Python Advanced Practice Worksheet

How to use this worksheet:

- 1. Read the grey **Example** box. Look at the code AND the output.
- 2. Type the example into Python and run it. Check you get the same output.
- 3. Do the **Try it** tasks underneath. They are almost the same as the example - just change one or two words.
- 4. If you get stuck: check spelling, check " quotes, check ( ) brackets, check : colons and spaces at the start of lines.
- 5. Tick off each task when it works.

This worksheet follows on from the Beginners worksheet (Sections 1-13). It assumes you can already do print, variables, input, int, if/elif/else, strings, for loops, turtle drawings and while loops.

Golden rules for this worksheet:

- List positions start at 0, not 1. First item is my_list[0], last item is my_list[-1].
- Dict keys need quotes: student["name"], not student[name].
- After def, if, elif, else, for, while, always put : and indent (4 spaces) on the next line.
- A function that only prints gives back None. A function that returns gives back an answer you can store.
- Files must be loaded before you read them. If the file is missing you get FileNotFoundError.
- Decide the answer BEFORE you run a test. assert passes silently and shouts on failure.

---

14. Lists - Keeping things in order

A list keeps many items in order. Positions start at 0. Negative numbers count from the end.

Code:

```
scores = [12, 7, 19, 4]
print(scores[0])
print(scores[-1])
print(len(scores))
scores.append(15)
print(scores)
```

Output:

```
12
4
4
[12, 7, 19, 4, 15]
```

Example 14B - Loop, total, average

Code:

```
scores = [12, 7, 19, 4, 15]
total = 0
for s in scores:
    total = total + s
print("Total:", total)
print("Average:", total / len(scores))
```

Output:

```
Total: 57
Average: 11.4
```

Tip: scores[0] is the first item. scores[-1] is the last item. len() is how many items. append() adds to the end.

Try it - Section 14

- 14.1 Make scores = [12, 7, 19, 4]. Print the first item with scores[0].
- 14.2 Print the last item with scores[-1]. Print how many with len(scores).
- 14.3 Add 15 with scores.append(15). Print the list - where did it go?
- 14.4 Loop over the list to work out the total (copy Example 14B).
- 14.5 Print the average: total divided by len(scores). You should see 11.4.
- 14.6 Ask for a 5th score with int(input(...)) and append it. Print the new average.
- 14.7 PREDICT: What does scores[4] give AFTER the append? What about BEFORE? Run it.
- 14.8 FIX IT: This crashes with IndexError. Why? Fix it.

Code:

```
scores = [12, 7, 19, 4]
print(scores[10])
```

Challenge 14 - Favourites List

Make a list of 4 favourite games. Print each one with a loop, then print how many. Then append one more game and print the list again.

---

15. Dicts - Labelled boxes and mini datasets

A dict labels each value, so order does not matter. A list of dicts is a mini dataset - one dict per row.

Example 15A - One dict

Code:

```
student = {"name": "Aisha", "age": 16, "city": "Dublin"}
print(student["name"])
print(student["age"])
student["age"] = 17
print(student)
```

Output:

```
Aisha
16
{'name': 'Aisha', 'age': 17, 'city': 'Dublin'}
```

Example 15B - List of dicts (mini dataset)

Code:

```
class_list = [
    {"name": "Aisha", "mark": 72},
    {"name": "Ben", "mark": 45},
    {"name": "Cara", "mark": 81},
]
for s in class_list:
    print(s["name"], s["mark"])
```

Output:

```
Aisha 72
Ben 45
Cara 81
```

Tip: Dict keys need quotes. student["name"] works. student[name] crashes with NameError because Python thinks name is a variable.

Try it - Section 15

- 15.1 Make the student dict from Example 15A. Print the name and age.
- 15.2 Change the age to 17 and print the whole dict.
- 15.3 Make the 3-row class_list. Loop and print each name + mark.
- 15.4 Add a 4th row (yourself) with .append(). Print the list.
- 15.5 Print len(class_list). How many rows now?
- 15.6 PREDICT: What does student["city"] give? What about student[0]? Try both.
- 15.7 FIX IT:

Code:

```
student = {"name": "Aisha", "age": 16}
print(student[name])
```

Challenge 15 - Grade Field

Add a grade field to each row in class_list: Gold for 70+, else Silver. Print name + grade. Use a loop and an if/else inside it.

---

16. Functions - Define once, call often

Functions bundle steps under a name. Define once, call often. print() shows it, return gives it back.

