Lab 09 — The Petri Dish

Guess and check, close enough, bisection; exceptions and assertions

DURATION ~3 hours **COVERS** Lectures 15 & 16 **TASKS** 7 files **CHECKER** `check.py`

How this lab works

- Every section opens with a worked example. It is a file in your folder, `example1.py` to `example5.py`: open it in Thonny, run it, and see the run printed on the sheet.
- Work top to bottom. Read a task, then its five keys. Four are practical: type a line in Thonny's Shell, or change one thing in the example file and run it again, and look at what it shows. The fifth is worked out by hand, Shell closed, with the small numbers it gives.
- Type the five answers into `check.py` when it asks. All five right, and the task's file opens. A wrong answer costs nothing: type the line again and look.
- Tasks are the deliverable: `task01.py` ... `task07.py` in your `lab09` folder. Aim to pass them all. A failed check costs nothing: read its reason and try again.
- Run `check.py` as often as you like.
- Ask. Talk to your neighbour, the TA or the instructor about the idea. Type your own code: no copy-paste.
- The minutes on a task are a guide for pacing, not a limit.
- This lab assumes Lab 08 and the labs before it: `for` and `while` loops, a list built with `append`, reading a line with `split()` and `int()`, and a function that returns its answer.

Before you start: get the files

Download the `lab09` folder (from the course LMS page) onto your machine. It contains `task01.py` ... `task07.py`, the worked examples `example1.py` ... `example5.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. The example files are yours to change; nothing in them is handed in.

The story of this lab

In 1928 Alexander Fleming came back from a holiday. Mould had grown on one of his round dishes of germs. Around the mould, the germs had died. That mould gave us penicillin, the first antibiotic. Today a hospital lab still grows germs in those dishes, counts them, and finds the dose of medicine that stops them. All the code in this lab is about that work.

1 Try Every Answer (~23 min)

Ref: Lecture 15 — Searching, Approximation Methods

Sometimes you can check an answer but cannot work it out. Then try every possible answer, in

order, and check each one. A flag, a name that holds `True` or `False`, remembers if one passed. The search must stop somewhere, and its last possible answer must be tried too.

Worked example: when does the dish hold this many?

Germs split in two every hour. One cell becomes 2, then 4, then 8: after h hours there are `2 ** h` cells. This program reads a number of cells and tries hours 0 to 12. It is `example1.py` in your folder: open it, run it, type 64, and see this run.

```
cells = int(input())
found = False
for hour in range(13):
    if 2 ** hour == cells:
        found = True
        break
if found:
    print("Hour:", hour)
else:
    print("Never")
```

Input

```
64
```

Output

```
Hour: 6
```

Notice: `found` is set to `False` above the loop, and only a guess that passes makes it `True`. After the loop, the flag decides what to print. `range(13)` gives 0 to 12, so hour 12 is tried too. Key 1.2 changes the 13.

Three things check.py is strict about

- Read each line with `input()` and nothing inside the brackets, as in the CCC problems. A prompt such as `input("s: ")` is printed too, so the task fails.
- A key's answer is what the Shell showed, typed as it showed it: a number, `True`, `False`, an error's name such as `ValueError`, or the one word a `print` showed. No units, no quotes. A by-hand key's answer has the same shape.
- In Task 7, `check.py` also calls your function by itself, as you can in the Shell after F5. Some of its calls must stop with an error. So keep the exact name and parameters the task gives, hand the answer back with `return`, and print nothing inside the function.

TASK 1 On the Hour

EASY `task01.py` ~11 min

A dish does not always start with one cell. A dish that starts with s cells holds `s * 2 ** h` cells after h hours. The lab watches the dish for one day: from hour 0, the start, to hour 24.

Your task: Read s and n. Try every hour from 0 to 24. Print the hour at which the dish holds exactly n cells. If no hour does, print `Not today`.

Input: Line 1: s, the cells at the start, a whole number, 1 or more. Line 2: n, the cells to look for, a whole number, 1 or more.

Output: `Hour: h`, the hour from 0 to 24; or `Not today` when no hour gives exactly n.

