Unit Testing For Data Scientists

Hanna Torrence

Data Scientist

ANN TAYLOR

EXPRESS

bloomingdale's

COLE HAAN

BOSS
HUGO BOSS

Neiman Marcus

Neiman Marcus
lastcall

Lord & Taylor

Amazon Prime for everyone else:
Our 6 million members get free two-day shipping, returns, and deals across a growing network of 140+ retailers.

Saks Fifth Avenue

Saks Fifth Avenue OFF 5TH

Vera Bradley

Timberland

JIMMY CHOO

GIORGIO ARMANI

VINCE.

Calvin Klein

NBA STORE
NBASTORE.COM

BERGDORF GOODMAN

bebe

1. **Getting Started with Unit Tests**
2. Pytest
3. Fixtures
4. Mocks
5. Helpful Libraries
6. Resources

Testing Terms You May Have Heard

Unit test

Test of a single unit of code in isolation

Integration test

Test how different components of a system work together

Regression test

Testing a previously working program after a change to ensure no problems have been introduced

Smoke test

A subset of test cases verifying the core of a system

Alpha test

In-house test of final functionality of an application

Beta test

Initial release to a subset of real users

Testing Terms You May Have Heard

Unit test

Test of a single unit of code in isolation

Integration test

Test how different components of a system work together

Regression test

Testing a previously working program after a change to ensure no problems have been introduced

Smoke test

A subset of test cases verifying the core of a system

Alpha test

In-house test of final functionality of an application

Beta test

Initial release to a subset of real users

Benefits of Unit Testing

Well-tested code helps you:

- Find bugs earlier
- Iterate faster
- Debug more easily
- Design better code

Benefits of Unit Testing

Confidence that your code does what you think it does

So why doesn’t everyone write unit tests?

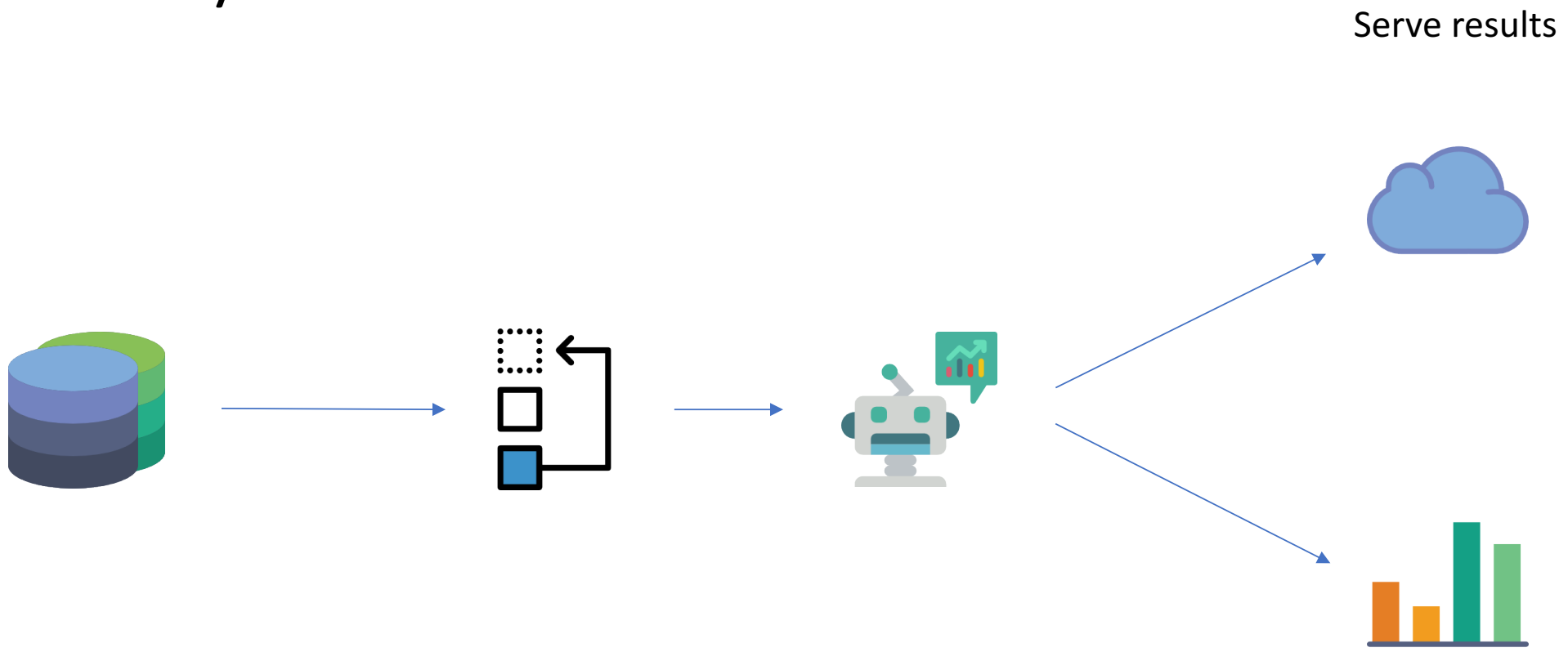
So why doesn’t everyone write unit tests?