Example 16A - Define and call

Code:

```
def greet(name):
    print("Hello", name)

greet("Aisha")
greet("Ben")
```

Output:

```
Hello Aisha
Hello Ben
```

Example 16B - return sends an answer back

Code:

```
def add(a, b):
    return a + b

print(add(3, 4))
total = add(10, 20)
print(total)
```

Output:

```
7
30
```

Note: If a function only prints and has no return, storing the result gives None. Try x = greet("Ben") then print(x) - you will see None.

Try it - Section 16

- 16.1 Copy greet and call it with two different names.
- 16.2 Copy add. Print add(3, 4), then store add(10, 20) in total and print it.
- 16.3 Write double(n) that returns n * 2. Print double(7). You should see 14.
- 16.4 Write area(side) that returns side * side. Print area(5). You should see 25.
- 16.5 Write is_even(n) that returns True if n % 2 == 0 else False. Test 4 and 7.
- 16.6 PREDICT: What prints here? Run it to check.

Code:

```
def greet(name):
    print("Hello", name)

x = greet("Ben")
print(x)
```

- 16.7 FIX IT: This returns nothing. Fix it with return.

Code:

```
def add(a, b):
    total = a + b

print(add(3, 4))
```

Challenge 16 - Grade Function

Write grade(mark) returning Gold / Silver / Bronze (like Section 07: 70+, 40+, else). Test 85, 50, 20. Then test the boundaries 70 and 40.

---

17. Files + CSV - Load, clean, sort data

Real data lives in files. A CSV file is text with commas. In PyTeach you load it with the + CSV / text button under the editor. Files stay in YOUR browser only - nothing is uploaded.

data.csv looks like this:

Output:

```
name,mark
Aisha,72
Ben,45
Cara,81
Dan,64
```

Example 17A - Read the file

Code:

```
import csv
with open("data.csv") as f:
    rows = list(csv.DictReader(f))
print(rows[0])
```

Output:

```
{'name': 'Aisha', 'mark': '72'}
```

Note: Marks come in as TEXT ('72', with quotes). You must convert with int() before doing maths.

Example 17B - Clean and sort (highest first)

Code:

```
import csv
with open("data.csv") as f:
    rows = list(csv.DictReader(f))

clean = []
for r in rows:
    name = r["name"].strip()
    mark = r["mark"].strip()
    if name == "" or mark == "":
        continue
    clean.append({"name": name, "mark": int(mark)})

clean.sort(key=lambda r: r["mark"], reverse=True)
for r in clean:
    print(r["name"], r["mark"])
```

Output:

```
Cara 81
Aisha 72
Dan 64
Ben 45
```

Try it - Section 17

- 17.1 Make a data.csv (name,mark + 4 rows) and load it with + CSV / text.
- 17.2 Read it with csv.DictReader and print the first row.
- 17.3 Print every row's name and mark with a loop.
- 17.4 Add the cleaning step: strip names, skip blanks, int() the marks.
- 17.5 Sort highest-first and print the leaderboard (copy Example 17B).
- 17.6 PREDICT: What happens if you run BEFORE loading the file? Run it and read the error.
- 17.7 FIX IT: A mark of absent crashes int(). How would you skip that row? Hint: check mark.isdigit() or use try.
- 17.8 FIX IT: This crashes. Why do marks need int() first?

Code:

```
first = rows[0]["mark"]
print(first + 10)
```

Challenge 17 - Pass List

Print only rows with mark >= 50, plus a count of how many passed. Example output ends with Passed: 3.

---

18. Statistics - Mean, median, mode + text charts

The statistics module is built in - no install needed. PyTeach has no plotting, so draw bars with text (*).

Example 18A - The three averages

Code:

```
import statistics
marks = [72, 45, 81, 64, 58, 90, 45]
print("Mean:", round(statistics.mean(marks), 2))
print("Median:", statistics.median(marks))
print("Mode:", statistics.mode(marks))
```

Output:

```
Mean: 65
Median: 64
Mode: 45
```

Tip: Mean = total divided by how many. Median = middle value when sorted. Mode = most common value.

