

Packages & Testing

Overview: what you'll produce

By the end of this lab you will have a single folder, `code/labs/lab-05/`, containing six working files, each runnable (or, for the test file, discoverable by `pytest`) from the terminal:

- `random_practice.py` - uses the built-in `random` module to simulate dice rolls and coin tosses
- `namespaces.py` - compares import styles and shows how naming collisions can happen
- `third_party.py` - uses a package installed from PyPI with `pip`
- `assertions.py` - replaces manual print-and-check testing with `assert` statements
- `calculator.py` - a small module of arithmetic functions to be tested
- `test_calculator.py` - a `pytest` test file that checks `calculator.py` automatically

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 to 4 — you should already have a `code/labs/` folder from last time. Save every file inside a new folder called `code/labs/lab-05/` to keep things organised. Check that:

- ☐ You completed Labs 1 to 4, and their folders still exist inside `code/labs/`
- ☐ Your machine has a working internet connection — this lab installs packages with `pip`, which needs one

Lab objectives

- Use a built-in package (`random`) via the `import` keyword
- Compare import styles, and understand how naming collisions can happen
- Install and use a third-party package from PyPI with `pip`
- Replace manual, by-eye testing with `assert` statements
- Write and run automated tests with `pytest`, including checking that an error is correctly raised

Timing guide

Time	Exercise	Focus
0:00 – 0:10	Setup	Create working folder
0:10 – 0:35	Exercise 1	The random module
0:35 – 0:55	Exercise 2	Namespaces and import styles
0:55 – 1:10	Exercise 3	Installing packages with pip
1:10 – 1:20	Break	—
1:20 – 1:50	Exercise 4	Assertions
1:50 – 2:45	Exercise 5	Mini project: testing with pytest
2:45 – 3:00	Wrap-up	Self-check, stretch goals

A note on packages

A package is a collection of code that someone else has already written, ready for you to reuse instead of writing it yourself. Python's core installation is deliberately light — it covers general-purpose programming, but not every specialist need — so extra functionality comes from packages. Some ship with Python already (the `random` module you'll use in Exercise 1 is one of these); others live on the Python Package Index (PyPI, at pypi.org) and are installed onto your machine with `pip`, Python's package manager.

Exercise 1 — The random module (25 min)

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

1. Import the `random` module, then use `random.randint(1, 6)` to simulate a dice roll and print the result. Run the script a few times and notice the result changes. Syntax hint:

```
value = random.randint(low, high)
```

2. Use `random.choice()` on a list of at least four fruit names to print one chosen at random.
3. Simulate a coin toss using `random.randint(0, 1)`, treating `0` as heads and `1` as tails, and print which one it was.
4. Now simulate 10 coin tosses in a for loop, counting how many were heads and how many were tails, and print both totals at the end.

Checkpoint

`random_practice.py` should use `random.randint()` and `random.choice()` at least once each, and simulate 10 coin tosses using a loop, counting heads and tails.

Stretch goal: use `random.choice()` to pick from a list of at least five short "fortune" messages and print one each time the script runs.

Exercise 2 — Namespaces and import styles (20 min)

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

1. Import the whole module with `import random`, and call `random.choice()` using the module prefix, on a short list of your own.
2. Now import just the function instead, and call it directly without the `random.` prefix. Syntax hint:

```
from module_name import function_name
```
3. Below that, define your own function also called `choice()` that just returns a fixed string, then call `choice()` again and see which one actually runs.
4. Your own `choice()` silently overwrote the one imported from `random` — this is a naming collision. Using `import random` and always writing `random.choice()` avoids this, because your own code and the module's code live in separate namespaces.

Checkpoint

`namespaces.py` should call `random.choice()` using the module-prefixed style, then again via `from random import choice`, then demonstrate — by defining your own function with the same name — how that style can silently collide with your own code.

Stretch goal: try `import random as rnd` and call `rnd.randint(1, 10)` — a third style, useful when a module's real name is long or clashes with something else.

Exercise 3 — Installing packages with pip (15 min)

Create a new file called `third_party.py` for this exercise. This one needs a package that doesn't come with Python, installed from PyPI.

1. In the terminal (not in VS Code's editor), make sure you're inside `code/labs/lab-05/` and run:

```
pip install cowsay
```

2. This downloads `cowsay` from pypi.org and installs it onto your machine — the same process works for any of the hundreds of thousands of packages published there. If this step fails, check your internet connection or ask a demonstrator.