Learning to write good tests is an investment…

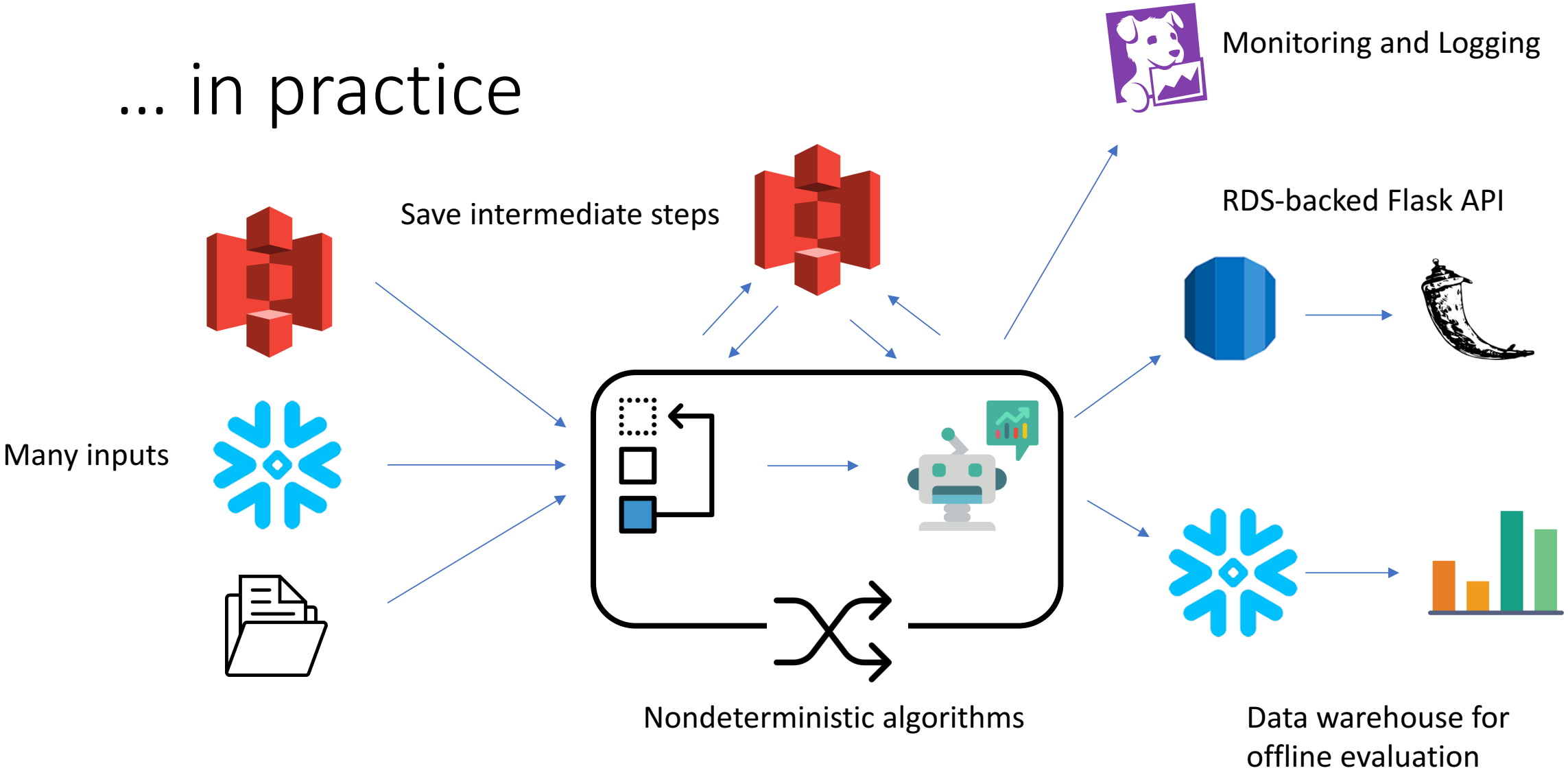
Learning to write good tests is an investment…

… and data science work follows different patterns that much of software engineering

In theory…


Serve results
Read in data
Build features
Build a model
Evaluate offline

… in practice


Monitoring and Logging
Save intermediate steps
RDS-backed Flask API
Many inputs
Nondeterministic algorithms
Data warehouse for offline evaluation

This quickly gets overwhelming

Where to start?

