Using `pytest`

Basic Usage

- Using the assert statement
- Pytest discovery and using `setup.py`

Integration with Visual Studio Code

- Use “Python: discover tests” to setup the integration with VSC

Pytest helpers

- If you want to use a temporary folder use `tempdir`
- Use `pytest.approx` for float comparisons

Exercise 1

Write a test for the writefile function in `funcs.py`

Write testable code

Mock and patch

- You can use `unittest.mock.patch` to "rewire" a function to an object that you control in the test
- This object is a `unittest.mock.MagicMock`
  - For example, to set a return value, use `mock.return_value = True`
  - This is a method for dealing with external dependencies

Exercise 2

Write a test for `funcs.py::get_buildings` that mocks the connection.

Using fixtures

- Pytest has a couple of builtin helpers, such as `tempdir`
- You can create your own custom helpers, called “fixtures”
- This is useful for things that are used across multiple tests
- Can also be done with `unittest.TestCase.setup`

Using fixtures

```
def test_badsum_with_fixture(correct_sums):
    for a, b, c in correct_sums:
        assert funcs.badsum(a, b) == c

@pytest.fixture
def correct_sums() -> List[Tuple[int, int, int]]:
    pass
```

Integration with VS Code / workflow

- Kristiaan

Testing implementation vs. interface

- See example, `impl.py` and `test_impl.py`

```
class GroupSummer:
    """
    Things should be a list of tuples (group, amount)

    [
        ('A', 1), ('B', 3), ('C', 4)
    ]
    """

    def __init__(self, things: List[Tuple[str, int]]):
        ...

    def groupsum(self, group) -> int:
        ...
```

Testing implementation vs. interface

- You should test the interface, which is the creation and the `groupsum` method
- The field `_groups` is part of the implementation
- This is also implied by the underscore
- You should also not rely on implementation from outside of the function
  In Python, if a name is intended to be "private", it is prefixed by an underscore. Private variables are enforced in Python only by convention.

Problem

- Is mocking always bad because it relies on implementation?

Test driven development

Test-driven development cycle

1. Add a test
- In test-driven development, each new feature begins with writing a test.
2. Run all tests and see if the new test fails
3. Write the code
4. Run tests
5. Refactor code