3. In `third_party.py`, import `cowsay` and use it to print a message — the function you want is `cowsay.cow(message)`.

4. Run the script and check the output appears correctly in the terminal.

Checkpoint

`third_party.py` should successfully import and use a package that was installed with `pip` rather than being part of the core Python distribution.

Stretch goal: find a second small package on pypi.org, install it with `pip`, and use one of its functions in the same file.

— 10 minute break —

A note on testing

So far, every exercise in this course has been checked manually — you ran your code, looked at the output, and judged for yourself whether it was right. That works, but it's tedious, easy to get wrong, and has to be repeated in full every single time the code changes. Testing replaces that manual judgement with code that checks itself: you write down what you **expect** a function to do, compare it automatically to what it **actually** does, and let the computer decide pass or fail — every time, without you watching.

Exercise 4 — Assertions (30 min)

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

1. Write a simple function `add(a, b)` that returns the sum of its two arguments.
2. Test it the manual way first: `print(add(2, 3))` and read the result yourself to check it's 5. Notice you have to check every result by eye, and remember what each one was supposed to be.
3. Now replace that with an `assert` statement instead. Run it — no output at all means the assertion held. Syntax hint:

```
assert expression == expected_value
```

4. Add a few more assert statements covering different cases: negative numbers, and zero.
5. Deliberately write a wrong assertion, for example `assert add(2, 3) == 6`, run it, and read the `AssertionError` — it points at the exact line that failed. Then correct it back to 5.

Checkpoint

`assertions.py` should replace manual print-and-check testing with several `assert` statements covering different cases of a function, including one you deliberately made fail so you could read the `AssertionError` before fixing it.

Stretch goal: give one of your assertions a custom message, e.g. `assert add(2, 3) == 5, "add() should sum two numbers"`, and see that message appear if it ever fails.

Exercise 5 — Mini project: testing with pytest (55 min)

Assertions are the building block, but `pytest` automates finding, running and reporting every test for you. Bring this together across two files.

Requirements:

- Install `pytest` in the terminal: `pip install pytest`
- Create `calculator.py` containing three plain functions: `add(a, b)`, `subtract(a, b)`, and `divide(a, b)` — `divide` should raise a `ValueError` if asked to divide by zero
- Create `test_calculator.py` in the same folder, that imports from `calculator` and defines several small, descriptively-named test functions, each prefixed `test_` and each containing exactly one `assert`
- Include one test that uses `pytest.raises(ValueError)` to check that `divide(10, 0)` correctly raises an error
- `test_calculator.py` should contain no `main()`, no `if __name__ == "__main__":`, and no `try/except` — `pytest` handles running and reporting for you

This exercise introduces two genuinely new pieces of syntax, so here are hints for both — not the calculator tests themselves, which you need to write.

Syntax hint for a single test:

```
def test_description():  
    assert expression == expected_value
```

Syntax hint for checking that an error is correctly raised — `with` opens a block that's expected to raise the named error:

```
def test_error_case():  
    with pytest.raises(SomeError):  
        call_that_should_raise(...)
```

Example run, once your tests exist and `calculator.py` is correct:

```
===== test session starts =====  
collected 5 items  
  
test_calculator.py ..... [100%]  
  
===== 5 passed in 0.01s =====
```

1. In the terminal, from inside `code/labs/lab-05/`, run `pytest` and read the output — a green dot for each passing test.
2. Deliberately break `calculator.py` — for example, make `add` return `a - b` by mistake — and run `pytest` again. Read the red `F` and the traceback, which points to the exact assert that failed and shows the expected vs. actual value. Then fix it back.

Checkpoint

`test_calculator.py` should contain several small, descriptively-named test functions — no `main()`, no `if __name__`, no `try/except` — including one using `pytest.raises` to check a `ValueError` is correctly raised. Running `pytest` should show all five tests passing.

Stretch goals (if time remains)

- Create a `tests/` subfolder with an empty `__init__.py` file, move `test_calculator.py` into it, and run `pytest` from `lab-05/` to see it still find and run the tests automatically
- Add a further test for `subtract` using negative numbers, following the same naming convention as the others
- Introduce a second deliberate bug in `calculator.py` and confirm `pytest`'s output pinpoints it correctly before fixing it back

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-05/` contains: `random_practice.py`, `namespaces.py`, `third_party.py`, `assertions.py`, `calculator.py`, `test_calculator.py`
- ☐ `pip install pytest` completed successfully, and `pytest` runs from the terminal
- ☐ `test_calculator.py` contains no `main()`, no `if __name__`, and no `try/except`
- ☐ At least one test uses `pytest.raises` to check that an error is correctly raised
- ☐ Running `pytest` on `lab-05/` shows all tests passing