# Software Engineering I

Raul P. Pelaez

February 11, 2026

## Contents



## 1 Introduction

In this lesson, we will learn about some of the most important practices in software engineering, mainly:

- Testing your code
- Documenting your code

Info

**Software engineering**
Software engineering is the application of engineering to the design, development, implementation, testing, and maintenance of software in a systematic method. In simple terms, its about all the things related to programming that are not writing code per se.

Advice

**Writing code is the easy part**
The hard part of producing software is making sure that the code is correct, maintainable, and scalable. Modern tools can mostly write code for you, but struggle with putting together all the other elements.
At some point, you will have to produce *usable* software, be it for your own applications, for a client or for the community. Knowing how to do so in a clean and standard way will give you in an invaluable advantage.
On the other hand, even with the high level nature of Python, your projects, if they are deemed useful, will inevitably grow in complexity. Your projects will be composed of many interleaved components providing different functionality. When adding a new feature, you will have to make sure that you do not break the existing functionality. Automated testing will give you the confidence to make changes without breaking things.

# 2 Testing your code

It is hard to predict all the ways in which your code can fail when designing it. Perhaps you wrote a function that takes a username as input, but you did not consider someone passing a sequence of emojis to it, or a username with a length of 1000 characters. Oversights like these can lead to your code crashing in unexpected ways, or worse, to security vulnerabilities. And have in fact been the cause of many high-profile security breaches[1].

Testing your code serves two purposes:

- It gives you and the users of your code the certainty that your code works as expected.
- It helps you to design and develop your code.

Today, I want to introduce you to two concepts that will help you to write better code: Unit testing and Test-Driven development.

Info

**Unit tests**
Unit testing is the practice of testing individual units or components of a program to ensure they work as expected. Each unit, typically a function or method, is tested in isolation from the rest of the codebase. Unit tests are automated and are used to validate that the logic of each unit is correct, handle edge cases, and catch bugs early. By running unit tests frequently, developers can maintain code reliability, make refactoring safer, and quickly identify problems introduced during development. Popular testing frameworks like `pytest` in Python help simplify writing and running tests.

Info

**Test-Driven development (TDD)**
TDD is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

Advice

We do not have to apply this technique in a dogmatic way. However, the principles behind it are sound when applied in a flexible way. The main idea is to try to have tests for the important functionality in your code, which will help you to design it and to keep it working at every step of the way.

## 2.1 Pytest

There are several frameworks that deal with unit testing in Python. The most popular one is `pytest`, which makes it easy to write simple tests, but also scales well to highly complex scenarios.

`pytest` is both a python module that you can import with a lot of testing goodies and also a standalone command-line tool that you can use to run your tests.

Info

**The pytest command**
In essence, the `pytest` command will look for files that start with `test_` or end with `_test.py` and run the functions with names that start with `test_` inside them.

A typical workflow will consist of creating a file (or files, depending on the size of the project) called `test_something.py` (replacing `something` with a relevant name) and write inside functions with names

[1] This attack vector is called an "input validation attack", and can be abused to take advantage of things like buffer overflows, SQL injection, and many other types of attacks.

starting with `test_` that will test the functions in the main codebase.

For instance, write the following code in a file called `test_example.py`:

```
def test_example():
    assert 1 == 1
```

Now, run the following command in the terminal:

```
pytest -v
```

The `-v` flag is optional, it will give you a more verbose output. You should see an output similar to this:

```
====================== test session starts ======================
collected 1 item

test_example.py::test_example PASSED                          [100%]

======================= 1 passed in 0.01s =======================
```

Pytest looked inside all the files in the current directory, found the file `test_example.py`, and ran the function `test_example` inside it. The function `test_example` contains an assertion that will always be true, so the test passed.

Lets give it an impossible situation now, i.e. a test that will always fail. Add this to the test script:

```
def test_another():
    assert 1 == 2
```

Run `pytest` again, you will see something like:

```
====================== test session starts ======================
collected 2 items

test_example.py::test_example PASSED                    [ 50%]
test_example.py::test_another FAILED                    [100%]

============================ FAILURES ============================
_________________________ test_another __________________________

    def test_another():
>       assert 2 == 1
E       assert 2 == 1

test_example.py:6: AssertionError
==================== short test summary info ====================
FAILED test_example.py::test_another - assert 2 == 1
================= 1 failed, 1 passed in 0.04s ==================
```

The test `test_another` failed, as expected. The output is very informative, it tells you which test failed, and why it failed. Now imagine that these tests are non trivial, checking for edge cases and the like in a big project. Having this tool allows us to:

- Make sure that every part of the code works as expected at all times
- Make sure that when we add new features or modify existing ones, we do not break the existing ones
- Help us develop our code by writing the tests first, and then the code that makes them pass

Info

**Anatomy of a unit test**

A unit test is a function that tests a specific part of your code. It should be self-contained, and test only one thing. Lets see an example:

Say we have a function that calculates the area of a rectangle:

```
# This is stored in a file called area.py
def calculate_area(width, height):
    return width * height
```

We can write a test for this function like this:

```
# This is stored in a file called test_area.py
from area import calculate_area

def test_calculate_area():
    assert calculate_area(2, 3) == 6
```

This is the typical structure of a test file. We import the function we want to test, and then write a function that tests it. The function name should start with `test_`. Inside the function, we use the `assert` keyword to check the functionality with some known inputs/outputs.

Our function `calculate_area` should return 6 when given 2 and 3 as arguments, so the test should pass. However, what should happen if the function receives a negative number? Ideally, the function should not allow us to do so, raising an exception instead. We can add a test for this as well:

```
import pytest
def test_calculate_area_negative():
      with pytest.raises(ValueError):
          calculate_area(-2, 3)
```

We can test for positive outcomes, but also "negative" ones, i.e. when the function should raise an exception. This test reads as "providing a negative number to the function should raise a ValueError". Try to run it, you should see a failure.

We can now update our function to raise an exception when given a negative number:

```
def calculate_area(width, height):
    if width < 0 or height < 0:
        raise ValueError("Width and height must be positive")
    return width * height
```

Now, the test should pass.

Advanced

**Many tests in one**

Pytest allows to generate many tests with the same function using the `parametrize` decorator. This is useful when you want to test the same function with different inputs. For instance, we can rewrite the tests for the `calculate_area` function like this:

```
import pytest
@pytest.mark.parametrize("width", list(range(30)))
@pytest.mark.parametrize("height", list(range(30)))
def test_calculate_area(width, height):
    expected = width * height
    assert calculate_area(width, height) == expected
```

If you run this test, you will see that it runs 900 tests, one for each combination of width and height. This feature that allows you to test many edge cases with very little code.

# 3 Documenting your code

Python provides several powerful tools for documenting code that will make it easier to use for your users (which includes you in the future). Let's look at two of the most important ones: docstrings and type hints.

Advice

**Automatic documentation**
Check the numpy documentation. Most of it was generated automatically from the docstrings of the functions. There are tools, such as Sphinx, that will inspect your codebase and generate a website with all the documentation for you.

## 3.1 Docstrings

Info

**Docstrings**
A docstring (documentation string) is a string literal that appears as the first statement in a module, function, class, or method in Python. When enclosed in triple quotes, it becomes the special `__doc__` attribute of that object, making it accessible to both developers reading the code and tools that automatically generate documentation.
The most important elements of a docstring are:

1. A one-line summary of what the code does
2. A more detailed description if needed
3. Parameter descriptions including expected types
4. Description of what is returned
5. Any exceptions that might be raised
6. Examples of usage when helpful

Depending on the context, some of the elements may be omitted. For instance, a simple function may only need a one-line summary.

Here's an example of a fully documented function:

```python
def calculate_area(width: float, height: float) -> float:
    """Calculate the area of a rectangle.

    Takes the width and height of a rectangle and returns its area.
    Raises ValueError if either dimension is negative.

    Args:
        width (float): The width of the rectangle
        height (float): The height of the rectangle

    Returns:
        float: The area of the rectangle

    Raises:
        ValueError: If width or height is negative

    Example:
        >>> calculate_area(2.0, 3.0)
        6.0
    """
    if width < 0 or height < 0:
        raise ValueError("Width and height must be positive")
    return width * height
```

Info

**Docstring formats**

The previous docstring is an example of the "Google" format. Being just text, the format is not enforced by Python, but there are some conventions that are widely used. The most common ones are:

- Google: The one used in the example above.
- Numpy: Used in the numpy library.
- reStructuredText: Used by the Sphinx documentation generator.
- Epytext: Used by the Epydoc documentation generator.

For instance, this is a ReST style docstring:

```
def foo(param1, param2):
    """
    This is a reST style.

    :param param1: this is a first param
    :param param2: this is a second param
    :returns: this is a description of what is returned
    :raises keyError: raises an exception
    """
```

Tools that parse docstrings can normally understand all these formats, but you must be careful to adhere to one of them exactly (in terms of things like spaces, number of blank lines, etc). Check this stackoverflow question for more information.

