

Programming Environment, Functions & Variables

Overview: what you'll produce

By the end of this lab you will have a single folder, `code/labs/lab-01/`, containing five working Python scripts, each one runnable on its own from the terminal:

- `strings.py` - cleans up and reformats text entered by the user
- `numbers.py` - converts typed input into numbers and performs arithmetic on it
- `greeting.py` - asks the user questions and prints a greeting built from variables
- `shapes.py` - defines and tests your own functions for calculating area
- `receipt_splitter.py` - a small mini-project combining everything above to split a restaurant bill

Nothing here is handed in — it's practice, so work at your own pace and don't worry about getting every exercise perfect. Some students will get through this faster than others depending on how much Python experience they already have, and that's expected. If you don't finish everything in the three hours, use your independent study time to complete the remaining exercises.

Before you start

Work through the exercises in order — each one builds on the last. Save every script inside a single folder called `code/labs/lab-01/` to keep things organised. Check that:

- ☐ VS Code (or your chosen editor) is installed and opens
- ☐ Python 3 is installed — check with `python --version` in the VS Code PowerShell terminal
- ☐ You can open a terminal and an editor side by side (in VS Code: View → Terminal → New Terminal, or press Ctrl+`)

Lab objectives

- Navigate the file system and run programs entirely from the command line
- Read user input and produce formatted output using f-strings
- Manipulate strings and perform arithmetic correctly, including type conversion
- Write custom functions with parameters, default values and return values
- Organise a small program's logic inside a `main()` function

Timing guide

Time	Exercise	Focus
0:00 – 0:10	Setup	Confirm tools installed, create working folder
0:10 – 0:40	Exercise 1	Command-line warm-up
0:40 – 1:10	Exercise 2	Hello, functions and variables
1:10 – 1:40	Exercise 3	Working with strings
1:40 – 1:50	Break	—
1:50 – 2:20	Exercise 4	Working with numbers

2:20 – 3:05	Exercise 5	Writing your own functions
3:05 – 3:40	Exercise 6	Mini project: Receipt Splitter
3:40 – 3:50	Wrap-up	Self-check, stretch goals

A note on folders: organising your work as a tree

Every file on your computer lives inside a folder, and every folder lives inside another folder, all the way up to a single root — on Windows, that's usually `C:\`. This branching structure is called a folder tree: one folder can contain many subfolders, each of which can contain more subfolders or files, but every subfolder has exactly one parent. Knowing this makes it much easier to stay oriented when you're working entirely from a terminal, since there's no file browser window to glance at.

```
CODE_HOME\
├── code\
│   └── labs\
│       └── lab-01\
│           ├── strings.py
│           ├── numbers.py
│           └── greeting.py
```

`CODE_HOME` here just stands for wherever your own code folder actually lives — that's different on every machine, and is usually kept inside a folder that syncs to a cloud storage service, so your work is backed up and reachable from more than one computer.

For this module, keep everything inside one code folder, with each week's lab in its own numbered subfolder inside labs — so this week's work goes in `code/labs/lab-01/`, next week's in `code/labs/lab-02/`, and so on. Exercise 1 below has you build exactly this structure using PowerShell, before you create a single file.

Exercise 1 — Command-line warm-up (30 min)

A note on typing commands: every shaded, blue line in this document, like `pwd` below, is something you type into the terminal yourself and then press Enter (Return) to actually run it. Nothing happens until you press Enter, so if a step seems to do nothing, check you've pressed it.

Do every step below using only the terminal — no clicking in a file browser. If you make a typo, don't worry — press Enter, read the error message, and try again.

1. Print your current working directory — this shows you the folder you're currently "in".

```
pwd
```

2. Create a folder called code, and move into it (`mkdir` makes the folder, `cd` moves you into it).

```
mkdir code
cd code
```

3. Inside code, create a folder called labs, and move into it.

```
mkdir labs
cd labs
```

4. Inside labs, create this week's folder, lab-01, and move into it.

```
mkdir lab-01
cd lab-01
```

5. List the contents of the folder to confirm it's empty (`ls` lists whatever is inside your current folder).

```
ls
```

6. Create three empty files: `strings.py`, `numbers.py` and `greeting.py` — separating the names with commas creates all three in one command.

```
New-Item strings.py, numbers.py, greeting.py
```

7. Open the whole folder in VS Code — the dot in the command below means "this folder", i.e. the one you're currently in.

```
code .
```

Checkpoint

Your `code/labs/lab-01` folder should now contain `strings.py`, `numbers.py` and `greeting.py`. Print its contents with `cat` — for example, `cat strings.py` — to double-check (they'll be empty for now).

Exercise 2 — Hello, functions and variables (30 min)

Open `greeting.py` in VS Code for this exercise.

1. Store your name in a variable, then print a greeting using an f-string.

```
name = "Ada"
print(f"hello, {name}")
```

2. Run the script from the terminal and confirm the output.

```
python greeting.py
```

3. Ask the user for their name with `input()`, and greet them instead of using a fixed value.

```
name = input("What is your name? ")
print(f"hello, {name}")
```

4. Add a second question that asks for the user's favourite language, and print both answers on one line using concatenation with `+`.
5. Rewrite the same line using an f-string instead of `+`, and compare how much easier it is to read.
6. Add a comment above each block of code explaining, in plain English, what it does.

Checkpoint

Running `greeting.py` should ask two questions and print a friendly two-part greeting built from variables.

Stretch goal: use print's `sep` and `end` named parameters to change how your two pieces of output are separated, e.g. `print(a, b, sep=", ")`.

Exercise 3 — Working with strings (30 min)

Use `strings.py` for this exercise.