- Pick a single, specific functionality to verify

Where to start?

- Pick a single, specific functionality to verify
- Use available tools to get everything else out of the way

Where to start?

- Pick a single, specific functionality to verify
- Use available tools to get everything else out of the way
- In an existing codebase, don’t try to write all the tests at once

Where to start?

- Pick a single, specific functionality to verify
- Use available tools to get everything else out of the way
- In an existing codebase, don’t try to write all the tests at once
- Write tests as early as they can be valuable

1. Getting Started with Unit Tests
2. **Pytest**
3. Fixtures
4. Mocks
5. Helpful Libraries
6. Resources

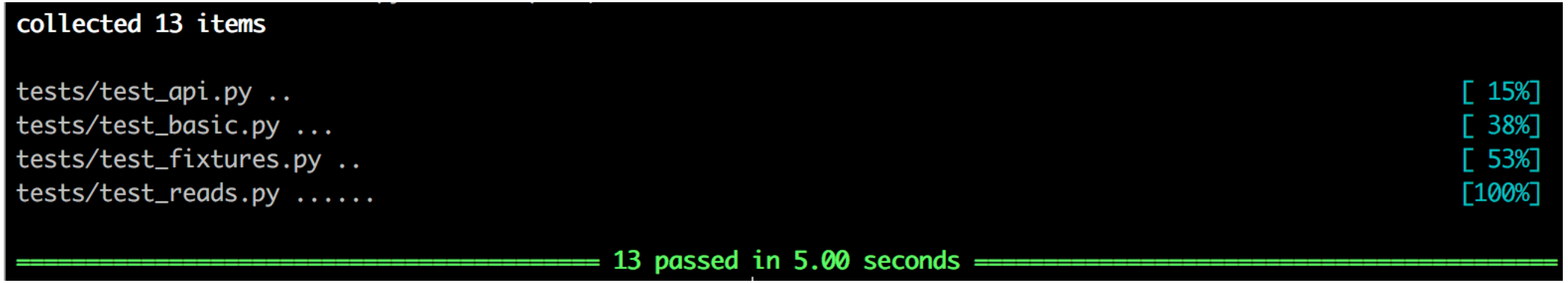
Pytest Framework

highly configurable + very little boilerplate

```
collected 13 items

tests/test_api.py ..                                                    [ 15%]
tests/test_basic.py ...                                                 [ 38%]
tests/test_fixtures.py ..                                               [ 53%]
tests/test_reads.py ......                                              [100%]

========================================== 13 passed in 5.00 seconds ==========================================
```

Useful Options for Pytest

-s (print all string output)
-v (print names of individual tests as they run)
-x (stop at first failure)
-k (only run tests matching following keywords)



Add a Column

```
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    df[new_col_name] = default_value

    return df
```

Add a Column

```
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    df[new_col_name] = default_value

    return df
```

Add a Column

```
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    df[new_col_name] = default_value

    return df
```

Add a Column

```
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    df[new_col_name] = default_value

    return df
```

Add a Column

```
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    df[new_col_name] = default_value

    return df
```

Add a Column

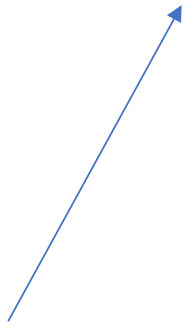
```
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    df[new_col_name] = default_value

    return df
```

Unit Test

```python
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    df[new_col_name] = default_value

    return df
```

```python
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

Unit Test



give our test a meaningful name starting with `test_`

```python
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

```python
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    Jf[new_col_name] = default_value

    return df
```

Unit Test

```python
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    df[new_col_name] = default_value

    return df
```

define the input to the function we're testing

```python
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

Unit Test

```python
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    df[new_col_name] = default_value

    return df
```

```python
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

call the function we're testing →

Unit Test

```python
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    df[new_col_name] = default_value

    return df
```

```python
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

define the output we are expecting

Unit Test