Sample 1 (the checker uses this one)

Input

```
3
96
```

Output

```
Hour: 5
```

Why: The dish holds 3, 6, 12, 24, 48 and then 96 cells, at hours 0 to 5. So 96 is reached at hour 5.

Hint: start from the worked example. Change the test to use `s`, and change the `range` so that hour 24 is tried too: Key 1.4 shows why `range(24)` is not enough.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task01.py` opens.

Key 1.1 ⌨ Run `example1.py` and type `4096`. What number is on the `Hour:` line?

Answer: .....

Key 1.2 ⌨ Now change `range(13)` to `range(12)` and run it again on `4096`. What is printed now?

Answer: .....

Key 1.3 ⌨ Type `5 * 2 ** 3`. What came back?

Answer: .....

Key 1.4 ⌨ Type `24 in range(24)`. What came back?

Answer: .....

Key 1.5 ✎ **By hand, Shell closed.** A dish starts with 5 cells. At which hour does it hold exactly 40?

Answer: .....

TASK 2 Two Pipettes

EASY `task02.py` ~12 min

⌨ A pipette drops the same amount of liquid at every squeeze. Pipette A gives a microlitres (µL) a squeeze, and pipette B gives b µL. To measure exactly t µL, squeeze A i times and B j times, so that `a * i + b * j == t`. `task02.py` tries every pair, one loop inside the other. It prints each pair that works, then the number of ways. It runs, but it prints the wrong things.

Your task: It has exactly **two** bugs. Find and fix them. Do not rewrite the program.

Input: Line 1: a. Line 2: b. Line 3: t. Each is a whole number of µL, 1 or more.

Output: one line per way, `A: i B: j`, with i from small to large; then `Ways: W`, how many ways there are.

Sample 1 (the checker uses this one)

Input

```
1
4
8
```

Output

```
A: 0 B: 2
A: 4 B: 1
A: 8 B: 0
Ways: 3
```

Why: 0 + 8, 4 + 4 and 8 + 0 all make 8. The last way squeezes A 8 times, so i must reach 8.

Hint: find the bugs by **bisecting the program**. Print `i` and `ways` at the end of each pass of the outer loop (Key 2.3), and see where the count goes wrong. A counter is set once, above both loops. Key 2.2 is the fact behind the other bug.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task02.py` opens.

Key 2.1 Open `task02.py`, run it as it is and type `1`, `4`, `8`. What number is on the `Ways:` line?

Answer: .....

Key 2.2 Type `max(range(8))`. What came back?

Answer: .....

Key 2.3 Put `print(i, ways)` as the last line of the outer loop's body: four spaces in, under `for j`. Run it on the sample. What number ends the line that starts with `5`?

Answer: .....

Key 2.4 Type `1 * 8 + 4 * 0 == 8`. What came back?

Answer: .....

Key 2.5 **By hand, Shell closed.** Pipette A gives 2 µL and pipette B gives 3 µL. How many ways make exactly 6 µL?

Answer: .....

2 Close Enough (~11 min)

Ref: Lecture 15 — Searching, Approximation Methods

A float is often a tiny bit off: `0.1 + 0.2` is not exactly `0.3`. So never test floats with `==`. Test whether they are close enough: within a small number called epsilon.

Worked example: the balance

A balance adds three portions of 0.1 g of powder. Is the total 0.3 g? This program asks two ways. It is `example2.py`: run it; nothing is typed.

```
1 total = 0.0
2 for i in range(3):
3     total = total + 0.1
4 print("Total:", total)
5 print("Equal:", total == 0.3)
6 epsilon = 0.0001
7 print("Close:", abs(total - 0.3) <= epsilon)
```

Output

```
Total: 0.30000000000000004
Equal: False
Close: True
```

Notice: the total is a tiny bit more than 0.3, so `==` says `False`. `abs(total - 0.3)` is how far apart the two are, and the last line asks if that is at most epsilon. Key 3.2 changes the epsilon.

TASK 3 Drop by Drop

EASY `task03.py` ~11 min

⌨ A pipette adds drops of d mL to a flask, one at a time, until the flask holds v mL. `task03.py` counts the drops. It holds the whole program but for one gap, marked `# TODO`: the test that stops the loop. As it is, the loop never stops.

