Lab 02 — First Steps

Thonny, Expressions & Variables

DURATION ~3 hours COVERS Lectures 1 & 2 TASKS 10 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 1–10) are the deliverable. Each one is its own file — `task01.py` to `task10.py` — inside your `lab02` folder.
- **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 10 tasks PASS**. Show the final report to your instructor.
- The checker only reads your program's *output*. Your variable names and comments are read by a human — make them good.
- **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 `lab02` folder (from the course LMS page) onto your machine. It contains `task01.py` ... `task10.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 Meet Thonny (~20 min)

Ref: Lecture 2 — Objects, Expressions & Variables

Open Thonny. You will use two windows all semester:

Shell (bottom)	You type one line, Python answers immediately. For experiments and quick questions.
Code Editor (top)	You write a whole *script*, save it, and run it with the green **Run** button (F5). For programs you want to keep.

W1. EASY In the Shell, type `2 + 3` and press Enter. Python answers instantly — this is the Shell's job.

W2. EASY Now type `5 / 0`. Read the last line of the red message out loud. Errors are not scoldings — they are Python telling you *exactly* what it could not do. You will read many of

these; start today, calmly.

W3. EASY Build any expression of your own that evaluates to exactly `100`. Use at least two operators.

Scripts must print

The Shell shows every value automatically. A *script* shows **nothing** unless you ask with `print(...)`. If you run a file and see no output, your script probably computed things — and then kept them to itself.

TASK 1 **First Script** EASY `task01.py` ~5 min

Open `task01.py` in Thonny. Make it print **exactly** these two lines, then run it with **Run** (F5):

```
Hello, Python!
I am giving instructions to a computer.
```

Spelling, spaces, capitals, punctuation — everything counts. A computer does exactly what you say, not what you mean (Lecture 1!).

Now meet your lab checker

Open `check.py` and press **Run**. It runs all ten task files and prints one report, like this:

```
Task 01  First Script           PASS
Task 02  Calculator Practice    FAIL
         your program printed nothing
...
Passed 1 of 10. Keep going!
```

If TASK 1 says `PASS`, you are officially a programmer. The nine `FAIL`s are simply today's to-do list. If a task says `CRASH`, the report shows the error and a hint — read it, fix it, run `check.py` again. It never changes your files.

2 Expressions: Python as a Calculator (~30 min)

Ref: Lecture 2 — Objects, Expressions & Variables

An expression combines objects and operators, and always evaluates to a **single value**. Predict each value below *in writing*, then check in the Shell.

W4. EASY Fill the middle column first, then verify:

Expression	Your prediction	Shell says
`(2 + 3) - 1`	.....	.....
`2 + (3 - 1)`	.....	.....
`3 / 2`	.....	.....
`3 ** 2`	.....	.....
`type(3 ** 2)`	.....	.....
`type(4.0 * 3)`	.....	.....

W5. MEDIUM Predict what happens for `2 + [3 - 1]`, then try it. Python allows **only parentheses** `( )` for grouping — never `[ ]` or `{ }`.

W6. MEDIUM In the Shell, find the difference between `7 / 2`, `7 // 2`, and `7 % 2`. In one phrase each, write what `//` and `%` do:
`//` ..... `%` .....

TASK 2 **Calculator Practice** EASY `task02.py` ~10 min

Open `task02.py`. It lists five expressions from the lecture. 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 values here first** — written exactly as Python will print them (is it `2` or `2.0`?):
....
Only then run it.

3 Objects and Types (~30 min)

Ref: Lectures 1 & 2 — every value in Python is an object with a type

W7. EASY Predict the type (`int`? `float`? `bool`? `NoneType`? something else?), then verify with `type(...)` in the Shell:

Object	Your prediction	Shell says
`9`	.....	.....
`9.0`	.....	.....
`9 / 3`	.....	.....
`True`	.....	.....
`None`	.....	.....
`'FCC'`	.....	.....

Two traps from the lectures are hiding in this table — did you catch both?

W8. MEDIUM From Lecture 1: which of the objects above are **scalar**, and which is **non-scalar**? Circle the non-scalar one in the table.

TASK 3 **Type Detective** EASY `task03.py` ~8 min

Open `task03.py`. It lists five objects: `1234` `8.99` `9.0` `True` `None`. **Write the type of each here first:**
....
Then print the type of each, one per line. The first line is done for you and prints `<class 'int'>` — yours must follow the same pattern.

int() chops — it never rounds

`int(7.9)` is `7`, not `8`. Casting to `int` *truncates*: it throws the decimal part away. If you want proper rounding, that is a different function: `round(7.9)` is `8`.

TASK 4 **The Casting Machine** EASY task04.py ~10 min

Open `task04.py`. Five casts are listed. **Write your five predictions here first:**

....

Then print the five values, one per line, and let `check.py` settle any argument between you and Python.

4 Variables: Names for Values (~45 min)

Ref: Lecture 2 — Objects, Expressions & Variables

An assignment statement **binds a name to an object**. The `=` is an *instruction* ("bind!"), not a claim that two sides are equal.

