Lab 2

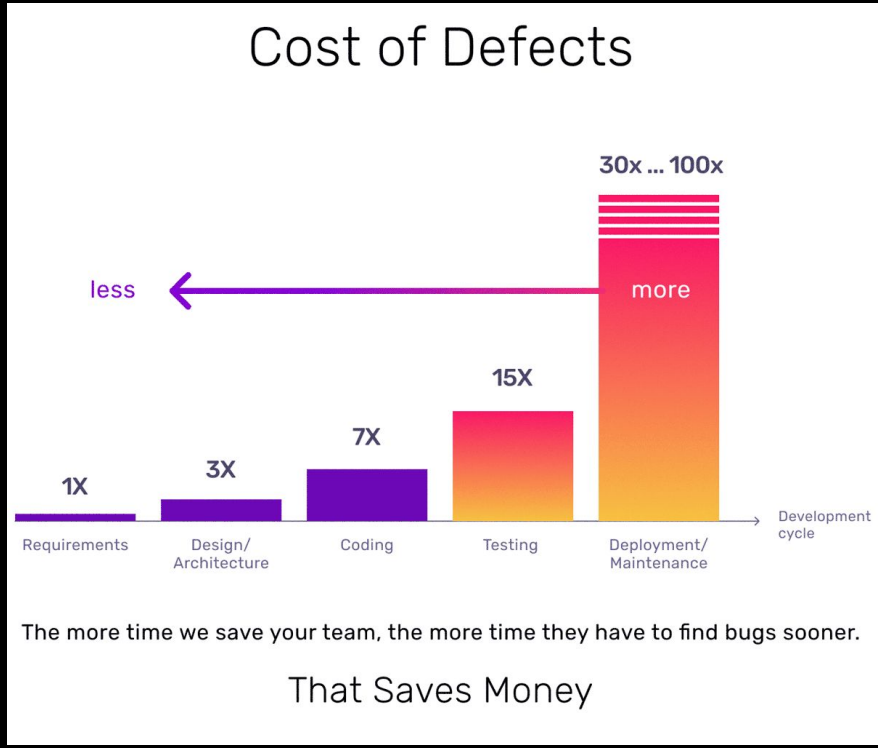
Introduction to **Testing Your Code**

What *is* code testing? (In your own words)

Why Bother Writing Tests?

Why Bother Writing Tests?

- Catches bugs early → cheaper & easier to fix
- Documents expected behaviour — tests are living specifications
- Enables safe refactoring and long-term code health
- Builds confidence for collaboration & continuous deployment
- Automates quality control — no more manual re-checking



What *is* a “unit test”

- Code to verifies one smallest piece of behavior (a “unit”) in isolation
- Runs fast (ms) and deterministically
- Has no external side-effects (DB, network, file system)

Note: this is different from integration & end-to-end tests, which we won’t really cover in this course

Quick Introduction to Testing

1. Simple `assert` statements

[1]
```python
## 1. Simple `assert` statements

# define a tiny function
def add_two(x: int) -> int:
    """Return x + 2."""
    return x + 2

# smoke-test with bare asserts
assert add_two(3) == 5
assert add_two(-2) == 0
print("add_two passed basic asserts")
```

```
add_two passed basic asserts
```

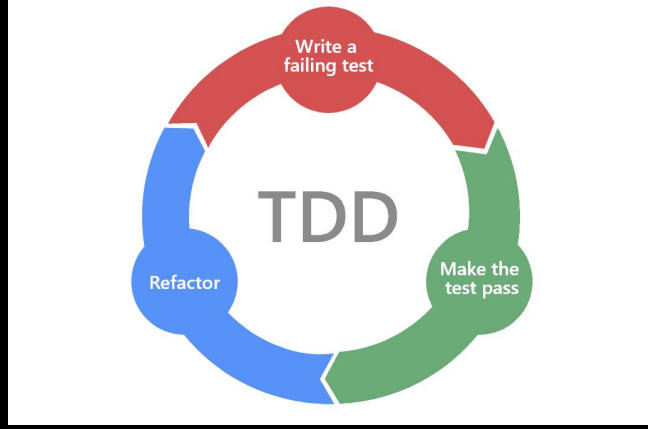
The AAA (Arrange • Act • Assert) Framework

