Lab 2 · Part 1 Foundations of Computation AI Level 0

Characters, Bases and Floating-Point Precision

Part A covers escapes, ord/chr, bases and two calculations. Part B digs all the way into floating point and finally answers the question Lab 1 left open — why two algebraically identical formulas give different answers. Part C is files and directories from the terminal.

Due: Homework 1 · Sat 10 Oct, 23:59

What this session is for

- Control the shape of your output with escape characters
- Get fluent with `ord()` / `chr()` and `bin()` / `oct()` / `hex()`
- **Measure for yourself which digit floats start lying at**, rather than just hearing that "floats are inexact"
- Internalize the type rules for `/`, `//`, `%` and `**`

AI policy

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

No loops yet

Loops are L4 and lists are L5. **Every problem here is written one line after another** — if a problem seems to demand a loop, you are overcomplicating it.

Two problems ask you to fill in a function body. The `def` line is already in the starter on the platform; you supply the logic. Functions themselves are L3.

Part A — required

2-1 Draw a bear with escape characters

Use `print()` to produce the bear below. It contains backslashes and quotes, and **every one of them has to be escaped correctly**:

```
      (()__(()
      /      \
     ( /    \  \
      \ o o    /
      (_()_)__/ \
     / _,==.____ \
    (   |--|      )
    /\_.|__|'-.__/\_
   / (        /     \
   \  \      (      /
    )  '._____)    /
(((____.--(((____/
```

Escape it right and you get a friendly bear. Escape it wrong and you get a deformed one.

Two routes

One is a `print()` per line, writing `\` as `\\`. The other is a single triple-quoted string (L2's "Multi-line strings"). **Try both** and see for yourself what the second one saves you.

2-2 Change of case (marked by the platform)

Convert an uppercase letter to lowercase. **Use only `ord()` and `chr()`** — `.lower()` is not allowed. Anything that is not an uppercase letter comes back unchanged.

```
to_lower('H')  →  'h'
to_lower('a')  →  'a'
to_lower('7')  →  '7'
```

The body is two or three lines

The starter already has `def to_lower(ch):`; you fill in the rest. Decide whether it is an uppercase letter (comparison operators work on characters: `'A' <= ch <= 'Z'`), and if so add 32 to its `ord()` and `chr()` it back. Otherwise return it unchanged.

2-3 The life of one character

Pick an **English letter** from your own name (say `Z`) and work through this in sequence — no functions, no loops:

1. Get its code with `ord()`
2. Print that code in binary, octal and hex with `bin()` / `oct()` / `hex()`
3. Turn the code back into a character with `chr()` and confirm you get what you started with
4. Add 32 to the code and `chr()` it again. What comes out?

Step 4 is not a coincidence

In ASCII, uppercase and lowercase letters are exactly 32 apart. 32 in binary is `100000` — the sixth bit. So “changing case” is, at the level of bits, **flipping a single bit**. Bitwise operators are L4.

2-4 Area of a triangle

Given sides `a`, `b` and `c`, Heron’s formula is:

```
s = (a + b + c) / 2
area = √(s(s−a)(s−b)(s−c))
```

Take the square root with `** 0.5` (`math.sqrt` needs an `import`, which is L3).

Compute both of these and **print the results exactly as they come, without rounding**:

	a	b	c
first	3	4	5
second	0.3	0.4	0.5

The second triangle's sides are exactly a tenth of the first, so its area should be exactly a hundredth.

Is it? One of the answer-sheet questions asks exactly this. Part B explains what you see.

2-5 Comparing floats safely (marked by the platform)

Implement `almost_equal(a, b, eps=1e-9)`: two floats count as equal when the absolute value of their difference is below `eps`.

One line is enough. `abs()` is built in.

```
almost_equal(0.1 + 0.2, 0.3)   →  True
almost_equal(1.0, 1.5)         →  False
```

Part B uses this function.

2-6 Predict the type

Without running anything, write down the **value and the type** of each expression. Then run them and check. The answer sheet picks one of them to ask about, and asks which ones you got wrong:

```
8 / 4
8 // 4
8.0 // 4
8 % 3
-8 % 3
-8 // 3
2 ** 10
2 ** -2
2 ** 0.5
True + True
```

Predict first — that is the whole point

Running them and copying the output teaches you nothing. These rules have to become automatic, because later, when you are writing loops and indexing lists, an unexpected `float` will crash your program outright.

2-7 Roots of a quadratic (marked by the platform)

Implement `roots(a, b, c)` for `a·x² + b·x + c = 0`, **larger root first**.

You may assume `a > 0` and that the discriminant `b² − 4ac` is positive — exactly two distinct real roots. (The other cases need `if`, which is L4.)

```
roots(1, -3, 2)   →  (2.0, 1.0)
roots(2, 5, -3)   →  (0.5, -3.0)
```

Square root by `** 0.5`, as before.

Returning two values

`return larger, smaller` is all it takes; Python packs them into a tuple. L3 covers this properly — borrow it for now.

Compute the discriminant once

Writing `(-b + (b*b - 4*a*c) ** 0.5) / (2*a)` twice computes the discriminant twice. Put `d = (b*b - 4*a*c) ** 0.5` on its own line and use it in both. **Saving the arithmetic is the minor gain; the line being readable is the real one.**

Part B — going deeper: which digit does floating point lie at?

"Floats are inexact" is an empty phrase until you know **at which digit, and under what circumstances**. Four questions here, each a few lines of straight-line code.

