

Workbook

Practice Problems for Programming 0 — Core, Intermediate and Stretch

About This Workbook

This workbook contains 79 Python practice problems, split into three difficulty tiers:

Core (44 problems)

Start here. Single loops, single conditionals, and direct arithmetic — the standard building blocks.

Intermediate (23 problems)

Combines two or more of the Core patterns, or introduces a new wrinkle — a little trickier.

Stretch (12 problems)

The most demanding problems in the set.

Start in Core and work upward as you gain confidence; within each tier, problems are listed alphabetically by function name, so pick whichever you like, rather than working strictly top to bottom.

Unless a problem's Tip specifically mentions one, don't use built-in Python functions or packages to solve it. The point of each problem is to practise writing the basic logic yourself — practicing assigning variables, loops, conditionals, and so on — rather than reaching for a shortcut that does the work for you. Where a built-in genuinely helps, the Tip will name it directly.

For every function listed in the tables below, a Python script named after that function is available — for example, the problem `sum_list()` belongs in a file called `sum_list.py`. Each script follows the same template, below.

The `main()` function is where you test your work: call your function with some inputs — including the one from the Description column's worked example — and check the result matches. Wrapping the call in a `print()` is the easiest way to see the result while you're testing. However, this isn't what we might call more 'proper' testing (this will be introduced later in the course).

```
def function_name():  
    pass  
  
def main():  
    function_name()  
  
if __name__ == "__main__":  
    main()
```

The final three lines in every script are ‘boilerplate’: that should appear in every script, unchanged:

```
if __name__ == "__main__":  
    main()
```

Do not change this area of the script, just test from main() and implement.

Skeleton code is available to download (solutions in a future week). Place in your CODE_HOME folder:

```
└─ workbook  
    └─ skeletons  
        │ └─ core  
        │ └─ intermediate  
        └─ stretch  
    └─ solutions  
        └─ core  
        └─ intermediate  
        └─ stretch
```

Problems by Tier

CORE

Core (44 problems)

Start here. Single loops, single conditionals, and direct arithmetic — the standard building blocks.

Function	Description	Problem Rationale
<code>abs_len_diff(list_a, list_b, n)</code>	Receives list_a and list_b, two lists, and n, a number. Work out the absolute difference between the two lists' lengths, raise that difference to the power of n, and return the result. Example: <code>abs_len_diff([1, 2, 3, 4], [5], 3)</code> should return 27, since the lengths differ by $4-1 =3$, and 3 cubed is 27. Tip: the built-in <code>len()</code> function gives you each list's length directly.	Uses <code>len()</code> and basic arithmetic only — no loop needed. Skills: <code>len()</code>, exponentiation, absolute value.
<code>abs_sum(a, b)</code>	Receives two numbers, a and b. Add them together, then return the result as a whole, positive number — meaning any decimal part is dropped and any negative sign is removed. Example: <code>abs_sum(3.14, 5)</code> should return 8, since $3.14 + 5 = 8.14$, and the absolute integer value of 8.14 is 8. Tip: some conditional logic (or the built-in <code>abs()</code> function) and the <code>int()</code> function might both be of use here.	One arithmetic step plus a single conditional for the absolute value. Skills: arithmetic, conditionals, absolute value.
<code>absolute_diff(m, n)</code>	Receives two numbers, m and n. Subtract one from the other and return the result as a positive number (its absolute value), regardless of which of m or n is larger. Example: <code>absolute_diff(9.5, 11)</code> should return 1.5, since $11 - 9.5 = 1.5$. Tip: the built-in <code>abs()</code> function returns the absolute value of a number directly, although your own conditional logic can also be used.	A single subtraction plus an absolute-value step. Skills: arithmetic, absolute value.

Function	Description	Problem Rationale
<code>area_of_rectangle(base, height)</code>	Receives base and height, two numbers describing a rectangle. Multiply them together and return the area. Example: <code>area_of_rectangle(10, 7.5)</code> should return 75.0, since $10 \times 7.5 = 75.0$.	A single multiplication — the simplest possible formula. Skills: arithmetic formulas.
<code>capitalize_words(sentence)</code>	Receives sentence, a string of one or more words separated by spaces. Capitalise the first letter of every word, and return the new sentence. Example: <code>capitalize_words("the quick fox")</code> should return "The Quick Fox". Tip: the string method .split() breaks a sentence into a list of words, and .join() puts a list of strings back together with spaces between them. You might also look at the .upper() method of the string. Of course, you <i>could</i> use the <code>.title()</code> function on the sentence, but that would miss the point of the exercise...try and implement without using it.	Combines splitting, per-word transformation, and rejoining — a natural first exercise using <code>.upper()</code> , <code>.split()</code> and <code>.join()</code> together. Skills: string splitting, string methods, list building.
<code>count_occurrences(a_list, target)</code>	Receives a_list, a list of items, and target, a single value to search for. Count how many times target appears in a_list, and return that count. Example: <code>count_occurrences([1, 2, 2, 3, 2], 2)</code> should return 3, since 2 appears three times.	A single loop with one comparison — the standard accumulator pattern, counting matches instead of summing values. Skills: loops, comparison, accumulator pattern.
<code>count_vowels(text)</code>	Receives text, a string. Count how many of its characters are vowels (a, e, i, o or u, in either case), and return that count. Example: <code>count_vowels("Hello World")</code> should return 3, since the vowels are e, o and o.	A single loop with one condition — the standard scanning-and-counting pattern. Skills: loops, conditionals, membership checks.