W9. EASY Predict which of these Python will accept, then try each in the Shell:

`x = 6` ..... `x * y = 3 + 4` .....

`6 = x` ..... `xy = 3 + 4` .....

W10. EASY Tick the **valid** variable names:

☐ `lunchPrice` ☐ `fried rice` ☐ `greek-salad` ☐ `chilli_sauce` ☐ `7up`

One of the invalid ones is not even an error — Python quietly reads it as *subtraction*. Which one? .....

The next three tasks are one story: you, the campus canteen, and your chai habit.

TASK 5 **Canteen Bill** MEDIUM task05.py ~12 min

Samosas are Rs 30, chai is Rs 60. You buy 3 samosas and 2 cups of chai. Open `task05.py` and print three lines: the samosa bill, the chai bill, and the total.

Before you code, write the three numbers you expect Python to print — exactly as it will print them:

samosas ..... chai ..... total .....

The rule that matters: bind every quantity to a *descriptive* name (`samosa_price`, `chais_bought`, ...) and let Python compute everything. If a number other than a price or a quantity appears in your code, you are doing Python's job for it. If Python prints `90.0` where you predicted `90`, find out which operator turned your whole numbers into floats.

TASK 6 **Round Tabletop** MEDIUM task06.py ~12 min

The canteen is ordering new round tabletops; yours has radius 2.2 feet. Using the lecture's approximation $\pi \approx 355/113$:

```
area          = pi * (radius ** 2)
circumference = pi * (radius * 2)
```

Before you code, estimate both to one decimal place (use $\pi \approx 3$ and $2.2 \approx 2$ in your head — a rough guess is the point, not a calculator):
area ≈ ..... circumference ≈ .....
Open `task06.py` and print the area (line 1) and the circumference (line 2). Model your style on the *best-style* example from Lecture 2: descriptive names, a comment saying what the code does. If Python's answer is not in the same ballpark as your estimate, one of your formulas has the wrong shape — check the `** 2`.

TASK 7 Price Hike

MEDIUM `task07.py` ~12 min

Chai is Rs 60 and you drink 30 cups a month — then the price jumps to Rs 80. Open `task07.py`; part of the code exists. **Before running anything**, fill this table with the value each name holds *after* each line runs (if a name does not exist yet, write a dash):

After this line runs...	chai_price	monthly_chai
`chai_price = 60`	.....	.....
`monthly_chai = 30 * chai_price`	.....	.....
`chai_price = 80`	.....	.....

Now finish the file so it prints the old bill, then `monthly_chai` again after the hike, then the recomputed bill. *Does rebinding `chai_price` update `monthly_chai` by itself? That is the whole point of this task.*

5 Debugging Practice (~25 min)

Ref: Lecture 1 (syntax vs. semantics) & Lecture 2

W11. MEDIUM Classify each line: **S** = syntax error (Python refuses to run it), **M** = runs, but the meaning is wrong, **OK** = perfectly fine.

Line	S / M / OK
`6 = x`	.....
`greek-salad = 20`	.....
`total = 3 - 1` *(you meant to add)*	.....
`x = 6`	.....

Remember from Lecture 1: Python catches broken *syntax* for you. Broken *meaning* is yours to catch — no error message will ever come.

TASK 8 The Broken Swap

MEDIUM `task08.py` ~10 min

`task08.py` is supposed to swap `x` and `y` so it prints `2` then `1`. It is broken. **Trace it on paper first** — fill in the value of each name after every line:

After this line runs...	x	y
`x = 1`	.....	.....
`y = 2`	.....	.....
`y = x`	.....	.....
`x = y`	.....	.....

Your trace will show exactly where the `2` is lost. Fix the program using a third variable `temp` — and no, hard-coding `x = 2` is not a swap.

6 Capstone Challenges (~20 min)

These two tasks combine everything from today: variables, expressions, and call expressions. Finish them, run `check.py` one last time, and go show off your `10 of 10`.

TASK 9 Right-Triangle Solver

HARD `task09.py` ~10 min

A right triangle has short sides 6 and 8. In `task09.py`, print three lines:

1. the longer of the two sides — use `max`
2. the hypotenuse — use `sqrt`, remembering where it lives: `from math import sqrt`
3. the square of the hypotenuse — use `pow`

Before you code, predict all three lines exactly as Python will print them — `10` or `10.0`? That question is the whole capstone:
line 1 ..... line 2 ..... line 3 .....
If line 3 does not come out to a suspiciously familiar number, revisit line 2.

TASK 10 Calls within Calls

HARD `task10.py` ~10 min

For the expression below, **draw its evaluation tree** in the box, the way Lecture 2 drew them — innermost calls at the bottom, one value flowing up to the top:

```
mul(2 * (3 + 1), add(1, 2))
```

Then open `task10.py` and print the values of all three listed expressions (`add`, `sub`, `mul` are already imported from the `operator` module). If Python disagrees with your tree, the tree is wrong — redraw it until you see why.