1. Ask the user to enter their full name, with extra spaces before or after on purpose (e.g. " ada lovelace ").
2. Use `.strip()` to remove the extra whitespace, then print the cleaned-up result.

```
raw_name = input("Enter your name: ")
clean_name = raw_name.strip()
```

3. Use `.title()` to print the name with each word capitalised, e.g. `print(clean_name.title())`.
4. Use `.split()` to break the full name into a list of separate words, and print how many words it contains using `len()`.

```
parts = clean_name.split()
print(f"That name has {len(parts)} word(s)")
```

5. Print a short bio sentence using an f-string that embeds the cleaned, title-cased name, e.g. "Hello, Ada Lovelace, welcome to the course!"
6. Use an escape character to include a literal quote mark or a new line inside one of your printed strings.

```
print("She said, \"hello!\"")
```

Checkpoint

Your script should tidy up messy input and produce clean, well-formatted output using at least `.strip()`, `.title()` and `.split()`.

Stretch goal: use `.capitalize()` on a sentence and compare its behaviour with `.title()`.

— 10 minute break —

Exercise 4 — Working with numbers (30 min)

Use `numbers.py` for this exercise.

1. Ask the user to enter two numbers with `input()`. Remember: `input()` always returns a string.
2. Convert both answers to numbers using `float()`, then add, subtract, multiply and divide them, printing each result.

```
a = float(input("Enter a number: "))
b = float(input("Enter another number: "))
print(a + b, a - b, a * b, a / b)
```

3. Use the modulo operator `%` to find the remainder when the first number is divided by the second, e.g. `a % b`.
4. Round the division result to 2 decimal places using `round()`, and print it, e.g. `round(a / b, 2)`.
5. Now format the same division result inside an f-string so it always shows exactly two decimal places, using `f"{value:.2f}"`.
6. Deliberately enter text instead of a number when the script asks for input, run it, and read the error Python gives you. Note which line it points to.

Checkpoint

Your script should correctly convert text input to numbers, perform all four basic operations plus modulo, and display a result formatted to two decimal places.

Stretch goal: use `pos = (pos + step) % width` with made-up numbers to see how modulo can be used to make a value wrap around, like a character moving off one edge of a screen and reappearing on the other.

Exercise 5 — Writing your own functions (45 min)

Create a new file called `shapes.py` for this exercise.

1. Write a function called `area_circle(radius)` that returns the area of a circle (use 3.14 for pi). Do not hard-code the radius — it must come in as a parameter.

```
def area_circle(radius):
    return 3.14 * radius * radius
```

2. Call your function with a few different radius values and print the results, e.g. `print(area_circle(3))`.
3. Write a second function, `area_rectangle(width, height)`, that returns the area of a rectangle.
4. Give `area_rectangle` a default value of 1 for height, so it can be called with just a width.

```
def area_rectangle(width, height=1):  
    return width * height
```

5. Write a function called `describe_shape(name, area)` that returns a formatted sentence, e.g. "A circle has an area of 12.56", using an f-string. Call it using the return values from your two area functions.
6. Inside one of your functions, create a variable that only exists in that function (e.g. a working total). Try to print that variable from outside the function afterwards and observe the error — this demonstrates scope.
7. Move all of your function calls into a function called `main()`, and call `main()` at the very bottom of the file, after every other function has been defined.

```
def main():  
    print(describe_shape("circle", area_circle(5)))  
    print(describe_shape("rectangle", area_rectangle(4, 3)))  
  
main()
```

Checkpoint

`shapes.py` should define at least three functions, each with a single clear purpose, using parameters and return values rather than hard-coded numbers or print statements inside the calculation functions.

Stretch goal: add a function called `area_triangle(base, height)` and include it in your `main()` output.

Exercise 6 — Mini project: Receipt Splitter (35 min)

Bring together everything from this lab into one program, `receipt_splitter.py`, that splits a restaurant bill between friends.

Requirements:

- Ask the user for the total bill amount and the number of people sharing it, converting input to the right numeric types
- Write a function `calculate_share(total, people, tip_percent=10)` that returns the amount each person should pay, including a default 10% tip, using parameters and a return value — not hard-coded numbers
- Use string methods (e.g. `.strip()` and `.title()`) to clean up a diner's name entered by the user
- Use an f-string with two-decimal-place formatting (`:.2f`) to display the final amount per person
- Organise the program's overall flow inside a `main()` function, called at the bottom of the file

Example run:

```
Enter the total bill: 84.50  
Enter number of people: 3  
Enter the diner's name:   guido  
Guido, each person owes £30.98 (including a 10% tip)
```

Checkpoint

Running `receipt_splitter.py` should ask for the bill, the number of people and a name, then print a correctly rounded, well-formatted amount per person, using at least one custom function with a default parameter.

Stretch goals (if time remains)

- Let the user type in their own tip percentage instead of always using the 10% default
- Add input validation: if the number of people entered is 0, print a friendly message instead of crashing on a division by zero
- Use help to look up one PowerShell command you haven't used yet (e.g. [Select-Object](#) or [Get-Content](#)) and try it once

Self-check

Before you finish up, look back over what you've written — this isn't handed in, but it's worth confirming you've hit the key ideas:

- ☐ `code/labs/lab-01/` contains: `strings.py`, `numbers.py`, `greeting.py`, `shapes.py`, `receipt_splitter.py`
- ☐ Every script runs from the terminal without errors: `python <file>.py`
- ☐ Every custom function uses parameters and, where appropriate, a return value — not hard-coded numbers
- ☐ At least one comment appears above each function explaining what it does
- ☐ `receipt_splitter.py`'s logic is organised inside `main()`, called at the bottom of the file