Function	Description	Problem Rationale
<code>cut_and_flip(string, index)</code>	Receives string, a piece of text, and index, a number marking a cut point. Cut the string into two pieces: the first piece runs from the start up to and including the character at index, and the second piece is everything after that. Return the second piece followed by the first piece, joined together. Example: <code>cut_and_flip("some!awe", 4)</code> should return "awesome!", since cutting at index 4 gives "some!" and "awe", and joining them second-piece-first gives "awe" + "some!" = "awesome!". Tip: Python's string slicing (e.g. <code>string[:5]</code> and <code>string[5:]</code>) is the natural tool here.	Two string slices swapped and joined — no loop needed once you see the slicing pattern. Skills: string slicing, string concatenation.
<code>factorial(n)</code>	Receives a single number, n. If n is negative, skip the maths and return the string "NAN" instead. Otherwise, work out and return the factorial of n: the result of multiplying every whole number from 1 up to n together. Example: <code>factorial(5)</code> should return 120, since $1 \times 2 \times 3 \times 4 \times 5 = 120$. Tip: check the definition of a factorial — you might need some if logic here.	A loop with a running product, plus one guard condition for negative input. Skills: loops, accumulator pattern, guard conditions.
<code>fahrenheit_to_celsius(fahrenheit)</code>	Receives fahrenheit, a temperature in degrees Fahrenheit. Convert it to degrees Celsius using the formula $^{\circ}\text{C} = (^{\circ}\text{F} - 32) \times 5/9$, and return the result. Example: <code>fahrenheit_to_celsius(68)</code> should return 20.0, since $(68 - 32) \times 5/9 = 20.0$.	A single formula with no branching or loop — the simplest possible conversion problem. Skills: arithmetic formulas.
<code>filter_non_negatives(values)</code>	Receives values, a list of numbers. Build and return a new list containing only the values that are zero or greater — in other words, leave out every negative number. Example: <code>filter_non_negatives([3, -2, 5, -1])</code> should return [3, 5]. Tip: build the new list inside a for loop, adding each value only when an if condition is met.	A single filtering pass over a list — keep-if-condition is the whole task. Skills: loops, conditionals, list building.

Function	Description	Problem Rationale
<code>first_repeated_character(text)</code>	Receives text, a string. Scan through it from the start, and return the first character that has already appeared earlier in the string. If no character repeats, return None. Example: <code>first_repeated_character("abcbad")</code> should return "b", since 'b' is the first character to reappear. Tip: keep a list (or a set) of characters seen so far as you loop, and check each new character against it with in before adding it.	Requires keeping track of everything seen so far while scanning — a genuine step up from a single-pass count or filter. Skills: loops, membership checks, state tracking.
<code>is_above_threshold(number, threshold)</code>	Receives number and threshold, two numbers. If number is greater than threshold, return the integer 1; otherwise, return the integer 0 (note: this returns 1 or 0, not True or False). Example: <code>is_above_threshold(15, 10)</code> should return 1, since 15 is greater than 10. Tip: a single if / else is enough — just make sure you return the numbers 1 and 0, not the booleans True and False.	A single comparison, returning 1/0 instead of True/False. Skills: conditionals, comparison.
<code>is_leap_year(year)</code>	Receives year, a whole number. Return True if it's a leap year, or False otherwise. A year is a leap year if it's divisible by 4, except that years divisible by 100 are not leap years, unless they're also divisible by 400. Example: <code>is_leap_year(2000)</code> should return True (divisible by 400), but <code>is_leap_year(1900)</code> should return False (divisible by 100 but not 400). Tip: three separate % checks combined with and/or are enough — work through the three rules in the order given, since each one is an exception to the one before it.	Three overlapping conditions layered as exceptions to each other — a step up from the single- or double-condition checks earlier in Core. Skills: modulo arithmetic, nested conditionals, boolean logic.
<code>is_letter(character)</code>	Receives character, a single character. If it is a letter in the alphabet return True. If it isn't a letter at all — for example a digit or a symbol — return False. Example: <code>is_letter('@')</code> should return False, since '@' isn't a letter. Tip: the built-in string method .isalpha() can check whether a character is a letter at all.	A membership check against a small fixed set, plus one extra condition for non-letters. Skills: membership checks, conditionals.

Function	Description	Problem Rationale
<code>is_palindrome(word)</code>	Receives word, a string. Check whether it reads the same forwards and backwards, and return True if so, or False otherwise. Example: <code>is_palindrome("level")</code> should return True, since reversing "level" gives the same word.	A single comparison once you know the reversal trick. Skills: string reversal, comparison.
<code>is_square_matrix(matrix)</code>	Receives matrix, a list of lists representing a 2D grid, where each inner list is one row. Return True if the grid has exactly as many rows as it has columns (making it square), and False otherwise. Example: <code>is_square_matrix([[1, 2], [3, 4]])</code> should return True, since it has 2 rows and 2 columns. Tip: the built-in <code>len()</code> function tells you how many rows there are, and can be used also to tell how many columns each row has.	A single length comparison between rows and columns. Skills: <code>len()</code> , nested lists, comparison.
<code>is_strings(values)</code>	Receives values, a list of items of any type. Check every item in the list: if every single one is a string, return True; if even one item isn't a string, return False. Example: <code>is_strings(["R", "2", "D", "2"])</code> should return True, since all four items are strings (note that "2" here is text, not the number 2). Tip: the built-in <code>isinstance()</code> function, used with the <code>str</code> type, can check whether a value is text.	A loop with one type check per element — the standard 'all elements satisfy X' pattern. Skills: loops, type checking, conditionals.
<code>is_whole_number(number)</code>	Receives number, a real number (which may include a decimal point). Return True if number has no fractional part — that is, if it's a whole number like 9.0 or 4 — and return False if it has any decimal digits that aren't zero. Example: <code>is_whole_number(9.0)</code> should return True, since 9.0 has no fractional part. Tip: Python's <code>%</code> operator (modulo), checking <code>number % 1</code>, tells you whether there's a fractional part.	A single modulo check — renamed from <code>is_even_or_odd</code> , since the worked examples actually test whole-ness rather than parity. Skills: modulo, conditionals.