Advice

Depending on the complexity of the function, method, or class being written, a one-line docstring may be perfectly appropriate. These are generally used for really obvious cases, such as:

```
def add(a, b):
    """Add two numbers and return the result."""
    return a + b
```

Just using the signature of the function, code editors will provide information about the expected types of the arguments and the return value. However, for more complex functions, a more detailed docstring is the way to go.

Advanced

**Doctest: test in the documentation**

Python has a built-in module called `doctest` that allows you to write tests in the docstrings of your functions. These tests are run when you run the `doctest` module (although pytest can also run them). Doctests are examples of how to use the function, and they are written in the same way as the Python interactive shell. For instance, the example in the docstring above is a doctest.

Advice

**Copilot is extremely good at writing docstrings**

The formatting of docstrings can be a bit tedious, but nowadays we can first write the code and then let copilot write the documentation. You can go the other way around, write the docstring first, and then let copilot write the code. But in my experience this is less effective.

## 3.2 Type Hints

Info

**Type hints**
Type hints are annotations added to function signatures that indicate the expected types for arguments and return values. While Python remains a dynamically typed language at runtime, type hints serve as a form of documentation and enable static type checking through external tools.
Type hints provide three main benefits:

1. Documentation: They clearly communicate expected types to other developers
2. IDE Support: Enable better code completion and error detection
3. Static Analysis: Tools like mypy can catch type-related bugs before runtime

Type hints use colon notation for parameters and arrow notation for return types:

```python
from typing import List, Dict, Optional

def process_data(items: List[int],
                 config: Optional[Dict[str, str]] = None) -> List[float]:
    """Process a list of integers using an optional configuration.

    Args:
        items: List of integers to process
        config: Optional dictionary with processing parameters

    Returns:
        List of processed values as floats
    Example:
        >>> process_data([1, 2, 3])
        [1.0, 2.0, 3.0]
        >>> process_data([1, 2, 3], {'scale': '0.5'})
        [0.5, 1.0, 1.5]
    """
    # Function implementation
```

Many type hints are built into Python, but you can also use the `typing` module to define more complex types. For instance, `List` and `Dict` are defined in the `typing` module.

Advice

**Type hints are more than documentation**
Type hints are a way to document your code, *hinting* to the reader (of both the code and its documentation) what the expected types are for the arguments and return values of your functions. But they are also a way to catch errors early. For one, your editor will warn you if you are passing the wrong type to a function before you even run the code (this is called static analysis).
Modern Python development increasingly relies on type hints, and many popular libraries and frameworks expect or require them.

# 4 Exercises

## 4.1 Corner cases

**Goal**

Lets write some simple functions and some unit tests for them using `pytest`. Start by installing the `pytest` package. First activate your conda environment:

```
conda activate your_env
```

Then install the package:

```
conda install pytest
```

**hint**

Better yet, if you are keeping an `environment.yml` for this class, add `pytest` there as a dependency and run the following from the same folder that file is at:

```
conda env update
```

**Milestone**

Write a function that returns `True` if the input is a prime number, and `False` otherwise. Place your function in a file called `prime.py`. Your function should have this signature:

```
def is_prime(n):
    # Your logic here
```

**Milestone**

Write some tests for this function in a file called `test_prime.py`. Your test file should have this structure:

```
from prime import is_prime

# Your test functions here
```

Use some known prime numbers and non-prime numbers to test your function. Make sure to test edge cases, like 0 and 1.

**hint**

- A number `a` is divisible by another number `b` if `a % b == 0`.

**Milestone**

Write two new tests:

- One that tests the function with a negative number
- One that tests the function with a float

Your `is_prime` function should only accept integers, so it should raise a `TypeError` if it receives a float or a negative number.

**hint**

- You can use the `pytest.raises` context manager to check if a function raises an exception.

**Advanced milestone**

Write your correctness tests using the `parametrize` decorator.

**Milestone**

Build the following functionality into our function:

- If the input is a string, the function should try to convert it to an integer and use that. If it cannot, it should raise a `ValueError`.

Start by adding a test for this functionality, which will probably fail. Then modify `is_prime` so that the test (and all the others) pass.

## 4.2 How do I use this thing?

**Goal**

Lets add some documentation to our functions from today. We will use docstrings and type hints.

**Milestone**

Add a docstring to your `is_prime` function. It should contain a one-line summary, a more detailed description, the expected types for the arguments and return value, and an example of usage.

**Milestone**

Add type hints to your function.

**hint**

- The function always returns a boolean, but the input can be an integer or a string.
- Use the `Union` type from the `typing` module to indicate that the input can be of two different types.