator follows. The last two need `%` and an index — Lectures 2 and 3 are still in play.

W17. HARD **The detective version of "= binds, == asks".** Start with `lives = 3`. For each line, predict what the Shell prints *and* what `lives` holds afterwards. Then run them in this exact order.

Line	Shell prints	`lives` is now
`lives == 4`	.....	.....
`lives = 4`	.....	.....
`lives == 4`	.....	.....
`lives = lives == 4`	.....	.....
`type(lives)`	.....	.....

Line 4 is legal Python, and nothing warns you: the `==` runs first, then `=` binds its *answer*. Which is why you type `==` on purpose, every time you mean to ask.

TASK 8 True Detective MEDIUM `task08.py` ~12 min

Open `task08.py`. Six expressions await a verdict. **Write all six verdicts here first**, in order:
....
Then print each expression, one per line — **never** type the words `True` or `False` yourself; Python is the judge, not you. Three of the six are traps you have already met in this section: a boundary, a capital letter, and a float.

6 Logic: and, or, not (~30 min)

Ref: Lecture 4 — Boolean Expressions

Real questions are compound: *is the battery low **and** is the charger at home?* Python combines booleans with `and`, `or`, and `not` — remember, `and` is strict (*both!*) while `or` is generous (*either!*).

The pecking order

Python settles a crowded expression in a fixed order:

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

Parentheses overrule everyone — and they cost nothing. When in doubt, bracket your meaning.

W18. EASY Fill both result columns from memory — no Shell until you have committed:

A	B	A and B	A or B
True	True	.....	.....
True	False	.....	.....
False	True	.....	.....
False	False	.....	.....

W19. MEDIUM `not` grabs only the very next value. Predict, then verify:

Expression	Your prediction	Shell says
`not True and True`	.....	.....
`not False or False`	.....	.....
`False or True and True`	.....	.....

W20. MEDIUM On paper: add parentheses showing the order Python uses, then evaluate step by step (no Shell yet):

`not 8 > 2 or 6 > 1`

.....

`10 - 4 > 5 and not True`

.....

W21. MEDIUM Casting to `bool`: predict each, then verify. Two traps hide here.

	`bool(7)`	`bool(0.0)`	`bool("0")`	`bool("")`	`bool(None)`
Prediction	.....	.....	.....	.....	.....
Shell says	.....	.....	.....	.....	.....

W22. MEDIUM Now with names in the mix. A fitness watch logged one lazy Sunday: bind `steps = 2600`, `floors = 2`, `day = "Sunday"`, `streak = False`. Predict, then verify:

Expression	Your prediction	Shell says
`steps > 2000 and floors > 2`	.....	.....
`day == "Sunday" or floors == 5`	.....	.....
`not streak or steps < 1000`	.....	.....
`streak and not day == "Sunday"`	.....	.....
`steps % 2 == 0 and floors % 2 == 0`	.....	.....
`not (steps > 2000 and floors > 2)`	.....	.....

Each side of `and`/`or` is settled on its own first, then the two answers are combined. Line 4: `not` waits for `==` (comparisons come first in the pecking order), so it flips the answer to `day == "Sunday"` — not `day` itself.

TASK 9 **Logic Lines** MEDIUM `task09.py` ~10 min

Open `task09.py`. Five compound expressions, five verdicts — W18–W22 were your training. **Write all five verdicts here first**, in order:

. . . .

Condition	Opposite (no `not`)	Test with
`steps >= 5000`	.....	`steps = 5000`
`day == "Friday"`	.....	`day = "friday"`
`floors > 0 and steps > 0`	.....	`floors = 0, steps = 5`
`x < 10 or x > 20`	.....	`x = 10`
`not (streak and floors >= 2)`	.....	`streak = True, floors = 2`

Proof of work: for *any* values, `original == (not opposite)` must print `True`. Try it on the third row with three different pairs of values.

W25. HARD **Always, never, or depends?** Some conditions are `True` for every `x`, some for none. Decide on paper first — then try to find a counter-example in the Shell.