Your task: Fill the gap with two lines: if `total` is within `epsilon` of `v`, `break`. Change nothing else.

Input: Line 1: v, the mL the flask must hold, 0 or more. Line 2: d, the mL in one drop, more than 0. Both are floats, and d goes into v exactly.

Output: `Drops: k`, how many drops were added.

Sample 1 (the checker uses this one)

Input

```
1.0
0.1
```

Output

```
Drops: 10
```

Why: Ten drops of 0.1 make 1.0. On the computer they add up to 0.9999999999999999, which is within 0.0001 of 1.0, so the loop stops at 10.

Hint: the test is the last line of the worked example, with `total` and `v`. With `total == v` the loop never stops (Key 3.3 shows why); with `total >= v` it adds an eleventh drop. A loop that never stops is ended with the red Stop button, or Ctrl + c in the Shell.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task03.py` opens.

Key 3.1 ⌨ Run `example2.py`. What is the last digit on the `Total:` line?

Answer: .....

Key 3.2 ⌨ Change `epsilon = 0.0001` to `epsilon = 0` and run it again. What is on the `Close:` line now?

Answer: .....

Key 3.3 ⌨ Type `0.1 + 0.2 == 0.3`. What came back?

Answer: .....

Key 3.4 ⌨ Type `abs(0.3 - 0.1 * 3) <= 0.0001`. What came back?

Answer: .....

Key 3.5 ✎ **By hand, Shell closed.** Drops of 0.5 mL go into a flask until it holds 3.0 mL. How many drops is that?

Answer: .....

3 Halve the Range (~16 min)

Ref: Lecture 15 — Searching, Approximation Methods

Bisection search guesses the middle of a range. Each answer says which half to keep, so each guess throws away half of what is left. `low` and `high` start one step outside the range, and the answer is always between them.

Worked example: find the box

A freezer holds sample boxes numbered 1 to 100, in order. To find box n the program guesses the middle, then keeps the half that holds n. It is `example3.py`: run it and type 70.

```
n = int(input())
low = 0
high = 101
guess = (low + high) // 2
while guess != n:
    print("Try", guess)
    if guess < n:
        low = guess
    else:
        high = guess
    guess = (low + high) // 2
print("Found box", guess)
```

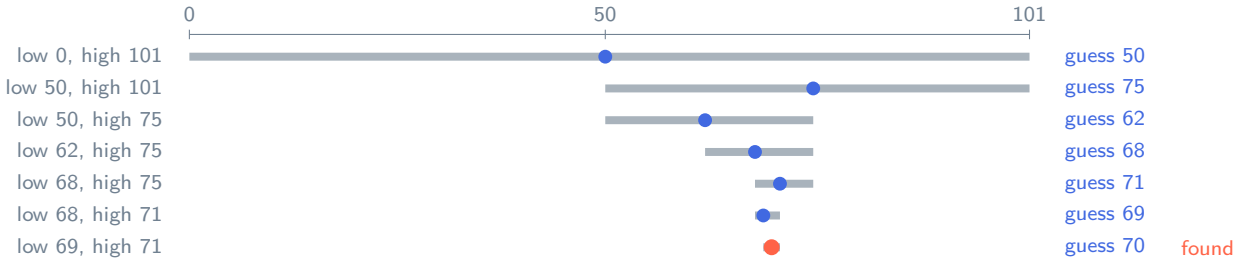
Input

```
70
```

Output

```
Try 50
Try 75
Try 62
Try 68
Try 71
Try 69
Found box 70
```

Notice: `low` and `high` start at 0 and 101, one step outside the boxes 1 to 100. A guess too small moves `low` up to it; a guess too big moves `high` down. Key 4.2 changes the 101.



TASK 4 Smallest Dose

MEDIUM `task04.py` ~16 min

An antibiotic stops a germ from growing once the dose is big enough. The lab wants the **smallest** whole dose, in mg, that stops it. Every dose of d mg or more stops the germ, and every smaller dose fails. A dose of n mg always stops it. Each test takes a day, so the lab uses a bisection search. Here `check.py` types d too, to play the germ: the program must still find d by testing doses.