Example 18B - Text bar chart

Code:

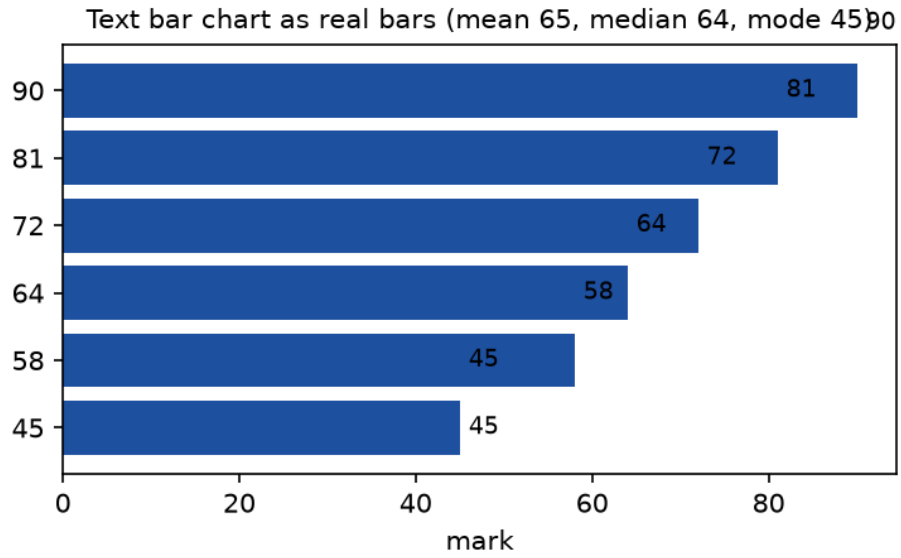
```
marks = [72, 45, 81, 64, 58, 90, 45]
for m in sorted(marks):
    print(str(m) + " " + "*" * (m // 5))
```

Output:

```
45 *********
45 *********
58 ***********
64 ************
72 **************
81 ****************
90 ******************
```

Same data as real bars:

Bar chart of the marks



Try it - Section 18

- 18.1 Copy Example 18A. Run it and read off mean, median, mode.
- 18.2 Add your own mark to the list. Which of the three moves most?
- 18.3 Draw the text bar chart. What does m // 5 control? Try m // 10.
- 18.4 Work out the mean of [10, 20, 30] by hand, then check with Python.
- 18.5 PREDICT: Median of [1, 2, 100] vs mean of [1, 2, 100] - which is bigger?
- 18.6 FIX IT: This crashes. Guard it with an if.

Code:

```
import statistics
print(statistics.mean([]))
```

Challenge 18 - CSV Stats

From your CSV leaderboard (Section 17), print mean / median / mode of the marks plus a text bar chart.

---

19. Text analytics - Count and clean words

Count with a dict: word maps to how many times seen. Real text is messy - clean it first with .lower() and .strip().

Example 19A - Counter

Code:

```
text = "red blue red green blue red"
counts = {}
for word in text.split():
    if word in counts:
        counts[word] = counts[word] + 1
    else:
        counts[word] = 1
print(counts)
```

Output:

```
{'red': 3, 'blue': 2, 'green': 1}
```

Example 19B - Most common + cleaning

Code:

```
counts = {'red': 3, 'blue': 2, 'green': 1}
best = max(counts, key=counts.get)
print("Most common:", best, counts[best])

raw = "  Hello, HELLO... bye! "
clean = raw.lower().strip(" .,!")
print(clean)
```

Output:

```
Most common: red 3
hello, hello... bye
```

Tip: max(counts, key=counts.get) means the key with the biggest count. Without the if in Example 19A, a new word crashes because counts[word] does not exist yet.

Try it - Section 19

- 19.1 Copy the counter. Run it and read the dict.
- 19.2 Change the text to three words you choose. Which wins?
- 19.3 Print the most-common word AND its count (copy Example 19B).
- 19.4 Clean raw with lower + strip. Print before and after.
- 19.5 Count LETTERS instead of words (loop over the string, not .split()).
- 19.6 PREDICT: Do Red and red count together before cleaning? After .lower()?
- 19.7 FIX IT: Why does this crash on a new word? Fix it with if.

Code:

```
counts = {}
for word in "red blue red".split():
    counts[word] = counts[word] + 1
```

Challenge 19 - Top 3 Words

Paste a sentence from another subject. Print the top 3 words + counts. Hint: sort with sorted(counts, key=counts.get, reverse=True).

---

20. Search - Linear vs binary

Linear search checks each item in turn. Binary search halves a SORTED list each time - much faster on big lists.

