

Lists & Loops

Overview: what you'll produce

By the end of this lab you will have a single folder, `code/labs/lab-03/`, containing six working Python scripts, each one runnable on its own from the terminal:

- `lists.py` - creates, indexes and appends to a list, then splits it into two by removing items
- `slicing.py` - slices lists into sub-parts and builds a 2D list
- `loops_for.py` - uses for loops with `range()` and with lists directly
- `graphics.py` - uses nested for loops to build small grids of characters, wrapped in a simple function
- `loops_while.py` - uses while loops, including input validation
- `quiz_scores.py` - a small mini-project that collects and summarises a list of scores

Nothing here is handed in — it's practice, so work at your own pace and don't worry about getting every exercise perfect. Some students will get through this faster than others depending on how much Python experience they already have, and that's expected. If you don't finish everything in the three hours, use your independent study time to complete the remaining exercises.

Before you start

This lab builds on Labs 1 and 2 — you should already have a `code/labs/` folder from last time. Save every script inside a new folder called `code/labs/lab-03/` to keep things organised. Check that:

- ☐ You completed Labs 1 and 2, and `code/labs/lab-01/` and `code/labs/lab-02/` still exist
- ☐ You can open a terminal and VS Code, and remember `cd` to move between folders

Lab objectives

- Create, index and change lists, including negative indexing and `append()`
- Remove items from a list with `remove()` or `pop()` to split one list into two
- Slice a list to pull out a sub-part, and build a simple 2D list
- Use `for` loops with `range()` and directly over a list, including the `_` convention
- Use a nested for loop to build a small grid of characters, and wrap it in a simple function
- Use `while` loops, including incrementing, decrementing, and input validation with `continue` and `break`

Timing guide

Time	Exercise	Focus
0:00 – 0:10	Setup	Create working folder
0:10 – 0:35	Exercise 1	Lists: the solar system
0:35 – 0:55	Exercise 2	Slicing and 2D lists
0:55 – 1:15	Exercise 3	For loops
1:15 – 1:25	Break	—
1:25 – 1:55	Exercise 4	Graphics with nested for loops
1:55 – 2:20	Exercise 5	While loops
2:20 – 2:50	Exercise 6	Mini project: Quiz Score Tracker
2:50 – 3:00	Wrap-up	Self-check, stretch goals

A note on lists and loops

A list is a single variable that holds many values in order, written between square brackets: `["apples", "bread", "milk"]`. Each value has a position, called its index, starting at `0` for the first item — this is called zero-indexing. Lists are also mutable, meaning you can change, add, or remove items after creating them. A loop lets you repeat an action once for every item in a list (or a range of numbers) without writing the same line out by hand — which matters because hand-written repetition breaks the moment your data changes size. This lab pairs the two ideas together: loops are most often used to work through lists.

Exercise 1 — Lists: the solar system (25 min)

Create a new file called `lists.py` for this exercise.

1. Create an empty list, then create a populated list mixing a few planets and moons together on purpose.

```
empty_list = []
solar_system = ["Mercury", "Venus", "Earth", "Moon", "Mars", "Phobos"]
```

2. Print the list, and print how many items it has using `len()`.
3. Print the first item using index `0`, and the last item using index `-1`.

4. Use `append()` to add two more bodies to the end of the list — "Deimos" and "Jupiter" — then print the list again.
5. Look closely — there's a typo in the third item of the list (it should say "Earth"). Work out its index, fix it by assigning directly to that position, then print the list to confirm.
6. Create a mixed-type list describing Earth: its name, its position from the sun (a number), and whether it has rings (a Boolean). Print it — lists can hold more than one type at once.
7. Your `solar_system` list currently mixes planets and moons together — exactly the unstructured data the lecture warned about. Create a new empty list called `moons`. Then remove "Moon" from `solar_system` and append it to `moons`. Syntax hint: `some_list.remove(value)` and `another_list.append(value)`.
8. Repeat the same two operations for "Phobos" and "Deimos", so all three moons end up in `moons` and only planets remain in `solar_system`.
9. Print both lists to confirm the split worked: `solar_system` should contain only planets, and `moons` should contain only moons.