Your task: Find the smallest dose with a bisection search:

Step 1: Read n and d. Set `low = 0` and `high = n + 1`, one step outside the doses 1 to n, and `tests = 0`. While `high - low > 1`: set `dose = (low + high) // 2`, add 1 to `tests`, and print `Dose` and the dose. Then, if `dose >= d` (it stops the germ), move `high` down to `dose`; if not, move `low` up to `dose`. Run it on the sample: you should see six `Dose` lines, 32, 16, 8, 12, 14 and 13.

Step 2: Move the print into the two branches, and end each line with `stops it` or `fails`. Run it again.

Step 3: After the loop, print `Smallest dose:` and `high`, then `Tests:` and `tests`. Run it: you should see the sample.

Input: Line 1: n, the largest dose in mg, a whole number, 1 or more. Line 2: d, the smallest dose that stops the germ, from 1 to n.
Output: one line per test, `Dose x stops it` or `Dose x fails`; then `Smallest dose: m`, and `Tests: k`.

Sample 1 (the checker uses this one)

Input

```
64
13
```

Output

```
Dose 32 stops it
Dose 16 stops it
Dose 8 fails
Dose 12 fails
Dose 14 stops it
Dose 13 stops it
Smallest dose: 13
Tests: 6
```

Why: 13 mg or more stops the germ. After 13 stops it and 12 fails, low is 12 and high is 13: no dose is left between them, so 13 is the answer.

Hint: the loop ends when `low` and `high` are next to each other (Keys 4.3 and 4.4). Then `high` is the smallest dose that stopped the germ, and `low` the largest that failed.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task04.py` opens.

Key 4.1 ⌨ Run `example3.py` and type `70`. How many `Try` lines are printed?
Answer: .....

Key 4.2 ⌨ Change `high = 101` to `high = 1001`, for boxes 1 to 1000, and run it again on `70`. How many `Try` lines are printed now?
Answer: .....

Key 4.3 ⌨ Type `(12 + 13) // 2`. What came back?
Answer: .....

Key 4.4 ⌨ Type `13 - 12 > 1`. What came back?
Answer: .....

Key 4.5 ✎ **By hand, Shell closed.** n is 8 and d is 3, so `low` starts at 0 and `high` at 9. How many tests does the search make?
Answer: .....

4 Catch the Error (~23 min)

Ref: Lecture 16 — Assertions & Exceptions

A bad value stops a program with an error, called an exception. Put the risky line inside `try`. If it fails, the `except` part runs and the program goes on. Name the error after `except`, so that other mistakes still show. An `else` part runs only when nothing failed.

Worked example: the plate reader's lines

A machine counts the colonies, the spots of germs, on each dish. It sends one line per dish, but sometimes the line is `ERR`. This program skips the bad lines and counts the good ones. It is `example4.py`: run it and type the five lines.

```
n = int(input())
good = 0
for i in range(n):
    line = input()
    try:
        count = int(line)
    except ValueError:
        print("Skipped:", line)
    else:
        good = good + 1
print("Good lines:", good)
```

Input

```
4
120
ERR
85
7.5
```

Output

```
Skipped: ERR
Skipped: 7.5
Good lines: 2
```

Notice: `int(line)` is the risky line, so it sits inside `try`. `int("7.5")` fails too: 7.5 is not a whole number. The `else` part counts a line only when `int` worked. Key 5.2 changes the error's name, and Key 6.1 changes the `else`.

TASK 5 Plate Reader

EASY `task05.py` ~11 min

The same machine. A good line is a whole number of colonies. A bad line is anything else: `ERR`, a blank line, or a number with a decimal point such as `12.5`. The lab wants to know how many lines were bad, and the biggest colony count.

Your task: Read the lines. Skip every bad line and count how many there were. Print that count, then the largest good count. The program must not crash on a bad line.

Input: Line 1: n, the number of lines that follow, 1 or more. Then n lines, one per dish, good or bad; at least one is good.

Output: `Bad lines: b`, how many lines were not whole numbers; then `Largest: m`, the largest whole number read.

Sample 1 (the checker uses this one)

Input

```
5
130
ERR
240
12.5
85
```

Output

```
Bad lines: 2
Largest: 240
```

Why: ERR and 12.5 are not whole numbers, so 2 lines are bad. Of 130, 240 and 85, the largest is 240.