Function	Description	Problem Rationale
<code>is_within_range(number, a_range)</code>	Receives number, a single number, and a_range, a list of two numbers [low, high]. Return True if number falls strictly between low and high, with neither the exact low nor the exact high value counting as 'within range'. Return False otherwise. Example: <code>is_within_range(5, [1, 10])</code> should return True, since 5 is strictly between 1 and 10. Tip: a single combined condition using and (checking both boundaries at once) is neater than two separate if statements.	One two-sided comparison — pay attention to 'exclusively' when the number equals a boundary. Skills: conditionals, combined boolean logic.
<code>largest_diff(numbers)</code>	Receives numbers, a list of numbers. Look at each pair of neighbouring values in the list, work out the absolute difference between them, and return the single largest difference found anywhere in the list. Example: <code>largest_diff([3, 10, 4, 12])</code> should return 8, since the neighbouring differences are $3-10 =7$, $10-4 =6$ and $4-12 =8$, and the largest of these is 8. Tip: looping with <code>range(1, len(numbers))</code> lets you compare each value with the one before it (at <code>index - 1</code>) as you go, and <code>abs()</code> handles the absolute difference.	One pass comparing each element to its neighbour while tracking the largest gap seen. Skills: loops, indexing, absolute value, comparison.
<code>list_product_root(numbers)</code>	Receives numbers, a list of numbers. Multiply every element in the list together, take the square root of that product, and return the result rounded down to a whole number (an integer). Example: <code>list_product_root([9, 8, 98, 2, 4])</code> should return 237, since $9 \times 8 \times 98 \times 2 \times 4 = 56,448$, and the square root of 56,448 is approximately 237.6, which becomes 237 as an integer. Tip: the built-in <code>int()</code> function will truncate a decimal number down to a whole number for you.	Combines a running-product loop with one extra arithmetic step at the end. Skills: loops, accumulator pattern, square roots, type conversion.

Function	Description	Problem Rationale
<code>min_max(numbers)</code>	Receives numbers, a list of numbers. Find the smallest and largest values in the list, and return them together as a two-item list: [minimum, maximum]. Example: <code>min_max([4, 9, 1, 7])</code> should return <code>[1, 9]</code>.	A single loop tracking two running values instead of one — a natural extension of the basic accumulator pattern. Skills: loops, comparison, accumulator pattern.
<code>multiply_five(a, b, c, d, e)</code>	Receives five separate numbers, named a through e (not a list). Multiply all five together and return the result. Example: <code>multiply_five(1, 1, 1, 2, 10)</code> should return <code>20</code>, since $1 \times 1 \times 1 \times 2 \times 10 = 20$. Tip: no loop or branching is needed here — a single expression will do.	Fixed number of arguments; no loop or branching required. Skills: fixed-arity arguments, chained multiplication.
<code>multiply_to_strings(scalar, numbers)</code>	Receives scalar, a single number, and numbers, a list of numbers. Multiply every value in the list by scalar, convert each result to a string, and return the new list of strings. Example: <code>multiply_to_strings(3.1, [150, 200])</code> should return <code>["465.0", "620.0"]</code>, since $150 \times 3.1 = 465.0$ and $200 \times 3.1 = 620.0$, each written as text. Tip: the built-in <code>str()</code> function converts a number into text.	A transformation loop plus a type conversion on each element. Skills: loops, type conversion.
<code>multiply_values(numbers, n)</code>	Receives numbers, a list of numbers, and n, a single number. Multiply every value in the list by n, and return the results as a new list. Example: <code>multiply_values([1, 2, 3], 3)</code> should return <code>[3, 6, 9]</code>. Tip: build a new list inside a for loop, multiplying each value by n as you go.	A single transformation applied to every element — no branching, no accumulation, just a straightforward loop. Skills: loops, list transformation.

Function	Description	Problem Rationale
<code>mystery_toggles(a_list)</code>	Receives a_list, a list of True/False values. Flip (invert) the values at indices 1, 2 and 3 specifically, leaving every other index unchanged, and return the modified list. If the list isn't long enough to have an index 3, return an empty list instead. Example: <code>mystery_toggles([True, True, True, False])</code> should return <code>[True, False, False, True]</code>, since the values at indices 1, 2 and 3 have all been flipped. Tip: not flips a True/False value directly (e.g. not True is False).	A fixed, small set of indices to flip plus a single length guard — mechanical once you see the pattern. Skills: list indexing, boolean negation, guard clauses.
<code>odd_total(numbers)</code>	Receives numbers, a list of numbers. Add together only the values that are themselves odd numbers (this is about the value of each number, not its position in the list), and return that total. Example: <code>odd_total([1, 2, 3, 4, 5])</code> should return 9, since 1 + 3 + 5 = 9. Tip: a number is odd when dividing it by 2 leaves a remainder of 1 — Python's % operator can check this directly.	An accumulator-plus-filter pattern, filtering by value instead of position. Skills: loops, conditionals, modulo.
<code>ones(n)</code>	Receives n, a single whole number. Build and return a string made up of exactly n copies of the character "1". Example: <code>ones(5)</code> should return "11111", a string of five "1" characters. Tip: adding one "1" to a result string inside a for loop, n times, builds the string up gradually. You can also achieve this without a for loop.	The simplest possible loop-and-build pattern in the set. Skills: loops, string building.
<code>paint_needed(length, width)</code>	Receives length and width, two numbers describing the dimensions of a wall in metres. Work out the area of the wall, then multiply it by 1.5 (since the wall needs 1.5 litres of paint per square metre), and return the total litres required. Example: <code>paint_needed(3, 5)</code> should return 22.5, since the area is 3 × 5 = 15 square metres, and 15 × 1.5 = 22.5 litres.	A direct formula with no branching or loop. Skills: arithmetic formulas.