Self-Assessment Checklist

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

- ☐ I can tell the Shell apart from the Code Editor, and I know which one shows values automatically.
- ☐ I can save a script and run it with **Run** (F5).
- ☐ I can read a Python error message without panicking.
- ☐ I can evaluate `/`, `//`, `%`, and `**` by hand.
- ☐ I can predict the type of a value, and check myself with `type()`.
- ☐ I know `int()` chops and `round()` rounds.
- ☐ I can bind values to well-chosen variable names.
- ☐ I can spot an illegal variable name instantly.
- ☐ I know that rebinding a name does **not** update values computed from it earlier.
- ☐ I can swap two variables using a third.
- ☐ I can import and call a function from a module (`from math import sqrt`).
- ☐ I can draw the evaluation tree of a nested call expression.
- ☐ 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 10 of 10.*

If you arrived with Python already in your fingers, the ten tasks above were a warm-up. This part is different. It uses **nothing beyond today's vocabulary** — numbers, operators, calls, names — yet almost everyone who "already knows Python" gets at least five of the lines below wrong. Knowing how to *write* Python and knowing what Python is *doing* are two different skills. This is the second one.

B1 Sixteen lines your intuition gets wrong

Same rules as before, only stricter: **write every prediction before you touch the Shell.** Then run them all. Then read the notes.

	Expression	Your prediction	Shell says
1	`0.1 + 0.2`	.....	.....
2	`0.1 + 0.2 == 0.3`	.....	.....
3	`10 / 2`	.....	.....
4	`5 // 0.5`	.....	.....
5	`-7 // 2`	.....	.....
6	`-7 % 2`	.....	.....
7	`int(-7.9)`	.....	.....
8	`-7.9 // 1`	.....	.....
9	`round(2.5)`	.....	.....
10	`round(3.5)`	.....	.....
11	`2 ** 3 ** 2`	.....	.....
12	`-2 ** 2`	.....	.....
13	`2 ** 1000` (how many digits?)	.....	.....
14	`2.0 ** 10000`	.....	.....
15	`True + True`	.....	.....
16	`1e16 + 1 == 1e16`	.....	.....

Two more, for the `from math import` crowd:

17	`sqrt(2) ** 2 == 2`	.....	.....
18	`pi = 3` then `pi`	.....	.....

Read only after you have run all eighteen

- **1, 2, 16, 17** — a `float` is a binary fraction with 53 bits of precision. `0.1` has no exact binary form, the same way 1/3 has no exact decimal form. Every float result is the *nearest representable* number, not the true one. Moral: never test floats with `==`.
- **3, 4** — `/` *always* returns a float, even when the answer is whole. And `//` returns a float if *either* side is one.
- **5, 6, 7, 8** — `//` rounds *down* (towards $-\infty$); `int()` chops *towards zero*. They agree for positives and disagree for negatives. `%` is defined so that `a == (a // b) * b + a % b` always holds — check it for `-7` and `2`.

- **9, 10** — `round()` sends exact halves to the nearest *even* number ("banker's rounding"), so that rounding a long column of halves does not drift upward.
- **11, 12** — `**` groups from the *right*, and it binds tighter than the minus sign in front. Add parentheses until you can predict both.
- **13, 14** — an `int` grows as large as memory allows. A `float` has a ceiling near 1.8×10^{308}, and past it Python raises an error rather than lie to you.
- **15** — `bool` is a kind of `int`: `True` is `1`, `False` is `0`.
- **18** — an import just binds a name. Assignment rebinds it, and the module never notices. Nothing is sacred.

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

TASK 11 **Digit Surgery** HARD `task11.py` ~15 min

`task11.py` holds `number = 4721`. Print its four digits **one per line, last digit first**: `1`, `2`, `7`, `4`. The rule: **only `//` and `%`** — no quotes, no strings, no loops, and the digits themselves must never appear in your code.
Before you code, write the expression that extracts each digit from `number`:
last ..... third ..... second ..... first .....
Test your idea in the Shell with `4721 % 10` first. Then ask: what does `4721 // 10` leave behind?

TASK 12 **Swap Without Temp** HARD `task12.py` ~10 min

Task 8 swapped two names using a third. `task12.py` binds `x = 17` and `y = 42`; swap them using **no third name at all** — exactly three assignments, using only `+` and `-`. The first is given: `x = x + y`.
Trace on paper first — fill in what each name holds after each line, and work out lines 2 and 3 before typing them:

After this line runs...	x	y
`x = 17`	.....	.....
`y = 42`	.....	.....
`x = x + y`	.....	.....
`y =`	.....	.....
`x =`	.....	.....

After line 3, `x` holds the sum. Which single subtraction recovers the old *`x` from it?*

B3 Show your code, not your output

Reopen `task05.py`, `task06.py` and `task07.py` and hold them against the *best-style* example from Lecture 2: every quantity has a descriptive name, a comment says what the code does, and no number appears that is not a price, a size, or a count. Then show those three 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, and from Lab 03 onward a person will.