Checkpoint

`lists.py` should build a single list mixing planets and moons, fix a typo using index assignment, add new items with `append()`, and end up with two separate lists — `solar_system` containing only planets, and `moons` containing only moons — produced by removing items from the original list.

Stretch goal: try `list.pop()` instead of `remove()` for one of the moons — `pop()` removes by position instead of by value, and also returns the item it removed, which you can capture in a variable.

Exercise 2 — Slicing and 2D lists (20 min)

Create a new file called `slicing.py` for this exercise.

1. Create a list of six 100m sprint finish times, in seconds. Then use slicing (`list[start:end]`) to print just the middle four.

```
finish_times = [9.58, 9.69, 9.79, 9.84, 9.87, 9.92]
```

2. Use `[ :3 ]` to print the first three times, and `[ 3: ]` to print everything from index 3 onwards.
3. Use `[ -3: ]` to print just the last three times, without counting how many items are in the list.
4. Create two parallel lists — one of runner names, one of matching finish times — where the same position in each list refers to the same runner. Print the name and time of the runner in third place.

```
runners = ["Bolt", "Blake", "Gatlin", "Powell", "Bailey", "Gay"]
times = [9.58, 9.69, 9.79, 9.84, 9.87, 9.92]
```

5. Now build the same information as a single 2D list instead — a list of `[name, time]` pairs — and index into it to print the third-place finisher's full record.

```
race = [["Bolt", 9.58], ["Blake", 9.69], ["Gatlin", 9.79], ["Powell", 9.84],
["Bailey", 9.87], ["Gay", 9.92]]
```

Checkpoint

`slicing.py` should slice a list at least two different ways (for example a middle range and an open-ended slice), and create and index into a 2D list of at least three inner lists.

Stretch goal: build a 3×3 grid as a list of three inner lists (for example a noughts-and-crosses board), then print just the middle row.

Exercise 3 — For loops (20 min)

Create a new file called `loops_for.py` for this exercise.

1. Use a `for` loop with `range(5)` to print the numbers 0 to 4 — these will become the pen numbers on a zoo feeding round. Syntax hint:

```
for value in range(n):
    ...
```

2. Now use `range(1, 6)` together with an f-string to announce each feeding round from 1 to 5, e.g. "Feeding round 1 starting!" through "Feeding round 5 starting!".

3. Print "Restocking food store." exactly three times using a `for` loop where the loop variable is never used — use `_` as its name to signal that on purpose.

4. Loop directly over a list of animal names (not over a range of numbers) and print a feeding reminder for each one.

```
animals = ["tiger", "giraffe", "penguin"]
```

5. Add up a list of food quantities (in kg) using a running total inside a `for` loop, rather than writing out each addition by hand.

```
food_kg = [12, 8, 15, 20, 6]
```

Checkpoint

`loops_for.py` should use `range()` at least twice (once starting from 0, once starting from a different number), loop directly over a list, use `_` once for a loop variable that's never used, and total a list of numbers using a running total.

Stretch goal: predict what `range(2, 10, 2)` would print in a `for` loop, then try it and check your answer.

Exercise 4 — For loops: building graphics (30 min)

Create a new file called `graphics.py` for this exercise. For loops aren't just for numbers and lists — they're also how you build up repeated visual patterns, character by character. This exercise starts with plain hardcoded output and works up to a small function that can draw grids of any size.

1. Print a single row of five stars by typing the string directly — no loop needed yet: `print("*****")`.
2. Now print a block three rows tall and five stars wide, using three separate `print()` calls, one per row. Notice how repetitive this is, and how easily a typo could sneak in — this is exactly the kind of repetition loops exist to remove.
3. Produce that same 3×5 block of stars using a nested `for` loop instead: an outer loop for each row, and an inner loop that builds up one row of stars using string concatenation, printing the row once it's complete. Syntax hint:

```
for row in range(rows):
    line = ""
    for col in range(cols):
        line += "*"
    print(line)
```

4. You'll meet functions properly in the next lab, but here's an early taste: wrap your nested loop in a function called `print_grid(rows, cols)`, so you can print a grid of any size just by calling it with two different numbers, instead of rewriting the loops each time. Syntax hint:

```
def print_grid(rows, cols):
    ...

print_grid(3, 5)
print_grid(2, 8)
```

Checkpoint

`graphics.py` should print a hardcoded 3×5 block of stars, reproduce it using a nested `for` loop, and wrap that loop in a `print_grid(rows, cols)` function that successfully prints at least two different grid sizes.