Example 20A - Linear search (count the steps)

Code:

```
names = ["Aisha", "Ben", "Cara", "Dan", "Eva"]
target = "Cara"
checks = 0
for n in names:
    checks = checks + 1
    if n == target:
        print("Found at step", checks)
        break
```

Output:

```
Found at step 3
```

Example 20B - Binary search (list MUST be sorted)

Code:

```
nums = [4, 7, 12, 19, 25, 33]
target = 19
low, high = 0, len(nums) - 1
while low <= high:
    mid = (low + high) // 2
    if nums[mid] == target:
        print("Found at index", mid)
        break
    elif nums[mid] < target:
        low = mid + 1
    else:
        high = mid - 1
```

Output:

```
Found at index 3
```

Tip: Trace Example 20B on paper: low=0, high=5, mid=2 (12 < 19 so low=3). Then low=3, high=5, mid=4 (25 > 19 so high=3). Then mid=3 - found!

Try it - Section 20

- 20.1 Run linear search for Cara. How many checks? Now search for Aisha.
- 20.2 Search for Zoe (not there). What prints? Add a Found it? yes/no flag.

- 20.3 Rough timing: search a 7-item list vs a 700-item list (use range). Which is slower?
- 20.4 Run binary search for 19. Trace low, mid, high on paper, then check.
- 20.5 Search for 20 (missing). Where does the loop end? Print low after the loop.
- 20.6 PREDICT: Binary search on [4, 33, 7] unsorted for 7 - works? Try it.
- 20.7 FIX IT: while low < high misses when one slot is left. Fix it.

Code:

```
nums = [4, 7, 12, 19, 25, 33]
target = 4
low, high = 0, len(nums) - 1
while low < high:
    mid = (low + high) // 2
    if nums[mid] == target:
        print("Found at index", mid)
        break
    elif nums[mid] < target:
        low = mid + 1
    else:
        high = mid - 1
```

Challenge 20 - Race Them

Which is faster on 1000 sorted numbers - linear or binary? Count the checks for each. Try target 999 and target 0.

---

21. Sorting - Selection and bubble, count the work

Sorting puts a list in order. Count the work (comparisons / swaps) to see which method does more.

Example 21A - Selection sort (find smallest, swap to front)

Code:

```
data = [5, 2, 9, 1, 6]
comparisons = 0
for i in range(len(data)):
    smallest = i
    for j in range(i + 1, len(data)):
        comparisons = comparisons + 1
        if data[j] < data[smallest]:
            smallest = j
    data[i], data[smallest] = data[smallest], data[i]
print(data)
print("Comparisons:", comparisons)
```

Output:

```
[1, 2, 5, 6, 9]
Comparisons: 10
```

Example 21B - Bubble sort (swap neighbours, repeat)

Code:

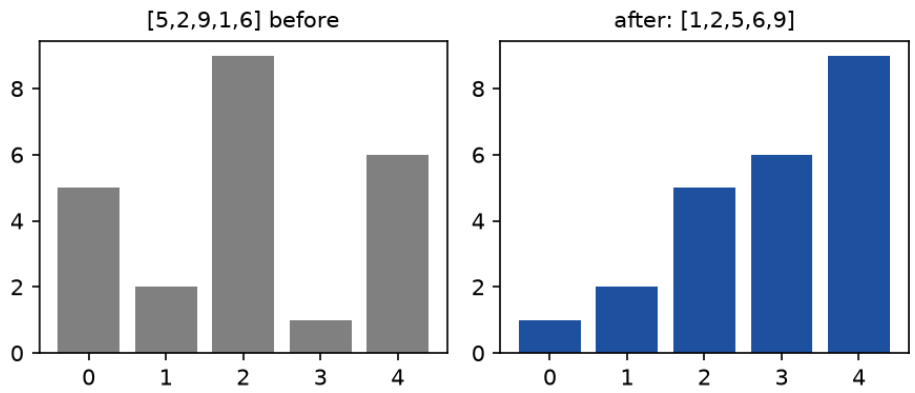
```
data2 = [5, 2, 9, 1, 6]
swaps = 0
for _ in range(len(data2)):
    for i in range(len(data2) - 1):
        if data2[i] > data2[i + 1]:
            data2[i], data2[i + 1] = data2[i + 1], data2[i]
            swaps = swaps + 1
print(data2)
print("Swaps:", swaps)
```

Output:

```
[1, 2, 5, 6, 9]
Swaps: 5
```

Before and after as bars:

Sorting before and after bars



Try it - Section 21

- 21.1 Run selection sort. Check the list AND the comparison count (10).
- 21.2 Change the list to [1, 2, 3, 4, 5] (already sorted). Does the count change? Why not?
- 21.3 Run bubble sort. Compare swaps (5) vs selection's comparisons (10).
- 21.4 Sort your CSV marks (Section 17) smallest-first with one of them.
- 21.5 PREDICT: Which does more work on 6 items - selection or bubble? Run and compare.
- 21.6 FIX IT: The swap line is missing. What breaks? Fix it.