Function	Description	Problem Rationale
<code>position(road)</code>	Receives road, a string that simulates a one-lane road: '*' marks an empty space and 'c' marks a car. There is exactly one car on the road. Find where it is and return its position as a number, counting the first character of the string as position 1 (not position 0). Example: <code>position("***c")</code> should return 4, since the 'c' is the 4th character in the string. Tip: looping with <code>range(len(road))</code> gives you each index directly, and <code>road[index]</code> looks up the character at that position.	A single loop over a string with one condition — the standard string-scanning pattern. Skills: loops, string indexing, conditionals.
<code>positions(road)</code>	Works like <code>position()</code> above, but the road may now contain more than one car. Receives road in the same format, and returns a list of every position where a 'c' appears — again counting the first character as position 1. Example: <code>positions("cc*c")</code> should return [1, 2, 4], since there are 'c' characters at the 1st, 2nd and 4th positions. Tip: <code>range(len(road))</code> works well here too, once you're building a list of positions instead of returning just one.	A single loop scanning a string and building a list of matches, rather than returning a single value. Skills: loops, string indexing, list building.
<code>powered_multiple(arg1, arg2, arg3)</code>	Receives three numbers: arg1, arg2 and arg3. Raise arg1 to the power of arg2 (multiply arg1 by itself arg2 times), then multiply that result by arg3, and return the final value. Example: <code>powered_multiple(2, 9, 3)</code> should return 1536, since 2 raised to the power of 9 is 512, and 512 × 3 = 1536. Tip: Python's <code>**</code> operator raises a number to a power directly (e.g. <code>2 ** 9</code>), so no loop is needed, but use one if you like.	Two arithmetic operators chained together; no loop or branching. Skills: exponentiation, operator chaining.
<code>product_sqrt(m, n)</code>	Receives two numbers, m and n. Multiply them together, then return the square root of that product. Example: <code>product_sqrt(4, 9)</code> should return 6, since 4 × 9 = 36, and the square root of 36 is 6. Tip: raising a number to the power of 0.5 (e.g. <code>36 ** 0.5</code>) gives its square root.	Two arithmetic operators chained together; no loop or branching. Skills: arithmetic, square roots.

Function	Description	Problem Rationale
<code>reverse_number(number)</code>	Receives number, a whole number. Reverse the order of its digits and return the result as a new whole number. Example: <code>reverse_number(123)</code> should return 321. Tip: repeatedly using the modulo (%) and floor-division (//) operators lets you peel off one digit at a time. If you'd rather not do it that way, converting number to a string with <code>str()</code> and back again with <code>int()</code>, in the process of solving the problem, is a perfectly good alternative.	Extracting digits one at a time via modulo and floor division — a common, reusable pattern. Skills: modulo, floor division, loops.
<code>row_totals(list_of_lists)</code>	Receives list_of_lists, a list where each element is itself a list of numbers (one row of data). Add up the numbers within each row separately, and return a new list containing each row's total, in the same order as the rows. Example: <code>row_totals([[1, 2], [4, 5, 6], [10]])</code> should return [3, 15, 10], since $1+2=3$, $4+5+6=15$, and the single-item row [10] totals 10. Tip: the built-in <code>sum()</code> function can add up each row for you, if you want to use it here.	A loop over rows, each summed independently. Skills: nested loops, accumulator pattern.
<code>safe_divide(m, n)</code>	Receives two numbers, m and n. Divide m by n and return the result. If n is zero, dividing isn't possible, so return the string "NAN" instead of attempting the division. Example: <code>safe_divide(10, 0)</code> should return "NAN", since dividing by zero isn't allowed. Tip: a single <code>if</code> check for whether n equals zero is enough to guard against the division.	One arithmetic operation guarded by a single zero-check. Skills: conditionals, division, guard clauses.
<code>signed_product(numbers)</code>	Receives numbers, a list of numbers. Multiply every element together. If the resulting product is positive or negative (i.e. not zero), return the string "is_signed"; if the product is exactly zero, return the string "unsigned" instead. Tip: keep a running product in a <code>for</code> loop, then check its sign with a single <code>if</code>.	A running-product loop plus a simple classification step at the end. Skills: loops, accumulator pattern, conditionals.

Function	Description	Problem Rationale
<code>single_left_shift(a_list)</code>	Receives a_list, a list of items. Move the first item to the end of the list, shifting every other item one position to the left, and return the new list. Example: <code>single_left_shift(['one','zero','two','nine','zero'])</code> should return <code>['zero','two','nine','zero','one']</code>, since 'one' (the first item) has moved to the end. Tip: slicing the list into two parts (e.g. a_list[1:] and a_list[:1]) and joining them back together is a clean way to do this.	A single slice-and-recombine operation, once you know the rotation pattern. Skills: list slicing, list concatenation.
<code>square_and_sum(lst)</code>	Receives lst, a list of numbers. Square every value in the list (multiply each one by itself), multiply each squared value by the approximate golden ratio (1.618), and return the new list of results. Tip: build the new list one value at a time inside a for loop.	Two chained transformations over a list — comfortable once basic loops are solid. Skills: loops, list transformation.
<code>square_each(numbers)</code>	Receives numbers, a list of numbers. Square every value in the list (multiply each one by itself), and return the new list of squared values. Example: <code>square_each([10, 5.5, 2, 1.4])</code> should return <code>[100, 30.25, 4, 1.96]</code>. Tip: build the new list inside a for loop, squaring each value as you go.	One transformation applied to every element — no branching. Skills: loops, list transformation.
<code>sum_list(a_list)</code>	Receives a_list, a list of numbers. Add every number in the list together and return the total as a single number. Example: <code>sum_list([5, 6, 20])</code> should return 31, since $5 + 6 + 20 = 31$. Tip: a for loop with a running total is all this needs.	Single loop with a running total — the most basic accumulator pattern. Skills: loops, accumulator pattern.
<code>sum_odd_indices(a_list)</code>	Receives a_list, a list of numbers. Add together only the values stored at odd index positions (index 1, 3, 5, and so on — remember indexing starts at 0), and return that total. Example: <code>sum_odd_indices([10, 20, 30, 40, 50])</code> should return 60, since the values at indices 1 and 3 are 20 and 40, and $20 + 40 = 60$. Tip: looping with range(len(a_list)) gives you each index directly, which you can then use to look up a_list[index].	The standard accumulator pattern, with one extra check on the index. Skills: loops, indexing, conditionals.

Function	Description	Problem Rationale
<code>temperature_status(temp)</code>	Receives temp, a temperature in degrees Celsius. Return the string "cold" if temp is below 10, "warm" if temp is between 10 and 25 inclusive, or "hot" if temp is above 25. Example: <code>temperature_status(5)</code> should return "cold", since 5 is below 10. Tip: a chain of if / elif / else conditions handles the three bands cleanly.	A three-band conditional — the standard shape for banded threshold logic. Skills: conditionals (if/elif/else).