```python
def add_col(df, new_col_name, default_value):
    """Add a new column with a default value"""

    df[new_col_name] = default_value

    return df
```

```python
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

check that the actual output matches the output we were expecting

1. Getting Started with Unit Tests
2. Pytest
3. **Fixtures**
4. Mocks
5. Helpful Libraries
6. Resources

Fixtures

- Special functions pytest keeps track of to safely share resources and/or resource definitions

- A modular approach to setup & teardown methods

Defining New Fixtures

```
@pytest.fixture()
def df():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })
```

```
@pytest.fixture()
def df_with_col_d():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })
```

Defining New Fixtures

```python
@pytest.fixture()
def df():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })
```

decorator tells pytest this is a fixture

```python
@pytest.fixture()
def df_with_col_d():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })
```

normal python function

Defining New Fixtures

```
@pytest.fixture()
def df():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })
```

```
@pytest.fixture()
def df_with_col_d():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })
```

Remember to import fixtures into `conftest.py` file!

Defining New Fixtures

```python
@pytest.fixture()
def df():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })


@pytest.fixture()
def df_with_col_d():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })
```

```python
def test_add_col_passes_with_fixtures(df, df_with_col_d):
    actual = add_col(df, 'col_d', 'd')
    expected = df_with_col_d

    pd.testing.assert_frame_equal(actual, expected)
```

Defining New Fixtures

pytest passes in value returned from `df` function

```
@pytest.fixture()
def df():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })


@pytest.fixture()
def df_with_col_d():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })
```

```
def test_add_col_passes_with_fixtures(df, df_with_col_d):
    actual = add_col(df, 'col_d', 'd')
    expected = df_with_col_d

    pd.testing.assert_frame_equal(actual, expected)
```

Defining New Fixtures

```python
@pytest.fixture()
def df():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })


@pytest.fixture()
def df_with_col_d():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })
```

pytest passes in value returned from `df_with_col_d` function

```python
def test_add_col_passes_with_fixtures(df, df_with_col_d):
    actual = add_col(df, 'col_d', 'd')
    expected = df_with_col_d

    pd.testing.assert_frame_equal(actual, expected)
```

Useful Built-in Fixtures

`capsys` captures any values written to `stderr` or `stdout` during the execution of the test

To see all available fixtures, run:

```
pytest --fixtures
```

```
def test_function_1(capsys):
    function_1()
    out, err = capsys.readouterr()
    assert out == 'Inside function 1\n'
```

Flexibility of Fixtures

You can:

- Define the scope of a fixture
- Compose fixtures
- Execute custom teardown code when leaving scope
- Access the test context inside fixture
- Parameterize fixtures

Flexibility of Fixtures

```
@pytest.fixture(scope='session')
def spark(request):
    spark = (
        SparkSession
        .builder
        .appName('pytest-pyspark-local-testing')
        .master('local[2]')
        .getOrCreate()
    )

    request.addfinalizer(lambda: spark.stop())

    return spark
```

Flexibility of Fixtures

define fixture scope

```
@pytest.fixture(scope='session')
def spark(request):

    spark = (
        SparkSession
        .builder
        .appName('pytest-pyspark-local-testing')
        .master('local[2]')
        .getOrCreate()
    )

    request.addfinalizer(lambda: spark.stop())

    return spark
```

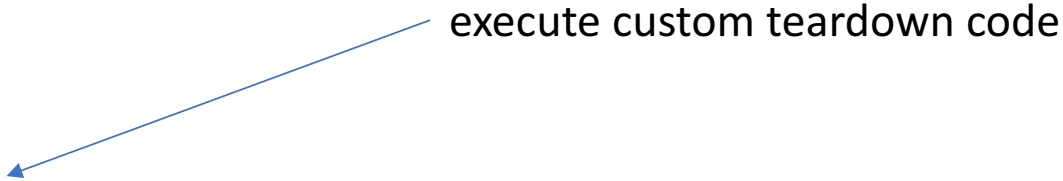
Flexibility of Fixtures

```
@pytest.fixture(scope='session')
def spark(request):
    spark = (
        SparkSession
        .builder
        .appName('pytest-pyspark-local-testing')
        .master('local[2]')
        .getOrCreate()
    )

    request.addfinalizer(lambda: spark.stop())

    return spark
```

access test context with `request` argument

Flexibility of Fixtures

```
@pytest.fixture(scope='session')
def spark(request):
    spark = (
        SparkSession
        .builder
        .appName('pytest-pyspark-local-testing')
        .master('local[2]')
        .getOrCreate()
    )

    request.addfinalizer(lambda: spark.stop())

    return spark
```



