Lab 3 · Part 1 Foundations of Computation AI Level 0

Strings and Your First Functions

Part A is seven problems on strings, formatting and functions — one built specifically to expose the print/return confusion, and one that turns Lab 2's Heron formula into a function and builds two more on top of it. Part B turns a raw line into an aligned table and runs straight into the real problem of wide characters. Part C covers running Python from a terminal, and redirection.

Due: Homework 1 · Sat 10 Oct, 23:59

What this session is for

- Get fluent with f-strings for decimals and alignment
- Handle what `input()` gives you correctly
- Write functions that take arguments and return values
- **Settle the difference between `print()` and `return` for good**

AI policy

This week is **Level 0**: no AI-generated or AI-completed code.

How marking works

The platform calls your function and compares the value it **returns**. Unless a problem says "print", always `return`. A function that prints instead hands back `None` and scores zero, however correct the algorithm is.

No loops yet

Loops come in L4 and lists in L5. Everything here needs only straight-line code and functions. `.split()` shows up once, but you only take two fixed pieces out of it — nothing to iterate over.

Part A — required

3-1 Temperature conversion

Implement `c_to_f(c)` using `F = C * 9/5 + 32`, returning a **float** rounded to one decimal place.

```
c_to_f(0)     →  32.0
c_to_f(37)    →  98.6
c_to_f(-40)   →  -40.0
```

提示

Return a number, not a string. `round(x, 1)` gives you a float; `f"{x:.1f}"` gives you a string. This problem wants the former.

3-2 Masking a student number

Implement `mask_id(sid)`: keep the **first four and last four** characters and star out everything in between.

```
mask_id("526030910001")  →  "5260****0001"
mask_id("12345678")      →  "12345678"
```

Slicing and concatenation are enough — no loop.

> **The second example is there to make you think**
>
> At exactly eight characters there is nothing in the middle to hide. What does your version do then? What about a string shorter than eight? Say how you handled it on the answer sheet.

3-3 An aligned results table

Implement `format_row(name, score)`, returning one formatted line: the name left-aligned in 10 characters, the score right-aligned in 6 with one decimal place.

```
format_row("Alex", 87.5)    →  "Alex        87.5"
format_row("Blake", 92.0)   →  "Blake       92.0"
format_row("Chen", 65.25)   →  "Chen        65.2"
```

Use an f-string with `:<10` and `:>6.1f`.

> **Width is counted in characters, not columns**
>
> An f-string field width counts **characters**, not how wide they look on screen. A Chinese character occupies two columns in a terminal but counts as one against `:<10`, so `f"{'张三':<10}"` gets padded with eight spaces and ends up visibly wider than `f"{'Alex':<10}"`, throwing the table out of line.
>
> This problem uses ASCII names only, so it will not bite you here. But know that it exists — aligning CJK text properly means measuring display width yourself, and the standard library gives you no direct way to do it.

3-4 Find and fix a print/return mistake

The code below is meant to average two numbers and double the result. It raises `TypeError`. **Say why, fix it**, and explain in a sentence or two on the answer sheet.

```python
def average(a, b):
    print((a + b) / 2)

result = average(10, 20) * 2
print(result)
```

This is the important problem this week

If it strikes you as too easy, good — that means you have it straight. But a large number of students make this exact mistake later in the term, where it is buried much deeper and far harder to see.

3-5 Reading a number safely

Implement `read_int(prompt)`: show the prompt, read a line, convert it to an integer and return it.

You do not have to handle non-numeric input yet — that needs the exception handling from L11 — but **try it once**: type `abc` and see which kind of error comes back. The answer sheet asks what `input()` returns, which is the same root cause.

提示

This problem asks you to `print` a prompt *and* `return` a value, which shows that "never print inside a function" is not the real rule. The real rule is that the **result** must be returned. A prompt is interaction, not a result.

3-6 Returning several values

Implement `stats(a, b, c)` returning the **minimum, maximum and mean** of the three numbers.

```
stats(3, 9, 5)  →  (3, 9, 5.666666666666667)
```

The caller should be able to write:

```
low, high, mean = stats(3, 9, 5)
```

Do not round the mean — return it as it comes.

3-7 Circumcircle and incircle (marked by the platform)

In Lab 2 you computed a triangle's area with Heron's formula as straight-line code. This time **write it as a function**, then build two more on top of it.

```
def area(a, b, c): ...
def circles(a, b, c): ...    # returns (circumradius R, inradius r)
```

The formulas:

```
S = the area, from Heron's formula
s = (a + b + c) / 2          circumradius R = a·b·c / (4S)
                             inradius      r = S / s
```

```
area(3, 4, 5)      →  6.0
circles(3, 4, 5)   →  (2.5, 1.0)
circles(5, 5, 6)   →  (3.125, 1.5)
```

`circles` **must call** `area` rather than repeating Heron's formula. That is exactly what this problem is about, and the marking checks for it.

