Nile Wilson, PhD
Senior Data Scientist, Microsoft Industry Solutions Engineering

Microsoft

Writing production-level data science code:

The value of unit testing

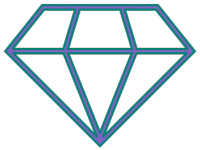
Nile Wilson

Sr Data Scientist, Microsoft Industry Solutions Engineering

- PhD in bioengineering,
  focus on Brain-Computer Interfaces
  (University of Washington, Seattle, 2019)

- Over 3.5 years of experience developing production-level data science solutions with enterprise customers

Session Goals

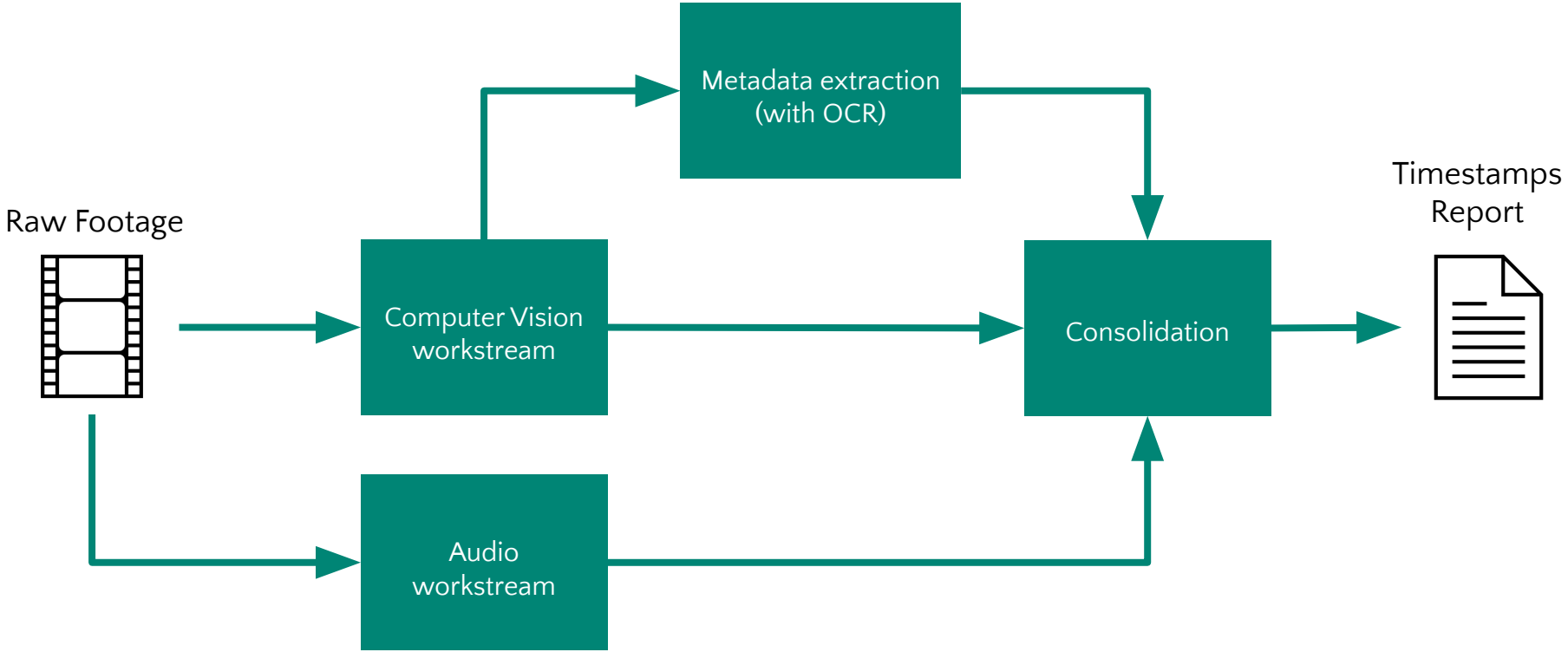


Highlight value of unit testing data science code

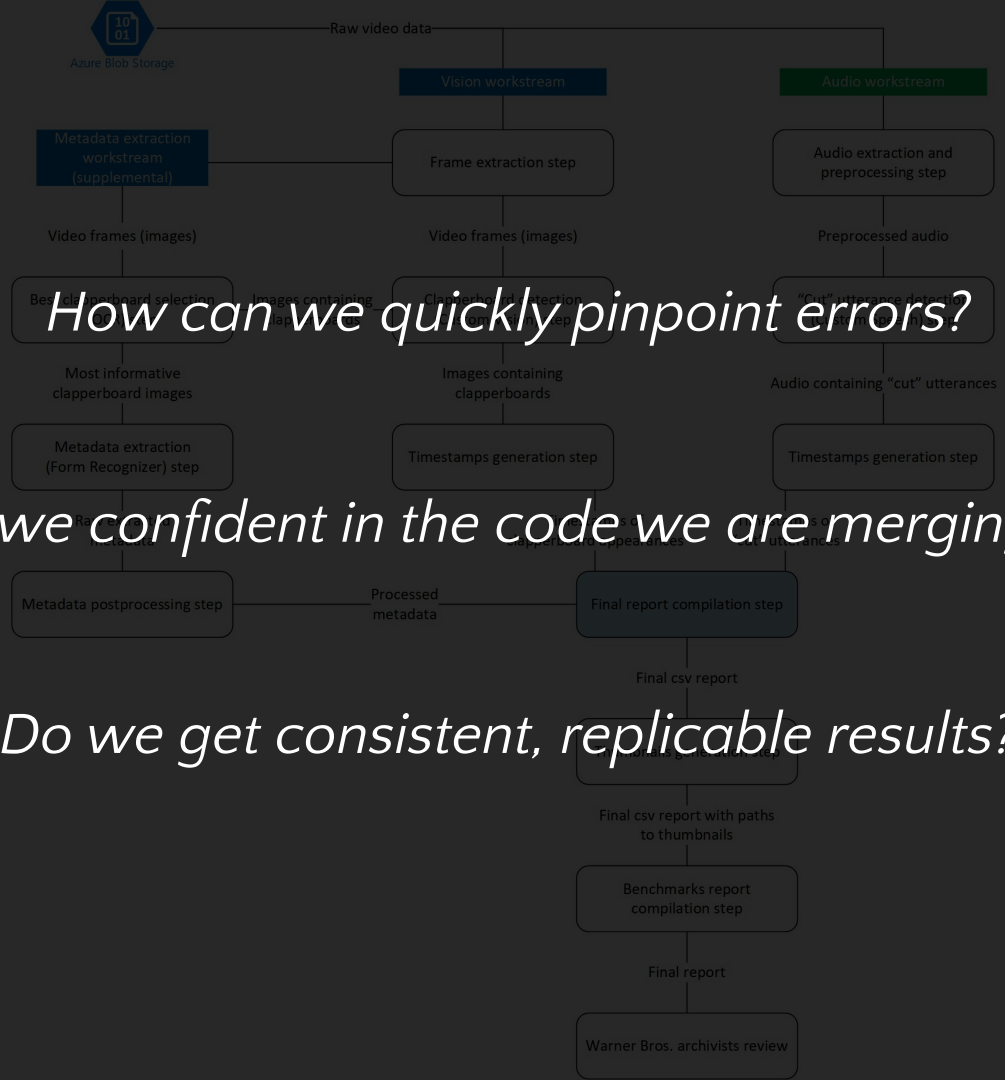
Cover key concepts

Empower you to write production-ready code

Case Study: WarnerMedia video archival



aka.ms/cse-warnermedia-action-cut



How can we quickly pinpoint errors?

Are we confident in the code we are merging in?