Hint: keep the good counts in a list. Append to it inside the `try`, so only a line that `int` accepted gets in. Add 1 to the bad count in the `except` part. `max` of the list is the last line.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task05.py` opens.

Key 5.1 Run `example4.py` and type `4`, `120`, `ERR`, `85`, `7.5`. What number is on the `Good lines:` line?

Answer: .....

Key 5.2 Change `except ValueError:` to `except ZeroDivisionError:` and run it again on the same five lines. The last line of the error starts with an error's name. Which?

Answer: .....

Key 5.3 Type `int("12.5")`. The last line starts with an error's name. Which?

Answer: .....

Key 5.4 Type `max([130, 85, 12])`. What came back?

Answer: .....

Key 5.5 **By hand, Shell closed.** The machine sends four lines: `40`, `ERR`, `65`, `1.5`. How many of them are bad lines?

Answer: .....

TASK 6 Growth Factor

EASY `task06.py` ~12 min

Now the machine counts each dish twice: in the morning, and one day later. It sends both counts on one line, such as `120 960`. The growth is the later count `//` the morning count: how many times bigger the colony got. A line can be bad in two ways. It may not hold two whole numbers. Or the morning count may be 0, and nothing can be divided by 0.

Your task: Start from your `task05.py`: copy it into this file, then change it. Keep the loop and the `try`. For each line, print its growth, or `Bad line` on a `ValueError`, or `No cells at start` on a `ZeroDivisionError`. Use two `except` parts, one for each error, and print the growth in an `else` part. Drop the two lines Task 5 printed at the end.

Input: Line 1: n, the number of lines that follow, 1 or more. Then n lines, one per dish.
Output: one line per dish: `Growth: g`, or `Bad line`, or `No cells at start`.

Sample 1 (the checker uses this one)

Input

```
4
120 960
0 50
ERR
30 90
```

Output

```
Growth: 8
No cells at start
Bad line
Growth: 3
```

Why: 960 // 120 is 8 and 90 // 30 is 3. 50 // 0 cannot be done. ERR is not two numbers.

Hint: split the line into two names inside the `try`: `start, end = line.split()`. A line with one piece makes that line fail with a `ValueError` (Key 6.3), and so does a piece that `int` cannot read. Then do the division inside the `try` too.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task06.py` opens.

Key 6.1 Put `except ValueError:` back in `example4.py` if you changed it. Then change `else:` to `finally:` and run it on the same five lines. What number is on the `Good lines:` line now?

Answer: .....

Key 6.2 Type `960 // 0`. The last line starts with an error's name. Which?

Answer: .....

Key 6.3 Type `a, b = "ERR".split()`. The last line starts with an error's name. Which?

Answer: .....

Key 6.4 Type `a, b = "30 120".split()`, then `int(b) // int(a)`. What came back?

Answer: .....

Key 6.5 **By hand, Shell closed.** The machine sends five lines: `40 200`, `0 7`, `12`, `x y`, `20 20`. How many of them print `Bad line`?

Answer: .....

5 Raise It, Assert It (~15 min)

Ref: Lecture 16 — Assertions & Exceptions

A function can stop with an error on purpose: `raise ValueError("why")`. Its caller cannot miss that, as it could miss a `print`, and can catch it with `try`. An `assert` line claims that something is true. If the claim is false, the program stops with an `AssertionError`.

Worked example: a dilution

To dilute a sample, the lab mixes it with water. A dilution of 10 makes it 10 times weaker, so it holds a tenth of the cells. This function refuses a factor under 1, and claims the cells are never negative. It is `example5.py`: run it; nothing is typed.

```
1  def dilute(cells, factor):
2      assert cells >= 0, "cells cannot be negative"
3      if factor < 1:
4          raise ValueError("factor must be 1 or more")
5      return cells // factor
6
7  print(dilute(8000, 10))
8  try:
9      print(dilute(8000, 0))
10 except ValueError:
11     print("Rejected")
```

Output

```
800
Rejected
```

Notice: `dilute(8000, 0)` raises a `ValueError`, so nothing comes back to print, and the caller's `except` prints `Rejected`. The `assert` guards against a mistake in the code, not in the data. Keys 7.1 and 7.2 break each of them.