You can verify the 3-4-5 case yourself

In a right triangle the circumcircle's diameter is the hypotenuse, so `R` should come out at exactly `5 / 2 = 2.5`. If it does not, go back to the formula.

Why repeating it is wrong

Repeating it works, of course. But the day you discover Heron's formula is inaccurate on very thin triangles and want to rewrite it (Lab 2's B-4 mentioned this), you now have two places to change — and one of them will be missed. **Write a thing once**: this course returns to that idea repeatedly.

Part B — going deeper: turning a raw line into an aligned table

Real data is rarely ready to use. It usually arrives as a line of text, and you have to break it apart, convert it to the right types, and lay it out so a person can read it. Four steps, each building on the last, **none of them needing a loop**.

The raw data, one record per line:

```
Alex,87.5
Blake,65.25
Christopher,92
```

B-1 Break one apart

Implement `parse(record)`, splitting a record into a name and a score, **the score as a float**:

```
parse("Alex,87.5")        →  ("Alex", 87.5)
parse("Christopher,92")   →  ("Christopher", 92.0)
```

Two hints

`"a,b".split(",")` gives you two pieces; take them with `[0]` and `[1]` — the same subscript notation L3 used for strings.

Note the second example: the input says `92` but the result must be `92.0`. Think about why that means `float()` rather than `int()`.

B-2 Lay one out

Feed the result of `parse` into your `format_row` from Part A: name left-aligned in 10 columns, score right-aligned in 6 with one decimal place.

Do it for all three records (write three calls — no loop) and paste the output onto the answer sheet.

B-3 Wide characters do not line up

Now run the same code over these three records instead:

```
张三,87.5
李四,65.25
欧阳修远,92
```

The columns go crooked. Answer:

1. What is `len("张三")`? What is `len("Alex")`?
2. Do they take up the same width on screen?
3. So what exactly is `f"{name:<10}"` counting when it pads?

A real problem, not a curiosity

f-string widths count **characters**, not display columns. A Chinese character occupies two columns in a monospaced font but counts as one character to `:<10`, so those rows get too much padding and the table skews.

You will meet this in any program that prints a table containing CJK text. The standard library has no direct answer; you have to measure display width yourself — L12 covers the proper way when it gets to Unicode.

B-4 A workaround

Without reaching for anything you have not been taught, find a way to get those three rows **roughly** aligned. Write it, and say where it breaks down.

There is no model answer

A few directions: count display width yourself and pad by hand; give CJK names their own narrower field; or abandon alignment and separate the columns some other way. **Every one of them has a flaw — naming the flaw matters more than picking the "right" one.**

Part C — fundamentals: running Python from the terminal, and redirecting output

Week 1 was moving around (`pwd` `ls` `cd`); week 2 was creating and changing things (`mkdir` `touch` `cp` `mv` `rm`). This week: **the two ways to run Python from a terminal**, and **sending output somewhere other than the screen**.

Interactive versus script

Typing `python` on its own opens the **interactive interpreter** (the `>>>` prompt), where you try one line at a time — good for checking an idea:

```
python
>>> 2 ** 100
>>> exit()          # or press Ctrl+D
```

Typing `python file.py` **runs a script** from top to bottom. That is how you run programs.

Key	What it does
`Ctrl+C`	interrupt a running program
`Ctrl+D`	leave the interactive interpreter (`Ctrl+Z` then Enter on Windows)
`↑` `↓`	scroll back through commands you have typed

Redirection: sending output to a file

```
python hello.py > out.txt      # write output to the file, replacing it
python hello.py >> out.txt     # append to the end of the file
python guess.py < input.txt    # feed the file to the program as if typed
```

Why this earns its place

Once your program prints a few hundred lines, scrolling is useless. Send it to a file and read it at your own pace with `cat` or an editor.

The `<` form is the more useful of the two: when a program calls `input()` five times, you do not want to type five values on every debugging run. Put them in a file, one per line, and feed it in.

C-1 Walk through it

Copy the commands and their output out of the terminal onto the answer sheet (no screenshot needed):

1. Open the interactive interpreter, evaluate `2 ** 100`, and leave

2. Redirect your B-2 output into `table.txt`, then look at it with `cat table.txt`
3. Create `input.txt` with two numbers on two lines, and feed it to your 3-5 program with `<`

C-2 Think it through

1. What is the difference between `>` and `>>`? How would you demonstrate it?
2. After using `>`, is the output still on screen? Why?

Before you submit

- ☐ Every function `return`s rather than `print`s (except 3-5's prompt)
- ☐ 3-1 returns a float, not a string
- ☐ 3-7's `circles` calls `area` rather than repeating Heron's formula
- ☐ The Part A answer sheet is submitted (3-2's edge case, 3-4's explanation and the three-inputs exercise are all on it)
- ☐ The Part B answer sheet is submitted
- ☐ The Part C answer sheet is submitted