Do we get consistent, replicable results?

aka.ms/cse-warnermedia-action-cut

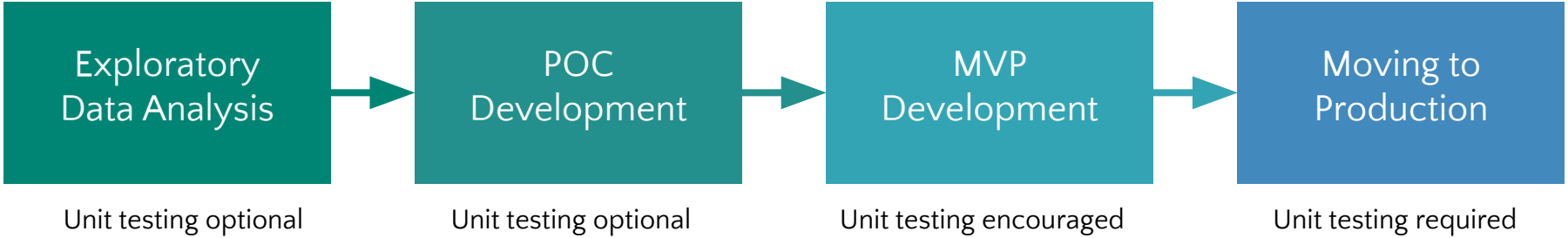
What is unit testing?

Verifies that each “unit” of code works as expected

Helpful with

- Collaborative coding
- Reproducibility
- Reducing debugging time

When are unit tests written?



Alternate Approach

Test-driven development (TDD), where tests are developed alongside code from the start

Unit Testing

General concepts and specific techniques

General Concepts

Clean code

The 3 A's

Sample Data

PR builds

Clean Code

```python
class ProduceInsights:
    """Class to provide insights on the example produce dataframe"""

    def __init__(self, data_path: str) -> None:
        """Initialize the ProduceInsights class.

        :param data_path: Path to input data csv file.
        """
        self.df = self._load_data(data_path)
        self.results = None

    def process_and_update_data(self, arg1: int = 5, arg2: float = 1.2) -> None:
        """Process the data as needed.
```

Docstring

Organized methods and classes, clear naming, docstrings and type hinting

The 3 A's

- Arrange –> Act –> Assert
- **Arrange**: Prepare arguments to pass into the function being tested
- **Act**: Call the function being tested
- **Assert**: Compare actual vs expected

```python
def test_load_data_happy_path(
    produce_fixture: ProduceInsights
):
    """Test happy path for _load_data."""
    # Arrange
    data_path = SOURCE_DATA
    expected_df = pd.read_csv(data_path)

    # Act
    loaded_data = produce_fixture._load_data(
        data_path=data_path
    )

    # Assert
    assert_frame_equal(
        loaded_data,
        expected_df
    )
```