INTERMEDIATE

Intermediate (23 problems)

Combines two or more of the Core patterns, or introduces a new wrinkle — a little trickier.

Function	Description	Problem Rationale
<code>balanced_parentheses(text)</code>	Receives text, a string that may contain the characters '(' and ')' among other characters. Return True if every opening parenthesis has a matching closing one in the correct order, and False otherwise. Example: <code>balanced_parentheses("(a(b)c)")</code> should return True, but <code>balanced_parentheses(")(")</code> should return False, since the closing parenthesis appears before its opening one. Tip: keep a running counter, adding 1 for every '(' and subtracting 1 for every ')' — the counter should never go below zero, and it must end at exactly zero.	The running-counter technique here is the first hint of what full bracket-matching (usually done with a stack) looks like, without needing a stack yet. Skills: loops, conditionals, running counters.
<code>bits_match_inverted(numbers, booleans)</code>	Receives numbers, a list containing only 0s and 1s, and booleans, a list of True/False values of the same length. The two lists are meant to line up using an inverted pairing: every 0 in numbers should be matched with True in booleans, and every 1 should be matched with False — the opposite of what you'd normally expect. Check every position in turn: if the pairing holds everywhere, return True; if it breaks at even one position, return False. Example: <code>bits_match_inverted([1, 1, 1], [True, False, True])</code> should return False, since the first 1 is paired with True, when a 1 should be paired with False. Tip: looping with <code>range(len(numbers))</code> lets you walk through both lists position by position at once, indexing into each one with the same counter.	The 0→True / 1→False mapping is the opposite of what you might expect — a genuine source of bugs, not just a mechanical loop. Skills: loops, paired indexing, boolean logic.

Function	Description	Problem Rationale
<code>black_hole(middle, radius, position)</code>	Receives middle, a list of two numbers giving the centre of a circle, radius, a number, and position, a list of two numbers giving another point in space. Work out the distance from position to middle (assume ordinary 2D Euclidean distance), then compare it to radius: return "Orbiting!" if position is outside the circle, "Event horizon!" if it's exactly on the edge, or "Beyond the event horizon!" if it's inside. Example: <code>black_hole([0, 0], 5, [3, 4])</code> should return "Event horizon!", since the distance from <code>[3, 4]</code> to <code>[0, 0]</code> is exactly 5, matching the radius. Tip: raising a number to the power of 0.5 (as in <code>distance ** 0.5</code>) gives its square root, which the Euclidean distance formula needs.	First problem needing the distance formula itself, plus a 3-way conditional on the result. Skills: distance formulas, square roots, multi-way conditionals.
<code>caesar_cipher(text, shift)</code>	Receives text, a string of lowercase letters, and shift, a whole number. Shift every letter in text forward by shift positions in the alphabet, wrapping back to 'a' after 'z', and return the new string. Example: <code>caesar_cipher("xyz", 3)</code> should return "abc", since shifting 'x', 'y' and 'z' forward by 3 wraps each of them back to the start of the alphabet. Tip: converting a letter to its alphabet position with <code>ord(character) - ord('a')</code>, adding shift, then using <code>% 26</code> to wrap around, and finally converting back with <code>chr()</code>, is the standard approach, and you can research those functions, but solve it however you like.	Introduces character-code arithmetic (<code>ord()/chr()</code>) and modular wraparound together. Skills: character codes, modulo arithmetic, string building.

Function	Description	Problem Rationale
<code>column_totals(matrix)</code>	Receives matrix, a list of lists representing a 2D grid of numbers, where each inner list is one row. Add up the values down each column (not across each row), and return a new list containing each column's total, in column order. Example: <code>column_totals([[1, 2], [3, 4], [5, 6]])</code> should return <code>[9, 12]</code>, since the first column is <code>1+3+5=9</code> and the second column is <code>2+4+6=12</code>. Tip: a nested loop works well here — loop over each column index first, and inside that, loop over each row, adding <code>matrix[row][column]</code> to that column's running total.	Needs to iterate 'down' columns rather than along rows — a genuine step up from single-list loops. Skills: nested loops, 2D indexing, accumulator pattern.
<code>digit_running_diff(number)</code>	Receives number, a whole number. Split it into its individual digits, then starting with the first digit, subtract each digit that follows it, one at a time, keeping a running total. Return the final result. Example: <code>digit_running_diff(1234)</code> should return <code>-8</code>, since <code>1 - 2 - 3 - 4 = -8</code>. Tip: converting the number to a string with <code>str()</code> makes it easy to loop over each digit one at a time.	Digit extraction combined with a subtract-in-sequence accumulator rather than a simple sum. Skills: digit extraction, accumulator pattern, string conversion.
<code>euclidean_distance(p, q)</code>	Receives p and q, two points of matching dimensions and returns the straight-line (Euclidean) distance between them. Return None if p and q are different lengths. Example: <code>euclidean_distance([0, 0], [1, 1])</code> should return <code>1.4142135623730951</code>, the length of the diagonal of a <code>1×1</code> square. Tip: <code>zip()</code> again helps pair up coordinates, and raising the summed squares to the power of <code>0.5</code> gives the square root you need.	Skills: loops, squaring, square roots, validation.