Stretch goal: draw an empty chess board — 8 squares by 8 squares — where each square is itself a block of characters rather than a single character, e.g. a `size × size` block of `"#"` for a dark square and `" "` for a light square, using a function `draw_board(size)`. You'll need a loop for the board's rows, a loop for its columns inside that, and then a further pair of nested loops to print each square's block of characters; whether a square is light or dark alternates based on whether `(row + col)` is even or odd, which you can check with the `%` operator from Lab 2.

Exercise 5 — While loops (25 min)

Create a new file called `loops_while.py` for this exercise.

1. Use a `while` loop to print the numbers 1 to 5, incrementing a counter by 1 each time round. Syntax hint:

```
count = start_value
while condition:
    ...
```

2. Now count down from 5 to 1 with a decrementing `while` loop, printing "Liftoff!" once the loop finishes.
3. Deliberately create an infinite loop by writing a `while` loop that forgets to update its counter. Run it, watch it print forever, then press `Ctrl+C` in the terminal to stop it — this returns control to you when a program won't.

```
count = 1
while count <= 5:
    print(count)
```

4. Delete that broken loop (or fix it by adding the missing update back in) before moving on.
5. Use `while True` together with `continue` and `break` to keep asking the user for an order number until they type one made only of digits, using `str.isdigit()` to check. Syntax hint:

```
while True:
    answer = input(...)
    if not_valid:
        continue
    break
```

Checkpoint

`loops_while.py` should count up with an incrementing `while` loop, count down with a decrementing `while` loop, and validate user input using `while True` with `continue` and `break`.

Stretch goal: extend the validation loop so it also rejects an empty input ("") with its own message.

Exercise 6 — Mini project: Quiz Score Tracker (30 min)

Bring together lists and loops from this lab into one program, `quiz_scores.py`, that collects a set of quiz scores and summarises them.

Requirements:

- Use a `while True` loop to repeatedly ask "Enter a score (or -1 to finish): ", converting each answer to an `int`
- Use `break` to stop the loop when the user enters -1, otherwise `append()` the score to a list
- Once collection has finished, use a `for` loop with `range(len(scores))` to print each score labelled with its position, e.g. "Score 1: 72"
- Use a running total inside a for loop to calculate the total and the average of the scores
- Use a for loop to work out the highest and the lowest score, by keeping track of the best (and worst) seen so far
- Organise the program's logic inside a `main()` function, called at the bottom of the file

Example run:

```
Enter a score (or -1 to finish): 72
Enter a score (or -1 to finish): 88
Enter a score (or -1 to finish): 65
Enter a score (or -1 to finish): -1
Score 1: 72
Score 2: 88
Score 3: 65
Average: 75.0
Highest: 88
Lowest: 65
```

Checkpoint

Running `quiz_scores.py` should collect scores with a `while True` loop until -1 is entered, list them with a for loop and their position number, and correctly compute the total, average, highest and lowest score, all organised inside `main()`.

Stretch goals (if time remains)

- Print the scores in reverse order using slicing (`scores[::-1]`)
- Count how many scores were 70 or above, using a counter variable updated inside a for loop
- Ask the user up front how many scores they plan to enter, then collect them with a `for` loop over `range()` instead of `while True`

Self-check

Before you finish up, look back over what you've written — this isn't handed in, but it's worth confirming you've hit the key ideas:

- ☐ `code/labs/lab-03/` contains: `lists.py`, `slicing.py`, `loops_for.py`, `graphics.py`, `loops_while.py`, `quiz_scores.py`

- ☐ Every script runs from the terminal without errors: `python <file>.py`
- ☐ `solar_system` and `moons` end up correctly split, with no moons left in `solar_system`
- ☐ `graphics.py`'s `print_grid(rows, cols)` function successfully prints at least two different sized grids
- ☐ At least one `for` loop and one `while` loop each appear, working correctly
- ☐ `quiz_scores.py` collects scores until `-1`, then reports total, average, highest and lowest, inside `main()`