Expression	always / never / depends
`x > 5 or x <= 5`	.....
`x < 1 and x > 10`	.....
`x != 3 or x == 3`	.....
`x > 0 and x < 10`	.....
`not (x > 3) or x > 3`	.....
`x == 5 and not x == 5`	.....
`x > 5 and x > 3`	.....
`x >= 5 or x > 5`	.....

The "never" rows are Lecture 4's Pitfall 4 in disguise: an `and` between two regions that do not overlap. The last two rows depend on `x` — but each collapses to a *single* comparison. Write them: ....

TASK 10 Say It in Python

MEDIUM `task10.py` ~12 min

Open `task10.py`. It binds `steps = 6400`, `floors = 3`, `day = "Saturday"`, `streak = False` — one Saturday on a fitness watch. Six rules follow, in English. Turn each into **one boolean expression bound to a well-named variable**, then print the six verdicts with f-strings so the run reads:

```
Active day: True
Weekend: True
Streak badge: False
Round steps: False
Odd floors: True
Not Friday: True
```

The rules: an **active day** is three or more floors climbed *and* more than 5000 steps; the **weekend** is Saturday or Sunday; a **streak badge** needs a streak going *and* at least 2000 steps; **round steps** is a multiple of 1000; **odd floors** means what it says; **not Friday** is any day but Friday. **Before you code**, write the six expressions here:

active day ..... weekend .....
streak badge ..... round steps .....
odd floors ..... not Friday .....

Once `check.py` passes, rebind the four variables to a slow Friday — `steps = 2000`, `floors = 4`, `day = "Friday"`, `streak = True` — predict all six on paper, and run again. Each day hides one boundary value; if a verdict surprises you, your operator is one "equals" off. (Put the Saturday values back before the final check.)

TASK 11 The Broken Concert Gate

HARD `task11.py` ~15 min

A concert's gate program, `task11.py`, decides who gets in. Five verdict lines, and every one is wrong in a different way. The five bugs, in scrambled order: an `=` that should be `==`; an `and` that can never be `True`; a string compared to a number; a `not` that grabs too little; and an `or` with only half a question on its right. Fix all five so the run reads:

```
Adult: True
Weekend: True
Late show: True
Ticket ok: True
Turned away: False
```

Before you fix anything, run the file once and read the crash — one bug is a `SyntaxError`, and Python names its line. Then go line by line: next to each verdict, write which of the five bugs it is *before* you edit it. Fix that one line, run `check.py`, and watch the report change one row at a time.

Line B is the sneaky one: with `day = "Sunday"` it does not crash and does not print `False` — it prints `Sunday`. Explain why in one line before you fix it: .....

Every side of `and`/`or` must be a whole question

`day == "Saturday" or "Sunday"` reads like English and means nothing of the sort. Python sees two operands: the question `day == "Saturday"` on the left, and the bare word `"Sunday"` on the right. A non-empty string is truthy (W21), so the right side always counts as "yes" — and `or` hands back whichever operand settled the matter, which is how a *word* ends up where a verdict should be. Spell the question out both times: `day == "Saturday" or day == "Sunday"`.

8 Capstone Challenges (~25 min)

Ref: Lectures 3 & 4 — Strings and I/O; Boolean Expressions

These two tasks combine everything from today: slices, `input()`, casting, f-strings, and boolean logic. Finish them, run `check.py` one last time, and go show off your **13 of 13**.

TASK 12 Ride Checker

HARD `task12.py` ~12 min

A theme park puts a checker at the front of its ride queues. `task12.py` reads an age and a height in cm (the checker types `10` and `150`). Compute three verdicts, each as **one boolean expression bound to a well-named variable**:

1. **needs adult** — under 12 years old *or* shorter than 120 cm
2. **bumper cars** — height between 120 and 190 cm, inclusive
3. **coaster blocked** — height under 140 cm *or* over 195 cm

Then print exactly:

```
Age: 10
Height (cm): 150
Needs adult: True
Bumper cars: True
Coaster blocked: False
```

*Which verdict wants **`and`**, and which two want **`or`**? Lecture 4's number line knows.*

Before you code, the sample run is one visitor. Fill in the three verdicts for two more — both sit on a boundary on purpose:

Age	Height	`Needs adult`	`Bumper cars`	`Coaster blocked`
12	120	.....	.....	.....
35	140	.....	.....	.....