Sample Data

Representative input and edge cases

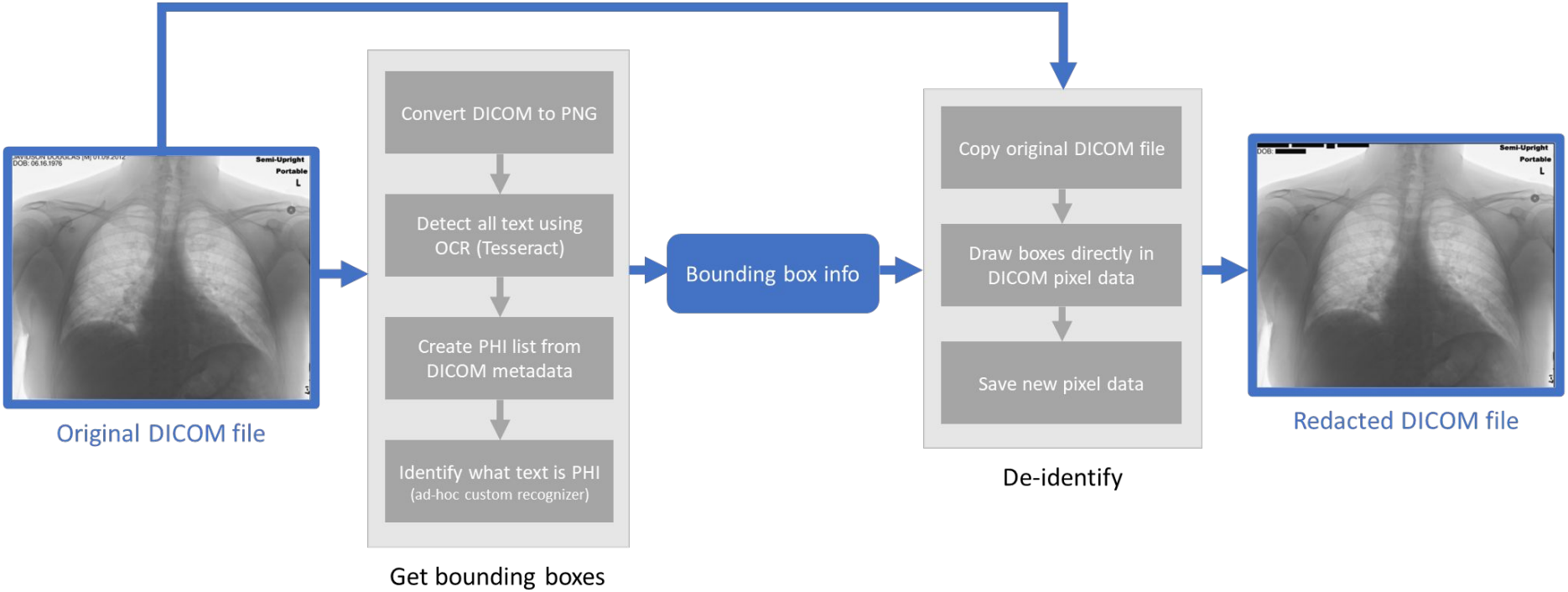
Why?

- Focus on the tested method
- Minimize execution time
- Maintain small repo size

How?

- Create samples from open-source data (with appropriate licensing)
- Create synthetic data

Example: Medical Image Redaction



aka.ms/presidio-dicom-blog

https://wiki.cancerimagingarchive.net/pages/viewpage.action?pageId=80969777

Confluence Spaces Search Log in

The Cancer Imaging Archive (TCIA) Public Access

Blog

SPACE SHORTCUTS

How-to articles

Troubleshooting articles

Content Formatting Templates

CHILD PAGES

Pages / Wiki / Collections

A DICOM dataset for evaluation of medical image de-identification (Pseudo-PHI-DICOM-Data)

Created by Erica Bilello, last modified on Sep 22, 2022

Summary

Selected images from open-source dataset for the evaluation of medical image de-identification and unit testing

PR Builds

Validate that tests pass before merging changes via Pull Request

How it works

- Build validation policy set in repo
- Tests are run automatically upon PR update
- PR is blocked if tests fail

Writing Techniques

Fixtures

Parameterization

Exceptions

Mocking

Fixtures

- Pass the same object into multiple tests
- Reduce redundancy
- Scope can be modified

```python
@pytest.fixture(scope="module")
def produce_fixture():
    """Produce fixture to use in tests."""
    return ProduceInsights(data_path=SOURCE_DATA)

def test_calculate_price_happy_path(
    produce_fixture: ProduceInsights
):
    """Test happy path for _calculate_price"""
    # Arrange
    input_df = pd.read_csv(SOURCE_DATA)
    expected_df = pd.read_csv(EXPECTED_DATA)

    # Act
    test_df = produce_fixture._calculate_price(input_df)

    # Assert
    assert_frame_equal(test_df, expected_df)

def test_load_data_happy_path(
    produce_fixture: ProduceInsights
):
    """Test happy path for _load_data."""
```

Parameterization

Arguments being parameterized

```
91   # Load data exceptions
92   @pytest.mark.parametrize(
93       "data_path, expected_error_type",
94       [
95           ("./non_existent_file.csv", "FileNotFoundError"),
96           ("../other_non_existent_file.csv", "FileNotFoundError"),
97           ("not_created_file.csv", "FileNotFoundError"),
98       ],
99   )
100  def test_load_data_exceptions(
101      produce_fixture: ProduceInsights, data_path: str, expected_error_type: str
102  ):
```

Sets of argument values

Test edge cases & multiple sets of arguments, reduce redundancy in test definition