TASK 7 Countable Plates

MEDIUM task07.py ~15 min

⌨ To count the cells in a sample, the lab dilutes it, spreads it on a dish and counts the colonies. Cells in the sample are colonies times the dilution. A dish can be counted only with 30 to 300 colonies, 30 and 300 included. Fewer are too few to trust, and more are too many to count.

Part 1, the function: `plate_count(colonies, dilution)` returns `colonies * dilution`. For colonies under 30 or over 300 it raises a `ValueError`: a problem in the data. Its first line is `assert dilution >= 1, "..."`: a dilution under 1 is a mistake in the code that calls it. It prints nothing. In the Shell, `plate_count(150, 1000)` gives `150000`, and `plate_count(12, 100)` stops with a `ValueError`.

Part 2, the program: Read the dishes. For each one, print `Cells:` and what `plate_count` returns, or `Recount` when it raises:

Step 1: Write the function with its `return` and the `raise`. Run the file, then type `plate_count(150, 1000)` and `plate_count(12, 100)` in the Shell: you should see 150000, then the error.

Step 2: Add the `assert` as the function's first line. Run the file, then type `plate_count(100, 0)`: you should see `AssertionError`.

Step 3: Below the function, read the dishes. Call the function inside `try`, and print `Recount` in `except ValueError`. Run it: you should see the sample.

Input: Line 1: n, the number of dishes, 1 or more. Then n lines: the colonies and the dilution, two whole numbers with a space between. The dilution is always 1 or more.

Output: one line per dish: `Cells: c`, or `Recount`.

Sample 1 (the checker uses this one)

Input

```
4
150 1000
12 100
300 10
450 10
```

Output

```
Cells: 150000
Recount
Cells: 3000
Recount
```

Why: 150 and 300 are countable: 150 times 1000, and 300 times 10. 12 is too few and 450 too many, so both dishes are recounted.

Hint: read a dish's line as in Lab 08: `colonies, dilution = [int(x) for x in input().split()]` (Key 7.4). Put `print("Cells:", plate_count(colonies, dilution))` inside the `try`: when the call raises, nothing is printed but `Recount`.

Keys with a keyboard: type the line in Thonny's Shell, or make the change in the example file, and look. The key with a pen: work it out by hand, Shell closed. Type the five answers into `check.py`, and `task07.py` opens.

Key 7.1 ⌨ Run `example5.py`. Then change `dilute(8000, 10)` to `dilute(-8000, 10)` and run it again. The last line of the error starts with an error's name. Which?

Answer: .....

Key 7.2 ⌨ Change it back to `dilute(8000, 10)`. Then change `factor < 1` to `factor < 0` and run it again. Which error's name is on the last line now?

Answer: .....

Key 7.3 ⌨ Type `30 <= 30 <= 300`. What came back?

Answer: .....

Key 7.4 Type `c, d = [int(x) for x in "45 100".split()]`, then `c * d`. What came back?

Answer: .....

Key 7.5 **By hand, Shell closed.** What does `plate_count(25, 1000)` do? Type the number it returns, or the name of the error it stops with.

Answer: .....

Self-Assessment Checklist

Before you leave, tick what you can now do:

- ☐ I can try every possible answer in order, and check each one.
- ☐ I can make sure a search tries its last value too: `range(25)` for hours 0 to 24.
- ☐ I can set a flag above a loop, make it `True` when a guess passes, and use it after the loop.
- ☐ I can try every pair with one loop inside another, and keep a counter set once, above both loops.
- ☐ I can bisect a program: print a value in the middle and see which half is wrong.
- ☐ I know why `0.1 + 0.2 == 0.3` is `False`, and I test floats with `abs(a - b) <= epsilon`.
- ☐ I can run a bisection search: guess the middle, and move `low` or `high` to the guess.
- ☐ I can put a risky line inside `try`, and catch the error it names with `except`.
- ☐ I can catch two different errors with two `except` parts, and I know when `else` and `finally` run.
- ☐ I can raise a `ValueError` from a function, and catch it where the function is called.
- ☐ I can write an `assert` line, and I know it stops the program with an `AssertionError`.