Phase	Purpose	Example
Arrange	set up data / state	input_arr = [3, 1, 2]
Act	call the unit under test	sorted_arr = sort_numbers(input_arr)
Assert	check expectations	assert sorted_arr == [1, 2, 3]

Anatomy of a Readable Test

```
# Arrange

data = "hello"  # input

# Act

result = reverse_string(data)

# Assert

assert result == "olleh"
```

Test-Driven Development (TDD)



Cycle: Red → Green → Refactor

- Red: write a failing test that captures desired behaviour
- Green: implement minimal code to make it pass
- Refactor: clean up code & tests with confidence

Benefits:

- Forces you to define behavior first
- Drives modular, decoupled design
- Produces up-to-date regression suite

Common Pitfalls & How to Avoid Them

Pitfall	Symptom	Fix
Brittle tests	Fail after harmless changes	Assert outputs, not internals
Over-mocking (a fake object that stands in for a real dependency in tests.)	Tests mirror what you already implemented, not behavior	Mock only hard-to-reach deps
False sense of security	High coverage ≠ bug-free	Combine with code reviews & static analysis

“Unittest” module and pytest

2. Using the built-in `unittest` framework

[2]
```python
# import unittest and your function
import unittest

# Re-use the same function from above

class TestAddTwo(unittest.TestCase):
    def test_positive(self):
        self.assertEqual(add_two(10), 12)

    def test_zero(self):
        self.assertEqual(add_two(0), 2)

    def test_negative(self):
        self.assertEqual(add_two(-5), -3)

# run tests in-notebook
if __name__ == "__main__":
    unittest.main(argv=[""], verbosity=2, exit=False)
```

```
test_negative (__main__.TestAddTwo.test_negative) ... ok
test_positive (__main__.TestAddTwo.test_positive) ... ok
test_zero (__main__.TestAddTwo.test_zero) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.010s

OK
```

3. Writing pytest files and invoking `pytest`

Let's make an example_stats.py file

%% Is a magic command to make our notebook behave more like normal terminals/IDEs

```
[3] # write an example module
    %%writefile example_stats.py
    def is_even(n: int) -> bool:
        """Return True if n is even."""
        return n % 2 == 0
```

```
Writing example_stats.py
```

We can see that we now have an example_stats.py file

```
[4] # write a pytest test file
    %%writefile test_example_stats.py
    import pytest
    from example_stats import is_even

    @pytest.mark.parametrize("n,expected", [
        (2, True),
        (3, False),
        (0, True),
        (-4, True),
    ])
    def test_is_even(n, expected):
        assert is_even(n) == expected

    def test_is_even_typeerror():
        with pytest.raises(TypeError):
            is_even("not a number")
```

```
Writing test_example_stats.py
```

```
[5] # run pytest in the notebook
    !pytest -q test_example_stats.py
```

```
.....                                                    [100%]
5 passed in 0.22s
```

Task today: Implement standard functions by hand

- Mean (Compute arithmetic mean without built-in sum.)
- Median (Compute median (manual sort allowed).)
- Summary statistics (Return {"mean": ..., "median": ..., "min": ..., "max": ...}.)
- Quantile (Generic quantile ($0 \leq q \leq 1$) using linear interpolation.)

Pre-Lab Instructions

1) Sign into GitHub, if you haven't already. Accept the invite for today's lab:
   a) https://classroom.github.com/a/Sfeyt03D
2) Make sure you're signed into your GitHub in your VSCode. Clone the repo (code -> local -> HTTPS to get the url)
   a) command/control + shift + p to get Git: clone in VSCode
3) Follow Set-Up instructions
4) To "turn it in" you will edit your forked GitHub repo online or push the changes.

GitHub Classroom (auto-grade your turned in assignments)