Exceptions

```python
# Load data exceptions
@pytest.mark.parametrize(
    "data_path, expected_error_type",
    [
        ("./non_existent_file.csv", "FileNotFoundError"),
        ("../other_non_existent_file.csv", "FileNotFoundError"),
        ("not_created_file.csv", "FileNotFoundError"),
    ],
)
def test_load_data_exceptions(
    produce_fixture: ProduceInsights, data_path: str, expected_error_type: str
):
    """Test error handling of _load_data in ProduceInsights."""
    with pytest.raises(Exception) as exc_info:
        # Arrange

        # Act
        _ = produce_fixture._load_data(data_path=data_path)

        # Assert
        assert expected_error_type in exc_info

```

Verify exceptions are raised correctly in certain conditions

```python
    def process_and_update_data(
        self,
        arg1: int = 5,
        arg2: float = 1.2
    ) -> None:
        """Process the data as needed.

        :param arg1: Some argument.
        :param arg2: Some other argument.
        """
        # Calculate total
        df_total = self._calculate_price(self.df)

        # Then apply complex transformation
        df_transformed = complex_transformation(
            df_total, arg1, arg2
        )

        # Update processed data
        self.results = df_transformed
```

Primary method (method we want to test)

Intermediary methods
(other methods called inside)

Mocking

- Focus on testing primary method
- Predefine intermediary return values (mock data)
- Check mocker patches are called

```python
@pytest.mark.parametrize(
    "arg1, arg2",
    [(5, 1.2), (-7, 0.65), (99013, 42.37)],
)
def test_process_and_update_data(
    mocker,
    produce_fixture: ProduceInsights,
    arg1: int,
    arg2: float
):
    # Arrange
    test_insight = deepcopy(produce_fixture)
    mock_calcaulte_price = mocker.patch(
        "src.utils.produce.ProduceInsights._calculate_price",
        return_value=pd.DataFrame(),
    )
    mock_complex_transformation = mocker.patch(
        "src.utils.produce.complex_transformation",
        return_value=pd.DataFrame()
    )
    # Act
    test_insight.process_and_update_data(arg1, arg2)

    # Assert
    assert mock_calcaulte_price.call_count == 1
    assert mock_complex_transformation.call_count == 1
    assert type(test_insight.results) == pd.DataFrame
```

Mocking

```python
19      def process_and_update_data(
20          self,
21          arg1: int = 5,
22          arg2: float = 1.2
23      ) -> None:
24          """Process the data as needed.
25
26          :param arg1: Some argument.
27          :param arg2: Some other argument.
28          """
29          # Calculate total
30          df_total = self._calculate_price(self.df)
31
32          # Then apply complex transformation
33          df_transformed = complex_transformation(
34              df_total, arg1, arg2
35          )
36
37          # Update processed data
38          self.results = df_transformed
```

Original code

```python
19      def process_and_update_data(
20          self,
21          arg1: int = 5,
22          arg2: float = 1.2
23      ) -> None:
24          """Process the dat
25
26          :param arg1: Some argument.
27          :param arg2: Some other argument.
28          """
29          # Calculate total
30          df_total = pd.DataFrame(
31                     )
32          # Then apply complex transformation
33          df_transformed = pd.DataFrame(
34                           )
35
36
37          # Update processed data
38          self.results = df_transformed
```

Return values specified in the mock calls

Effective code when we mock called methods

```
19      def process_and_update_data(
20          self,
21          arg1: int = 5,
22          arg2: float = 1.2
23      ) -> None:
24          """Process the dat
25
26          :param arg1: Some argument.
27          :param arg2: Some other argument.
28          """
29          # Calculate total
30          df_total = pd.DataFrame(
31                     )
32          # Then apply complex transformation
33          df_transformed = pd.DataFrame(
34                           )
35
36
37          # Update processed data
38          self.results = df_transformed
```

Return values specified in the mock calls

Yes, do mock:

- ✓ Imported library calls
- ✓ Custom methods with tests
- ✓ Calls to external services