Code:

```
data = [5, 2, 9, 1, 6]
for i in range(len(data)):
    smallest = i
    for j in range(i + 1, len(data)):
        if data[j] < data[smallest]:
            smallest = j
print(data)
```

Challenge 21 - Sort Dicts Two Ways

Sort class_list dicts by mark using sort(key=...), then do it by hand with selection sort. Check both give the same order.

---

22. Recursion + quicksort (HL stretch)

Recursion = a function that calls itself. You MUST have a STOP case, or it never ends. Keep lists SMALL - deep recursion can hit the 10 s guard.

Example 22A - Factorial (5 x 4 x 3 x 2 x 1)

Code:

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))
```

Output:

```
120
```

Tip: Trace on paper: 5 -> 4 -> 3 -> 2 -> 1 (stop), then multiply back up: 1 x 2 x 3 x 4 x 5 = 120.

Example 22B - Quicksort (pivot, smaller-left, bigger-right)

Code:

```
def quicksort(items):
    if len(items) <= 1:
        return items
    pivot = items[0]
    left = [x for x in items[1:] if x < pivot]
    right = [x for x in items[1:] if x >= pivot]
    return quicksort(left) + [pivot] + quicksort(right)

print(quicksort([5, 2, 9, 1, 6]))
```

Output:

```
[1, 2, 5, 6, 9]
```

Tip: First pivot is 5. Left = [2, 1], right = [9, 6]. Each side gets quicksorted the same way.

Try it - Section 22

- 22.1 Run factorial(5). Trace the calls on paper: 5 -> 4 -> 3 -> 2 -> 1, then multiply back up.
- 22.2 Print factorial(0). Why does the <= 1 stop case matter?
- 22.3 Run quicksort on [5, 2, 9, 1, 6]. What was the first pivot? Left? Right?
- 22.4 Sort [3, 1, 2] by hand with the pivot rule, then check with Python.
- 22.5 PREDICT: Quicksort on already-sorted [1, 2, 3, 4, 5] - how deep does it go?
- 22.6 FIX IT: Remove the stop case and run on 5 items. What error? Put it back.

Code:

```
def quicksort(items):
    pivot = items[0]
    left = [x for x in items[1:] if x < pivot]
    right = [x for x in items[1:] if x >= pivot]
    return quicksort(left) + [pivot] + quicksort(right)

print(quicksort([5, 2, 9, 1, 6]))
```

Challenge 22 - Count the Work (HL)

Count comparisons in quicksort vs selection sort (Section 21) on the same 8-item list. Which does fewer? Try a sorted list and a mixed list.

---

23. Ciphers - Caesar and substitution

A cipher scrambles text so only someone with the key can read it. Caesar shifts every letter. Substitution swaps the whole alphabet.

Example 23A - Caesar (shift forward 3, back -3 to decode)

Code:

```
def caesar(text, shift):
    out = ""
    for ch in text.lower():
        if "a" <= ch <= "z":
            out += chr((ord(ch) - ord("a") + shift) % 26 + ord("a"))
        else:
            out += ch
    return out

secret = caesar("attack at dawn", 3)
print(secret)
print(caesar(secret, -3))
```

Output:

```
dwwdfn dw gdzq
attack at dawn
```

Example 23B - Substitution (swap alphabets)

Code:

```
plain = "abcdefghijklmnopqrstuvwxyz"
cipher = "qwertyuiopasdfghjklzxcvbnm"

def substitute(text):
    out = ""
    for ch in text.lower():
        if ch in plain:
```

```
            out += cipher[plain.index(ch)]
        else:
            out += ch
    return out

print(substitute("hello"))
```

Output:

```
itssg
```

Tip: To decode substitution, run the same loop with plain and cipher swapped.

Try it - Section 23

- 23.1 Run Example 23A. Decode secret back with shift -3.
- 23.2 Encode your name with shift 3. Swap with a neighbour to decode.
- 23.3 Try shift 13 twice on the same text - what do you notice? (This is called ROT13.)
- 23.4 Substitution: encode hello (itssg). Now decode it by swapping plain and cipher.
- 23.5 Encode a full sentence with substitution. Which cipher was harder to break by hand?
- 23.6 PREDICT: Caesar with shift 26? Shift 0? Try both.
- 23.7 FIX IT: Uppercase input breaks the check. Why does .lower() fix it?

Code:

```
def caesar(text, shift):
    out = ""
    for ch in text:
        if "a" <= ch <= "z":
            out += chr((ord(ch) - ord("a") + shift) % 26 + ord("a"))
        else:
            out += ch
    return out

print(caesar("HELLO", 3))
```

Challenge 23 - Brute Force

Write brute(message) that tries all 26 Caesar shifts and prints each - spot the English one. Test it on dwwdfn dw gdzq.

---

24. Models - Rules plus a hand-fit line

A model turns inputs into a prediction. First model: hand-written rules. Second model: fit a line through data points (regression by hand - no extra libraries).

Example 24A - Rules-based model

Code:

```
def frost_risk(temp, rain):
    if temp < 0 and rain < 5:
        return "HIGH"
    elif temp < 3:
        return "MEDIUM"
    return "LOW"

print(frost_risk(-2, 1))
print(frost_risk(8, 12))
```

Output:

```
HIGH
LOW
```

Example 24B - Fit a line by hand (study hours -> marks)

Code:

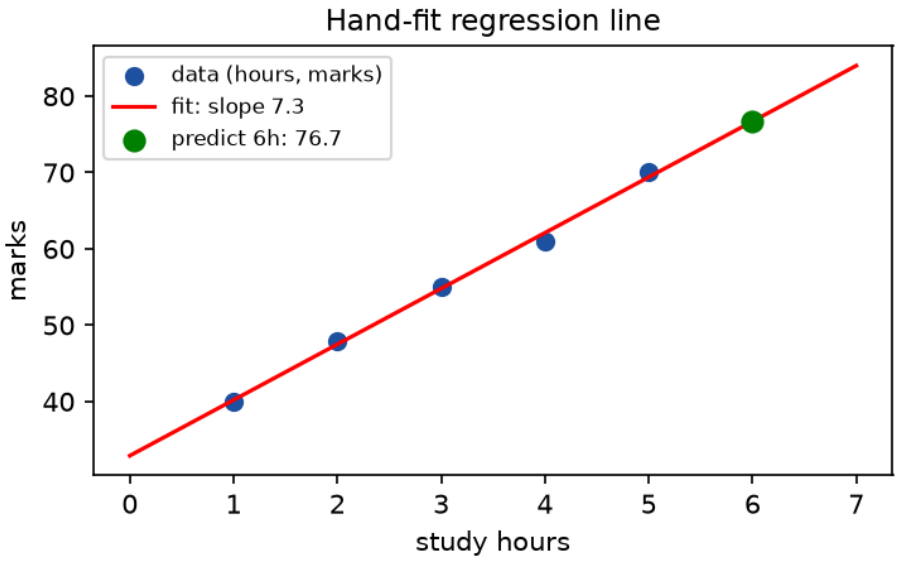
```
xs = [1, 2, 3, 4, 5]
ys = [40, 48, 55, 61, 70]
mx = sum(xs) / len(xs)
my = sum(ys) / len(ys)
slope = sum((x - mx) * (y - my) for x, y in zip(xs, ys)) / sum((x - mx) ** 2 for x in xs)
intercept = my - slope * mx
print("slope:", round(slope, 2), "intercept:", round(intercept, 2))
print("Predict 6 hours:", round(slope * 6 + intercept, 1))
```

Output:

```
slope: 7.3 intercept: 32.9
Predict 6 hours: 76.7
```

Tip: Slope = how many extra marks per extra hour (about 7.3). Now change ONE data point and watch the line move.

Hand-fit regression line with 6-hour prediction



Try it - Section 24

- 24.1 Run the frost model. Test (-2, 1), (2, 0), (8, 12) - HIGH / MEDIUM / LOW?
- 24.2 Add a rule: rain > 20 always returns LOW (heavy rain, no frost). Re-test.
- 24.3 Run the regression. What does the slope mean in plain words?
- 24.4 Change the last mark from 70 to 40. How do slope and prediction move?
- 24.5 PREDICT: Predict 0 hours and 10 hours. Sensible? Where does the model break?
- 24.6 FIX IT: Lists of different lengths break silently. Check len first.

Code:

```
xs = [1, 2, 3, 4, 5]
ys = [40, 48, 55]
print(len(xs), len(ys))
```

Challenge 24 - Fit Your Data

Fit YOUR data (Section 17 marks vs row number). Predict the next row. Is the prediction sensible?

---

25. Simulation - Dice models and what-ifs

A model is PURPOSE + INPUTS + ASSUMPTIONS. Simulate many rolls, count what happens, then change ONE thing (what-if) and compare.

Example 25A - Fair die (1000 rolls, count sixes)

Code:

```
import random
sixes = 0
rolls = 1000
for _ in range(rolls):
    if random.randint(1, 6) == 6:
        sixes = sixes + 1
print("Sixes:", sixes, "out of", rolls)
```

Example run (yours will differ - random!):

Output:

```
Sixes: 168 out of 1000
```

Example 25B - What-if: loaded die (six 1 time in 4)

Code:

```
import random
loaded = 0
rolls = 1000
for _ in range(rolls):
    if random.random() < 0.25:
        loaded = loaded + 1
print("Loaded sixes:", loaded)
```

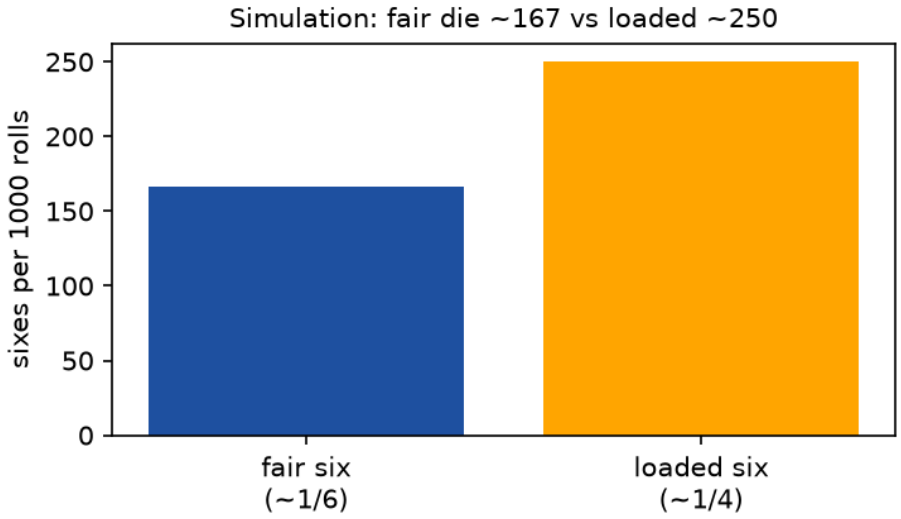
Example run:

Output:

```
Loaded sixes: 251
```

Tip: Random means every run differs. Use LOTS of rolls (1000+) so the share settles down. About 1/6 (~167/1000) for fair, about 1/4 (~250/1000) for loaded.

Expected sixes: fair vs loaded die



Try it - Section 25

- 25.1 Run the fair-die model 3 times. Same answer? Why not?
- 25.2 Change rolls to 100, then 10000. What happens to the share of sixes?
- 25.3 Run the loaded die. Compare fair vs loaded over 1000 rolls.
- 25.4 WHAT-IF: what if six counts double (house rule)? Change the counter and re-run.
- 25.5 Model check: purpose? inputs? assumptions? Write one line for each.
- 25.6 PREDICT: random.randint(1, 6) vs random.random() < 1/6 - same idea? Test both.
- 25.7 FIX IT: sixes reset INSIDE the loop always gives 0 or 1. Spot it, fix it.

Code:

```
import random
for _ in range(1000):
    sixes = 0
    if random.randint(1, 6) == 6:
        sixes = sixes + 1
print("Sixes:", sixes)
```

Challenge 25 - Two Dice Sevens

Model 2 dice - count how often the total is 7 in 1000 rolls. What share? Hint: roll two dice inside the loop and add them.

---

26. Testing - Prove it works with asserts

Testing rule: decide the answer BEFORE you run. assert passes silently, and SHOUTS on failure.

Example 26A - Test add

Code:

```
def add(a, b):
    return a + b

assert add(2, 3) == 5
assert add(-1, 1) == 0
assert add(0, 0) == 0
print("add passed!")
```

Output:

```
add passed!
```

Example 26B - Test your own function

Code:

```
def average(nums):
    return sum(nums) / len(nums)

assert average([10, 20, 30]) == 20
assert average([5]) == 5
print("average passed!")
```

Output:

```
average passed!
```

Tip: Write a test TABLE first (input -> expected -> got), then the code. Break add on purpose (return a - b) and run: which assert shouts first?

Try it - Section 26

- 26.1 Run Example 26A. Both passed! lines print - all green.
- 26.2 Break add on purpose (return a - b). Run: which assert shouts first? Fix it back.
- 26.3 Add a test: assert add(100, 200) == 300.
- 26.4 Write a test table for double(n) from Section 16: 3 cases, then the function.
- 26.5 PREDICT: assert average([]) == 0 - passes or crashes? Try it (read the error).
- 26.6 FIX IT: Guard average so empty lists return 0.

Code:

```
def average(nums):
    return sum(nums) / len(nums)

assert average([]) == 0
```

- 26.7 FIX IT: This test table has 1 wrong expected value. Which one?

Code:

```
def grade(mark):
    if mark >= 70:
        return "Gold"
    elif mark >= 40:
        return "Silver"
```

```
    return "Bronze"

assert grade(85) == "Gold"
assert grade(50) == "Silver"
assert grade(20) == "Gold"
```

Challenge 26 - Full Grade Tests

Test your grade(mark) function (Section 16): 85 Gold, 50 Silver, 20 Bronze, plus boundaries 70 and 40. All 5 must pass.

---

27. Final Mini-Projects - Pick ONE (or do all three!)

You now know lists, dicts, functions, files, stats, search, sort, ciphers, models and tests! Each project has steps. You have done each step before - just join them together.

Mini-Project 27A - CSV Leaderboard + Stats (Medium)

Steps:

- 1. Load data.csv with csv.DictReader (Section 17).
- 2. Clean it: strip names, skip blanks, int() the marks.
- 3. Sort highest-first and print the leaderboard.
- 4. Print mean / median / mode + a text bar chart (Section 18).
- 5. Add assert tests for your average function (Section 26).

Example run:

Output:

```
Cara 81
Aisha 72
Dan 64
Ben 45
Mean: 65.5
Median: 68.0
Mode: 72
```

Mini-Project 27B - Text Detective (Medium)

Steps:

- 1. Paste a paragraph from another subject into a text variable.
- 2. Clean it (lower + strip punctuation) and count words with a dict (Section 19).
- 3. Print the top 3 words + counts.
- 4. Sort the words by count with sorted(..., key=..., reverse=True) (Section 21).
- 5. Caesar-encode your summary sentence with shift 3 so only your group can decode it (Section 23).

Mini-Project 27C - Model + Simulation + Tests (Harder)

Steps:

- 1. Write a rules model (e.g. frost_risk or your own grade/price rules) as a function (Sections 16 + 24).
- 2. Write 5 assert tests for it, including boundaries (Section 26).
- 3. Simulate 1000 runs of a dice or population model with random (Section 25).
- 4. Do a WHAT-IF: change one input and compare the two outputs.
- 5. Write one line each for purpose, inputs, assumptions, and where the model breaks.

Well done! If you finished all sections, you know lists, dicts, functions, files/CSV, statistics, text analytics, linear/binary search, selection/bubble sort, recursion/quicksort, ciphers, rules + regression models, simulation and testing - all the Leaving Cert Python beyond the basics.

Checklist - I can...

- [ ] make and index lists, append, loop, total and average
- [ ] make dicts, update fields, loop over a list of dicts
- [ ] define functions with params and return, call them
- [ ] load CSV files, clean rows, sort leaderboards
- [ ] work out mean/median/mode and draw text bar charts
- [ ] count words with dicts, clean text, find most common
- [ ] linear search and binary search (sorted only), count checks
- [ ] selection sort and bubble sort, count comparisons/swaps
- [ ] trace recursion (factorial) and quicksort with a pivot
- [ ] Caesar + substitution encode/decode, brute-force 26 shifts
- [ ] rules models + hand-fit regression line, predict and test limits
- [ ] simulate with random, do what-ifs, state purpose/inputs/assumptions
- [ ] write assert test tables, guard empty inputs, fix failing tests