Function	Description	Problem Rationale
<code>extract_by_indices(items, indices)</code>	Receives items, a list of any items, and indices, a list of whole numbers. For each index in indices, look up the item at that position in items. If every index is valid, return the list of matching items, in the same order as indices. If even one index is out of range for the items list, return False instead of a list. Example: <code>extract_by_indices(['dog','cat',1,10], [0,2,10])</code> should return False, since index 10 is out of range for a 4-item list. Tip: the built-in <code>len()</code> function tells you the valid index range for items, before you try to look anything up.	Every index must be validated before returning anything — a single out-of-range index invalidates the whole result. Skills: list indexing, validation, guard clauses.
<code>grade_average(scores, drop_lowest)</code>	Receives scores, a list of numbers, and drop_lowest, a whole number. Remove the drop_lowest lowest scores from the list, then return the average of what remains. Example: <code>grade_average([60, 75, 90, 40], 1)</code> should return 75.0, since dropping the lowest score (40) leaves [60, 75, 90], which averages to 75.0. Tip: sorting a copy of scores first makes it easy to identify and remove the lowest values before averaging the rest.	Combines sorting, removal and averaging into one function — a good check on whether students can sequence several list operations correctly. Skills: sorting, list slicing, averaging.
<code>has_two_distinct_values(a_list)</code>	Receives a_list, a list of items of any type. Count how many different (distinct) values appear in the list, ignoring repeats. Return True if there are one or two distinct values in total (i.e. no more than two), and False if there are three or more. Example: <code>has_two_distinct_values([2, 2, 2, 2, 2, 2])</code> should return True, since the list contains only one distinct value, 2, which is within the 'at most two' limit. Tip: the built-in <code>set()</code> function collects the distinct values in a list for you in one step — checking its length then tells you how many there are.	Counting distinct values in a list — conceptually close to using a set for membership checks. Skills: sets, counting distinct values.

Function	Description	Problem Rationale
<code>init_board(rows, cols, row, col, symbol)</code>	Receives rows and cols, two whole numbers giving the size of a 2D grid to build, plus row, col and symbol, describing one cell to mark. Build a grid of that size filled with the character "&", then place symbol at position (row, col) in the grid, and return the finished grid as a list of lists. If (row, col) falls outside the grid, instead return a grid of the same size filled entirely with "#" and leave it unmarked. Example: <code>init_board(2, 3, 1, 2, 'X')</code> should return <code>[['&','&','&'], ['&','&','X']]</code>. Tip: nested for loops — one for rows, one for columns — build the grid; check the bounds with a single if before you start.	Nested loops to build the grid, plus a bounds check that changes the fill character for the whole board. Skills: nested loops, 2D grid construction, bounds checking.
<code>is_comma_separated_integers(digits)</code>	Receives digits, a single string containing several values separated by commas (for example <code>'1, 2, 5'</code>). Split the string on its commas, then check whether every resulting piece can be converted into a whole number. Return True if they all can, or False if even one piece fails to convert. Example: <code>is_comma_separated_integers('1, 2, 5, 10, 5')</code> should return True, since every comma-separated piece is a valid whole number. Tip: the string method <code>.split(',')</code> breaks the string apart at each comma; wrapping a conversion attempt in try/except is one way to test whether a piece is a valid integer.	Splitting a string on commas and validating every resulting token — one bad token should fail the whole check. Skills: string splitting, type conversion, validation, exception handling.
<code>is_prime(n)</code>	Receives n, a whole number. Return True if it's a prime number (greater than 1, with no divisors other than 1 and itself), or False otherwise. Example: <code>is_prime(17)</code> should return True, since no whole number between 2 and 16 divides evenly into 17. Tip: looping from 2 up to n - 1 and checking n % candidate for each one lets you rule out anything that isn't prime.	The first problem to require ruling out every possibility rather than confirming one — a genuinely different kind of loop than the accumulator pattern used so far. Skills: loops, modulo arithmetic, conditionals.

Function	Description	Problem Rationale
<code>longest_word(sentence)</code>	Receives sentence, a string of one or more words separated by spaces. Find and return the longest word in the sentence. If there's a tie for longest, return the first one that appears. Example: <code>longest_word("the quick brown fox")</code> should return "quick", since 'quick' and 'brown' are tied at five letters, and 'quick' appears first. Tip: <code>.split()</code> breaks the sentence into a list of words, and <code>len()</code> on each word lets you track the longest one seen so far as you loop.	Combines string splitting with the running-maximum pattern — tracking the best candidate seen so far while scanning. Skills: string splitting, loops, comparison.
<code>manhattan_distance(p, q)</code>	Receives p and q, two lists of numbers representing points in space with the same number of dimensions (they don't have to be 2D). Return the Manhattan distance between them — the sum of the absolute differences between corresponding coordinates. If p and q are different lengths, return None instead. Example: <code>manhattan_distance([1, 2, 3], [4, 5, 6])</code> should return 9, since $1-4 + 2-5 + 3-6 = 3+3+3 = 9$. Tip: the built-in <code>len()</code>, <code>abs()</code> and <code>zip()</code> functions are all worth considering here — <code>zip()</code> in particular lets you pair up corresponding coordinates from p and q.	Generalizing to arbitrary dimensions (not just 2D) needs a loop instead of a fixed formula. Skills: loops, absolute value, generalised dimensions, validation.
<code>matrix_transpose(matrix)</code>	Receives matrix, a list of lists representing a 2D grid of numbers. Build and return a new grid where the rows and columns are swapped — the first column of matrix becomes the first row of the result, and so on. Example: <code>matrix_transpose([[1, 2, 3], [4, 5, 6]])</code> should return <code>[[1, 4], [2, 5], [3, 6]]</code>. Tip: a nested loop — one over each column index, and inside it one over each row — lets you build each new row by collecting <code>matrix[row][column]</code> for every row, one column at a time.	A genuine 2D restructuring problem, requiring careful row/column indexing throughout. Skills: nested loops, 2D indexing, list building.