Once `check.py` passes, run the file yourself with these two and compare. A wrong `<` where `<=` was meant shows up only on the boundary.

TASK 13 Square-Root Machine

HARD `task13.py` ~12 min

Newton's recipe from Lecture 3: to find $\sqrt{a}$, take any guess g and improve it with

$$g_{\text{next}} = \tfrac{1}{2}\left(g + \frac{a}{g}\right).$$

`task13.py` reads `a` (int) and `g` (float) — the checker types `10` and `3.2`. Print the improved guess alone on line 1, then an f-string verdict on line 2: `Close enough:  True` if the new guess squared is within `0.01` of `a`. Floats are never *exactly* equal (Task 8, last line!), so "within" means `abs(...)  < 0.01`.

```
Find the square root of: 10
Starting guess: 3.2
3.1625
Close enough: True
```

Before you code, do one step of the recipe by hand for $a = 10$, $g = 3.2$ (a calculator is allowed; the Shell is not):
$a/g =$ ..... $g_{\text{next}} =$ ..... $g_{\text{next}}^2 \approx$ ..... within 0.01 of 10? .....
If your line 1 differs in the last few digits, relax — the checker tolerates that. The verdict, it does not.

Self-Assessment Checklist

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

- ☐ I can concatenate strings with `+` and repeat them with `*`, and I know `"7" + "1"` is not `8`.
- ☐ I can measure any string with `len()`.
- ☐ I can pick out one character with `[ ]`, counting from the front or from the back.
- ☐ I can slice a substring with `[start:stop]` and reverse a string with `[::-1]`.
- ☐ I know strings are immutable — I build new strings instead of editing old ones.
- ☐ I know `print` with commas adds spaces, and `+` gluing does not.
- ☐ I can drop any expression into an f-string with `{ }`, and I never forget the braces.
- ☐ I know `input()` always returns a string, and I cast it before doing maths.
- ☐ I can predict `"7" * 3` versus `int("7") * 3` without running either.
- ☐ I can evaluate all six comparison operators, on numbers and on strings.
- ☐ I know `=` binds a name and `==` asks a question.
- ☐ I can evaluate `not`, `and`, `or` in the right pecking order, and I bracket when in doubt.
- ☐ I can turn an English rule into a boolean expression, and *at least* / *more than* tell me `>=` from `>`.
- ☐ I know *inside* a range is an `and` and *outside* is an `or` — and that `x < 1 and x > 10` can never be `True`.
- ☐ I can write the opposite of a condition without `not`: flip every comparison, swap `and` with `or`.
- ☐ I know every side of `and`/`or` must be a whole question — `day == "Sat" or "Sun"` is a bug.
- ☐ I know `0`, `""`, and `None` are falsy — and that `"0"` is truthy.
- ☐ I never compare floats with `==` — I check `abs(difference)` against a tolerance instead.
- ☐ I can read a `check.py` report and fix a `FAIL` or a `CRASH` on my own.

Part B — Going Deeper (~60 min)

Optional. Unmarked. Only for after `check.py` says 13 of 13.

Finished early? Then you knew how to *write* strings and booleans before you walked in. This part is about what Python is *doing* underneath — nothing beyond Lectures 3 and 4, yet most people who "already know Python" mispredict a third of these lines. **Write every prediction before you touch the Shell.**