B-1 Look at the actual values

```
print(0.1 + 0.2 == 0.3)
print(0.1 + 0.3 == 0.4)
```

One is `False` and the other is `True`. **Two decimals added in both cases — why the different answers?**

Print all four of these to twenty decimal places with `f"{x:.20f}"` and compare:

```
print(f"{0.1:.20f}")
print(f"{0.3:.20f}")
print(f"{0.1 + 0.3:.20f}")
print(f"{0.4:.20f}")
```

On the answer sheet: is the stored `0.1` above or below the real 0.1? Work out which way `0.3` leans too — the next question needs it.

B-2 Find three examples of your own

`0.1 + 0.2 == 0.3` is `False`. Find **three more** pairs that are equal in arithmetic but compare `False`, printing both sides of each at `.20f`.

Do not reuse the one you were given.

B-3 Large numbers swallow small ones

Try these two lines:

```
print(1e15 + 1 == 1e15)
print(1e16 + 1 == 1e16)
```

One is `False`, the other `True`. Past a certain magnitude, **adding 1 has no effect whatsoever**.

Answer:

1. From which power of ten does `+ 1` stop working?
2. At that magnitude, does `+ 2` still work? What about `+ 4`? Try it and record what you find.
3. Explain why, in one sentence. (Hint: a float has a fixed number of significant bits.)

> **This is not a toy problem**
>
> In 2016 a reporting program summed the amounts of hundreds of millions of records and came out tens of thousands short of a record-by-record count. This was why: by the end of the sum the running total was large enough that each small amount added to it was simply absorbed. **The remedies are to sort first and add from smallest up, or to store amounts as whole cents.**

B-4 Back to the question from Lab 1

In Lab 1's B-3 you replaced

```
x = 3.9 * x * (1 - x)
```

with the algebraically identical

```
x = 3.9 * x - 3.9 * x * x
```

Same starting value, same formula, different results. You were asked to guess then. Now you can answer.

1. At which round do the two versions first differ? (Run it and compare both values at twenty decimals.)
2. Why do they differ? Explain using B-1 to B-3.
3. So **which version is "right"**?

> **Question 3 has no model answer**
>
> Neither is right — neither computes the arithmetic expression, they are just wrong in different ways. That is exactly why numerical analysis exists as a subject: one formula has many algebraically equivalent spellings, and under floating point their errors can differ by orders of magnitude.
>
> Look back at 2-4 while you are here. The `0.3, 0.4, 0.5` triangle not coming out at exactly 0.06 is the same phenomenon. Heron's formula is disastrously inaccurate for very thin triangles, and a numerical analysis course will hand you a rewritten version that is not.

Part C — fundamentals: files and directories

Last week's `pwd`, `ls` and `cd` let you look and move. This week: **create, copy, rename, delete**.

Command	What it does	How to remember it
`mkdir NAME`	create a directory	**mk**e **dir**ectory
`touch FILE`	create an empty file	on an existing file it just updates the timestamp
`cp SRC DST`	copy	**cop**y
`mv SRC DST`	move — **and this is also how you rename**	**m**ove
`rm FILE`	delete	re**m**ove
`cat FILE`	print the contents	con**cat**enate

> **`rm` has no recycle bin**
>
> What you delete from the command line **does not go to the bin and cannot be recovered**. Run `ls` first and be sure. Above all, do not type `rm` when you are unsure which directory you are in — check with `pwd`.
>
> And never, ever type `rm -rf /`. That command erases the entire system.

Seeing the files you normally cannot

`ls` hides anything whose name starts with `.` — that is the convention for "hidden" files, and configuration files almost all look like that (`.gitignore`, `.bashrc`, macOS's `.DS_Store`).

```
ls -a       # show everything, dotfiles included
ls -l       # one per line, with size, time and permissions
ls -al      # both at once
```

`ls -a` turns up two extra entries, `.` and `..`. They are **not ordinary files** but two entries every directory carries:

	Points at
`.`	the directory itself
`..`	its parent

That is what last week's `cd ..` was using, and the trailing `.` in `cp ../hello.py .` too.

> **The permission column is L11**
>
> The `drwxr-xr-x` at the left of `ls -l` is the permission bits; you do not need to read them yet. Knowing that a leading `d` means a **d**irectory is enough for now.

Absolute and relative paths

- An **absolute path** is written out from the root: `/home/yourname/cs1602/hello.py` (on Windows, `C:\cs1602\hello.py`)
- A **relative path** starts from where you are: `hello.py`, `../hello.py`, `code/hello.py`

`.` is the current directory, `..` is the parent, `~` is your home directory.

C-1 Walk through it

Do these in order and copy the commands and their output onto the answer sheet:

1. `cd` into your cs1602 folder
2. `mkdir week2`, then `ls` to confirm it appeared
3. `cd week2`, then `touch note.txt` for an empty file
4. `cp ../hello.py .` — copy last week's `hello.py` here (mind that trailing `.`)
5. `mv note.txt readme.txt` — rename it, and `ls` to confirm
6. `cat hello.py` — look at the contents

7. `ls -a` — did `.` and `..` show up? Any other hidden files?
8. `cd ..`, then `pwd` to confirm you are back up a level

C-2 Think it through

No typing required:

1. After `cp a.txt b.txt`, is `a.txt` still there? And after `mv a.txt b.txt`?
2. You are in `~/cs1602/week2` and want to reach `~/cs1602/hello.py`. What is the relative path?

Submitting

Submit on the grading platform, inside Homework 1. 2-2, 2-5 and 2-7 are code problems, marked the moment you submit. Everything else — Part A's observations and all of Parts B and C — goes on the three **answer sheets**, where the multiple choice is marked immediately and your written answers go into the grading report for a TA.

> **Return, don't print**
>
> The platform calls your function and compares the value it gives back. This week's problems are straight-line code, so printing is fine; from L3 onward, when you start writing functions, `return` is the rule.