Function	Description	Problem Rationale
<code>merge_sorted_lists(list_a, list_b)</code>	Receives list_a and list_b, two lists of numbers that are each already sorted in ascending order. Merge them into a single sorted list containing every value from both, without using Python's built-in sorted() function. Example: <code>merge_sorted_lists([1, 4, 7], [2, 3, 8])</code> should return <code>[1, 2, 3, 4, 7, 8]</code>. Tip: keep one index into each list, always taking whichever of the two current values is smaller and moving that index forward — once one list runs out, add everything left in the other.	The classic two-pointer merge technique — a genuinely different loop structure from anything earlier, tracking two positions in two lists at once. Skills: index arithmetic, two-list traversal, comparison.
<code>middle(a_list)</code>	Receives a_list, a list of any items. If the list has an odd number of elements, return the single middle element on its own. If it has an even number of elements, return the two middle elements together as a list. Example: <code>middle(['hello', 12, 'goodbye', 90])</code> should return <code>[12, 'goodbye']</code>, since the list has 4 (an even number of) elements, and 12 and 'goodbye' are the two middle ones. Tip: the built-in len() function, combined with the % operator to check odd versus even, tells you which case you're in.	Two genuinely different return shapes depending on list length — needs a length-parity check before indexing. Skills: <code>len()</code> , parity checking, list indexing, conditional branching.
<code>powered_sum(numbers)</code>	Receives numbers, a list of numbers. Add together every element except the last one, raise that sum to the power of the last element, and return the result. Example: <code>powered_sum([1, 2, 3, 6, 10])</code> should return 61917364224, since 1+2+3+6=12, and 12 raised to the power of 10 is 61,917,364,224. Tip: list slicing (e.g. <code>numbers[:-1]</code> and <code>numbers[-1]</code>) is a clean way to separate the last element from the rest.	Combines list slicing, a summing loop, and exponentiation in one function. Skills: list slicing, accumulator pattern, exponentiation.

Function	Description	Problem Rationale
<code>rule_30(left, centre, right)</code>	Receives three True/False values: left, centre and right, representing three neighbouring cells in Wolfram's 'Rule 30' cellular automaton. Look up and return the correct next-state value for the centre cell, according to the standard Rule 30 truth table. Example: <code>rule_30(True, True, True)</code> should return False. Tip: writing out all 8 combinations as if/elif conditions, matching the truth table row by row, is the most direct approach.	An 8-case enumeration challenge, with an unfamiliar domain layered on top. Skills: boolean logic, truth tables, conditionals.
<code>spread_fire(neighbourhood)</code>	Receives neighbourhood, a list of three True/False values representing whether the left, centre and right cells in a row are currently 'on fire'. Return True if exactly one of the two outer cells (left or right, but not both) is currently burning — meaning the centre cell catches fire next — and return False otherwise. The centre cell's own current state doesn't affect the answer. Example: <code>spread_fire([True, False, False])</code> should return True, since exactly one neighbour (the left one) is burning. Tip: Python's != (not equal) operator between the two outer values can check 'exactly one is True' in a single comparison.	A small-input enumeration puzzle, but the 'exactly one' condition needs care. Skills: boolean logic, comparison operators.
<code>ticket_price(age, weekend)</code>	Receives age, a whole number, and weekend, a True/False value. Work out the base ticket price from age: £5 for under 12, £10 for ages 12 to 64 inclusive, or £6 for 65 and over. If weekend is True, add a further £2 surcharge. Return the final price. Example: <code>ticket_price(30, True)</code> should return 12, since a 30-year-old pays the £10 standard price plus a £2 weekend surcharge. Tip: work out the base price first with if/elif/else, then add the surcharge afterwards with a separate if.	Combines age-band branching with an additive modifier layered on top. Skills: conditionals, additive logic.

Function	Description	Problem Rationale
<code>word_number_power(word, power)</code>	Receives word, a number spelled out as text (one of "one", "two", "three", "eight" or "nine"), and power, any whole number. Convert word to its numeric value, raise it to the given power, and return the result. If word isn't one of the five allowed words, return the Python keyword None instead. Example: <code>word_number_power("three", 2)</code> should return 9, since 3 squared is 9. Tip: a chain of if / elif statements comparing word against each allowed string is a straightforward way to do the word-to-number mapping.	Requires mapping words to numbers without a dictionary, plus validating two separate inputs. Skills: conditionals, string comparison, exponentiation, input validation.

STRETCH

Stretch (12 problems)*The most demanding problems in the set.*

Function	Description	Problem Rationale
<code>binary_multiply(list_a, list_b)</code>	Receives list_a and list_b, two lists containing only 0s and 1s, of the same length, representing binary (base-2) numbers. Convert each list into its decimal (ordinary base-10) value, multiply the two decimal values together, and return the result. If the two lists aren't the same length, return -1 instead. Example: <code>binary_multiply([1, 1, 1], [1, 1, 1])</code> should return 49, since the binary number 111 equals 7 in decimal, and $7 \times 7 = 49$. Tip: each binary digit's value depends on its position (worth 2 raised to a power based on how far it is from the right-hand end) — build up the decimal total with a for loop over the digits.	Requires understanding positional/binary place value, not just list iteration — the trickiest concept in the whole set. Skills: positional/binary place value, loops, exponentiation.
<code>char_counts_balanced(a_string, a, b)</code>	Receives a_string, a string, and a and b, two sets of characters to compare (case sensitive). Count how many characters in a_string belong to set a, and how many belong to set b, and return True if these two counts are equal, or False otherwise. Example: <code>char_counts_balanced("101578001", "0", "1")</code> should return True. Tip: the string method <code>.count()</code> counts how many times a character appears in a string (e.g. <code>a_string.count('0')</code>).	Counting occurrences of two separate character sets and comparing them. Skills: character counting, comparison.
<code>distance(p, q, metric)</code>	Receives p and q, two points (as lists of numbers), and metric, a string that is either "euclid" or "manhattan". Depending on the value of metric, calculate and return either the Euclidean or the Manhattan distance between p and q. If metric is anything else, or if p and q have different lengths, return None instead.	A function that selects between two different distance formulas based on an input flag, plus extra validation.