B1 Twenty-one lines your intuition gets wrong

	Expression	Your prediction	Shell says
1	`"a" + "b" * 3`	.....	.....
2	`"3" * 3`	.....	.....
3	`"7" * 0`	.....	.....
4	`"vlog"[10]`	.....	.....
5	`"vlog"[10:]`	.....	.....
6	`"vlog"[1:1] == ""`	.....	.....
7	`"vlog"[::2]`	.....	.....
8	`"10" < "9"`	.....	.....
9	`"10" < 9`	.....	.....
10	`"vlog" < "vlog "`	.....	.....
11	`bool("False")`	.....	.....
12	`bool(" ")`	.....	.....
13	`1 == True`	.....	.....
14	`2 == True`	.....	.....
15	`True and 7`	.....	.....
16	`0 or "vlog"`	.....	.....
17	`3 < 5 < 4`	.....	.....
18	`f"{2 + 2}" + "1"`	.....	.....
19	`int(" 7 ")`	.....	.....
20	`int("7.0")`	.....	.....
21	`0.5 + 0.25 == 0.75`	.....	.....

Read only after you have run all twenty-one

- **1–3** — `*` still binds tighter than `+` on strings, and repeating a string zero times is not an error — it is the empty string.
- **4–6** — *indexing* past the end is an error; *slicing* past the end quietly clips to what exists, and a slice with nothing between its ends is `""`. Slices never crash. Indexes do.
- **7** — the third slot of a slice is the *step*. `[::-1]` is the step you already knew; `[::2]` is its cousin.
- **8–10** — strings compare one character at a time, left to right: `"1"` beats `"9"` before the `"0"` is ever looked at, and a shorter string that matches all the way is the smaller. A string against a number: no rule exists, so Python refuses.
- **11, 12** — only the *empty* string is falsy. Any non-empty string is truthy, including a lone space and the word `"False"`. The content is never inspected.
- **13, 14** — `True` *is* the integer 1, nothing else. `2` is truthy, but it is not `True`.
- **15, 16** — `and` and `or` do not manufacture a boolean. Each returns *one of its operands* —

the one that settled the question.

- **17** — `3 < 5 < 4` means `3 < 5 and 5 < 4`. Python lets comparisons chain like maths does.
- **18** — an f-string is a string, no matter what was computed inside the braces. `"4" + "1"` is `"41"`.
- **19, 20** — `int()` forgives surrounding spaces and refuses a decimal point. Go through `float()` first if you must.
- **21** — `0.5` and `0.25` are exact in binary; `0.1` is not. "Never `==` floats" exists because you cannot tell by looking which ones are.

B2 Two more tasks — for `check.py`'s bonus rows

TASK 14 **Mirror Test** HARD `task14.py` ~15 min

`task14.py` reads one word (the checker types `racecar`). Print four lines: the word reversed; `Palindrome: True` if it reads the same both ways; `Middle letter: e`; and `Same ends: True` if the first and last characters match. Every verdict is **one boolean expression** — no `if`, no typed `True`. The middle letter must come from `len()` and `//`, not from a number you counted.

Before you code, write the expression for each line, using only `word`, `len`, slices, `//` and `==`:

reversed ..... palindrome .....

middle ..... same ends .....

Then predict your middle-letter expression for `wifi` — a word with no middle — and run it to see which side Python picked.

TASK 15 **Leap Year Verdict** HARD `task15.py` ~15 min

A year is a leap year if it is divisible by 4 — *except* century years, which must be divisible by 400. So 2024 and 2000 are leap years; 1900 and 2100 are not. `task15.py` reads a year (the checker types `2100`) and prints three verdicts, each from one boolean expression: `Divisible by 4: True`, `Century: True`, `Leap year: False`.

Before you code, fill the table by the rule above, then write the single expression for the last column — it needs `and`, `or`, and one pair of parentheses:

Year	Divisible by 4	Century	Leap year
2024	.....	.....	.....
1900	.....	.....	.....
2000	.....	.....	.....
2100	.....	.....	.....

`is_leap =` .....

Get 2100 right and you have out-thought a great deal of shipped software.

B3 Show your code, not your output

Reopen `task10.py` and `task12.py`. Every verdict should read like English when you say its name aloud (`needs_adult`, not `n`). Show those two files — not the checker — to the lab engineer and ask for one thing you could name better. `check.py` cannot read style. A person can.