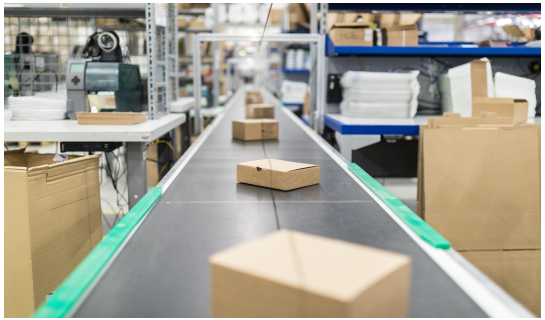
No, do not mock:

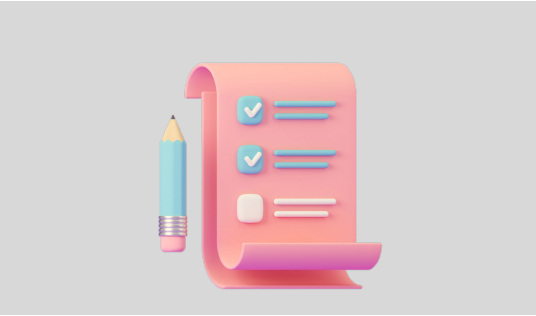
- ✗ Custom methods **without** tests
- ✗ Basic necessary operations
  (e.g., pandas.read_csv(), np.as_array())

Conclusion

Recap and Q&A

Value of unit testing DS code





Production

One essential part of writing production-level solutions

Quality

Promotes quality code and reduces downstream headache

Collaboration

Improve collaborative development experience

Unit testing is not just for software engineers. It's for data scientists, too!

Covered concepts

Clean code

The 3 A's

Sample Data

PR builds

Fixtures

Parameterization

Exceptions

Mocking

Resources

01 Code-with engineering playbook
https://microsoft.github.io/code-with-engineering-playbook/machine-learning/ml-testing/

02 Test practices for data science applications using Python
https://medium.com/data-science-at-microsoft/testing-practices-for-data-science-applications-using-python-71c271cd8b5e

03 Example repository used in this deck
https://github.com/niwilso/ds-unit-testing

04 Unit testing best practices
https://brightsec.com/blog/unit-testing-best-practices/

05 Pytest documentation
https://docs.pytest.org/en/7.2.x/

06 Testing Best Practices for ML Libraries
https://towardsdatascience.com/testing-best-practices-for-machine-learning-libraries-41b7d0362c95

Microsoft

Q&A

Contact Info:

/nile-wilson

Appendix

What to Test

- Unit tests exist to ensure that the **functions we develop** work as expected
- Calls to external services should not be tested in **unit** tests
- 100% code coverage is not usually required for a repository

Library to check Pytest coverage:
https://pypi.org/project/pytest-cov/

Different types of tests

- **Integration test**: Ensure full solution (which uses the units) works as expected given a toy dataset
- **Smoke tests**: Ensure all critical aspects of the solution work as expected given a toy dataset

Mock data

- When mocking calls, we sometimes need to create “mock data” to use as the return value
- Can be as simple or as complex as needed
- Can be defined in the test script (e.g., parameterized input, fixture, or in # Arrange) or imported from another file

```python
@pytest.mark.parametrize(
    "arg1, arg2",
    [(5, 1.2), (-7, 0.65), (99013, 42.37)],
)
def test_process_and_update_data(
    mocker,
    produce_fixture: ProduceInsights,
    arg1: int,
    arg2: float
):
    # Arrange
    test_insight = deepcopy(produce_fixture)
    mock_calcaulte_price = mocker.patch(
        "src.utils.produce.ProduceInsights._calculate_price",
        return_value=pd.DataFrame(),
    )
    mock_complex_transformation = mocker.patch(
        "src.utils.produce.complex_transformation",
        return_value=pd.DataFrame()
    )
    # Act
    test_insight.process_and_update_data(arg1, arg2)

    # Assert
    assert mock_calcaulte_price.call_count == 1
    assert mock_complex_transformation.call_count == 1
    assert type(test_insight.results) == pd.DataFrame
```