Function	Description	Problem Rationale
<code>game_of_life_step(grid)</code>	Receives grid, a list of lists of True/False values representing one generation of Conway's Game of Life, where True means a cell is alive. Work out the next generation, following the standard rules: a live cell with fewer than 2 or more than 3 live neighbours dies; a dead cell with exactly 3 live neighbours becomes alive; every other cell stays as it is. Return the new grid as a list of lists, the same size as grid. Consider only the 8 neighbours immediately surrounding a cell (up, down, left, right and the four diagonals), and treat anything outside the edge of the grid as dead. Tip: build the new grid as a completely separate list of lists rather than editing grid in place — otherwise cells you update early will affect the neighbour-counts of cells you check later in the same generation.	A genuine step up in 2D complexity — 8 neighbours to check per cell, plus edge-of-grid handling and computing the whole new state at once. Skills: nested loops, 2D neighbour counting, edge-case handling.
<code>hamming_distance(word_a, word_b)</code>	Receives word_a and word_b, two strings of the same length. Count the number of positions at which the two strings have different characters, and return that count. Example: <code>hamming_distance("001", "110")</code> should return 3, since the strings differ at three character positions. Tip: looping with <code>range(len(word_a))</code> lets you compare <code>word_a[index]</code> and <code>word_b[index]</code> at each position and count the mismatches.	A position-by-position comparison problem — matching characters between two strings of equal length. Skills: loops, indexing, comparison.
<code>is_overlap(circle_a, circle_b)</code>	Receives circle_a and circle_b, two lists of the form <code>[x, y, r]</code> describing a circle's centre coordinates and radius. Work out the distance between the two centres, and compare it to the sum of the two radii: if the circles touch or overlap, return True; if there's a gap between them, return False. Example: <code>is_overlap([0, 0, 5], [6, 0, 5])</code> should return True, since the distance between the centres (6) is less than the sum of the radii (10). Tip: <code>euclidean_distance()</code> from earlier in this workbook can be reused here directly, if you've already written it.	Combines a distance calculation with a comparison against the sum of two radii — a good capstone for the geometry set. Skills: distance formulas, comparison.

Function	Description	Problem Rationale
<code>is_strong_password(password)</code>	Receives password, a string. Check it against five separate rules: it must be at least 10 characters long, contain at least two uppercase letters, contain at least two lowercase letters, contain at least one digit, and contain at least one special character from the set <code>!@#\$%^&*()</code>. Return True only if all five conditions are met, or False if even one is not. Tip: the string methods <code>.isupper()</code>, <code>.islower()</code> and <code>.isdigit()</code>, checked one character at a time in a for loop, can help you count each category.	Five separate conditions must all be tracked and satisfied at once — the most involved validation problem in the set. Skills: string methods, multi-condition validation.
<code>matrix_multiply(matrix_a, matrix_b)</code>	Receives matrix_a and matrix_b, two lists of lists representing 2D grids of numbers, where the number of columns in matrix_a matches the number of rows in matrix_b. Return their matrix product as a new list of lists, following the standard rule: each value in the result is the sum of pairwise products between a row of matrix_a and a column of matrix_b. Example: <code>matrix_multiply([[1, 2], [3, 4]], [[5, 6], [7, 8]])</code> should return <code>[[19, 22], [43, 50]]</code>, since the top-left value is $(1 \times 5) + (2 \times 7) = 19$. Tip: three nested loops are the standard approach here — one over the rows of matrix_a, one over the columns of matrix_b, and an innermost one that sums the pairwise products for that row/column combination.	The most demanding indexing problem in the set — three nested loops and a genuine mathematical operation over two 2D grids. Skills: triple-nested loops, 2D indexing, accumulator pattern.
<code>run_length_decode(text)</code>	Receives text, a string produced by <code>run_length_encode()</code> earlier in this workbook — a sequence of characters each followed by a run length. Reverse the process: return the original, uncompressed string. Example: <code>run_length_decode("a3b2c1")</code> should return <code>"aaabbc"</code>, since 'a' repeats 3 times, 'b' repeats 2 times, and 'c' appears once. Tip: reading two characters at a time isn't quite right if a run length has more than one digit — instead, read the letter, then keep reading digits until you hit the next letter, converting what you've read with <code>int()</code>.	Reconstructs a string from a compressed run-length format — the multi-digit run-length case makes it slightly trickier than it first looks. Skills: string parsing, type conversion, loops.

Function	Description	Problem Rationale
<code>run_length_encode(text)</code>	Receives text, a string. Scan through it and replace every run of consecutive repeated characters with that character followed by how many times it repeats in a row, then return the compressed result as a new string. Example: <code>run_length_encode("aaabbc")</code> should return <code>"a3b2c1"</code>, since 'a' repeats 3 times, 'b' repeats 2 times, and 'c' appears once. Tip: keep track of the 'current character' and a running count as you loop through text with a for loop, and don't forget to flush the final group once the loop ends.	Run-length encoding — tracking a 'current character' and its count while scanning, then correctly flushing the final group. The highest-marked problem in its source for a reason. Skills: loops, state tracking, string building.
<code>smooth(lst)</code>	Receives lst, a list of numbers. Build a new list where each value is replaced by the average of itself and its neighbours: the first and last elements are averaged with just their one neighbour, and every other element is averaged with both its left and right neighbours. Return the new, 'smoothed' list. Example: <code>smooth([1, 2, 3, 4, 5])</code> should return <code>[1.5, 2.0, 3.0, 4.0, 4.5]</code>, since the first element becomes the average of 1 and 2, the second becomes the average of 1, 2 and 3, and so on. Tip: the built-in <code>len()</code> function helps you spot the first and last indices, which need different handling from the rest.	The edges need different handling from the middle of the list — a common source of off-by-one bugs. Skills: loops, edge-case handling, averaging.
<code>sudoku_row_valid(row)</code>	Receives row, a list of 9 numbers representing one row of a Sudoku grid (each number from 1 to 9, or 0 for an empty cell). Return True if every non-zero number in the row is unique — no digit from 1–9 appears more than once — and False if any digit repeats. Empty cells (0s) don't count as duplicates of each other. Example: <code>sudoku_row_valid([5, 3, 0, 0, 7, 0, 0, 0, 0])</code> should return True, since 5, 3 and 7 are all different and the 0s are ignored. Tip: collect the non-zero values into a list, then compare its length to the length of a <code>set()</code> built from the same values — if they're different, something repeated.	A focused, real-world validation rule that makes for a clean set()-based capstone. Skills: sets, filtering, comparison.