Flexibility of Fixtures

```python
@pytest.fixture(scope='session')
def spark(request):

    spark = (
        SparkSession
        .builder
        .appName('pytest-pyspark-local-testing')
        .master('local[2]')
        .getOrCreate()
    )

    request.addfinalizer(lambda: spark.stop())

    return spark
```

define a new fixture that depends on the first


```python
@pytest.fixture()
def spark_df(spark):
    return spark.createDataFrame(
        [
            ('a', 'b', 'c', 'd'),
            ('a', 'b', 'c', 'd')
        ],
        ['col_a', 'col_b', 'col_c', 'col_d']
    )
```

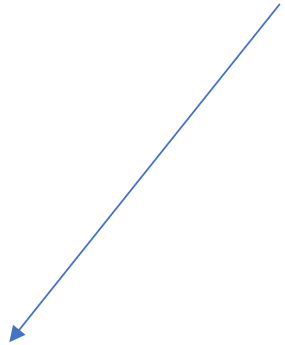
1. Getting Started with Unit Tests
2. Pytest
3. Fixtures
4. **Mocks**
5. Helpful Libraries
6. Resources

Reading from a Database

```
def generate_features(db_creds):

    #  ... some setup code ...

    eng = sqlalchemy.create_engine('fake_connection_string')
    df = pd.read_sql('SELECT col1, col2 FROM data_table;',con=eng)

    #  ... processing on df ...

    return features
```

Reading from a Database

Replace connection with a dummy object



```
def generate_features(db_creds):

    #  ... some setup code ...

    eng = sqlalchemy.create_engine('fake_connection_string')
    df = pd.read_sql('SELECT col1, col2 FROM data_table;',con=eng)

    #  ... processing on df ...

    return features
```

Reading from a Database

Replace connection with a dummy object

```
def generate_features(db_creds):

    #  ... some setup code ...

    eng = sqlalchemy.create_engine('fake_connection_string')
    df = pd.read_sql('SELECT col1, col2 FROM data_table;',con=eng)

    #  ... processing on df ...

    return features
```

Return some fake data into df

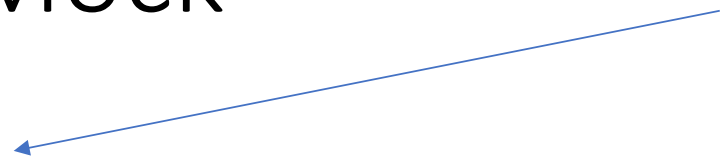
Test with Mock

```python
@mock.patch('pytest_examples.functions_to_test.sqlalchemy.create_engine')
@mock.patch('pytest_examples.functions_to_test.pd.read_sql')
def test_generate_features(read_sql_mock, engine_mock, db_creds, df):
    read_sql_mock.return_value = df

    actual_features = generate_features(db_creds)

    pd.testing.assert_frame_equal(actual_features, df)
```

```python
def generate_features(db_creds):

    #  ... some setup code ...

    eng = sqlalchemy.create_engine('fake_connection_string')
    df = pd.read_sql('SELECT col1, col2 FROM data_table;',con=eng)

    #  ... processing on df ...

    return features
```

Test with Mock

patch things we want to cut out with Mock objects



```python
@mock.patch('pytest_examples.functions_to_test.sqlalchemy.create_engine')
@mock.patch('pytest_examples.functions_to_test.pd.read_sql')
def test_generate_features(read_sql_mock, engine_mock, db_creds, df):
    read_sql_mock.return_value = df

    actual_features = generate_features(db_creds)

    pd.testing.assert_frame_equal(actual_features, df)
```

```python
def generate_featur

    #  ... some setup code ...

    eng = sqlalchemy.create_engine('fake_connection_string')
    df = pd.read_sql('SELECT col1, col2 FROM data_table;',con=eng)

    #  ... processing on df ...

    return features
```

Test with Mock

patch things we want to cut out with Mock objects

```
@mock.patch('pytest_examples.functions_to_test.sqlalchemy.create_engine')
@mock.patch('pytest_examples.functions_to_test.pd.read_sql')
def test_generate_features(read_sql_mock, engine_mock, db_creds, df):
    read_sql_mock.return_value = df

    actual_features = generate_features(db_creds)

    pd.testing.assert_frame_equal(actual_features, df)
```

set return value

```
def generate_featur

    #  ... some setup code ...

    eng = sqlalchemy.create_engine('fake_connection_string')
    df = pd.read_sql('SELECT col1, col2 FROM data_table;',con=eng)

    #  ... processing on df ...

    return features
```

Mocks

Replacing an object with a “mock” allows you to avoid external dependencies

Things you might want to mock:

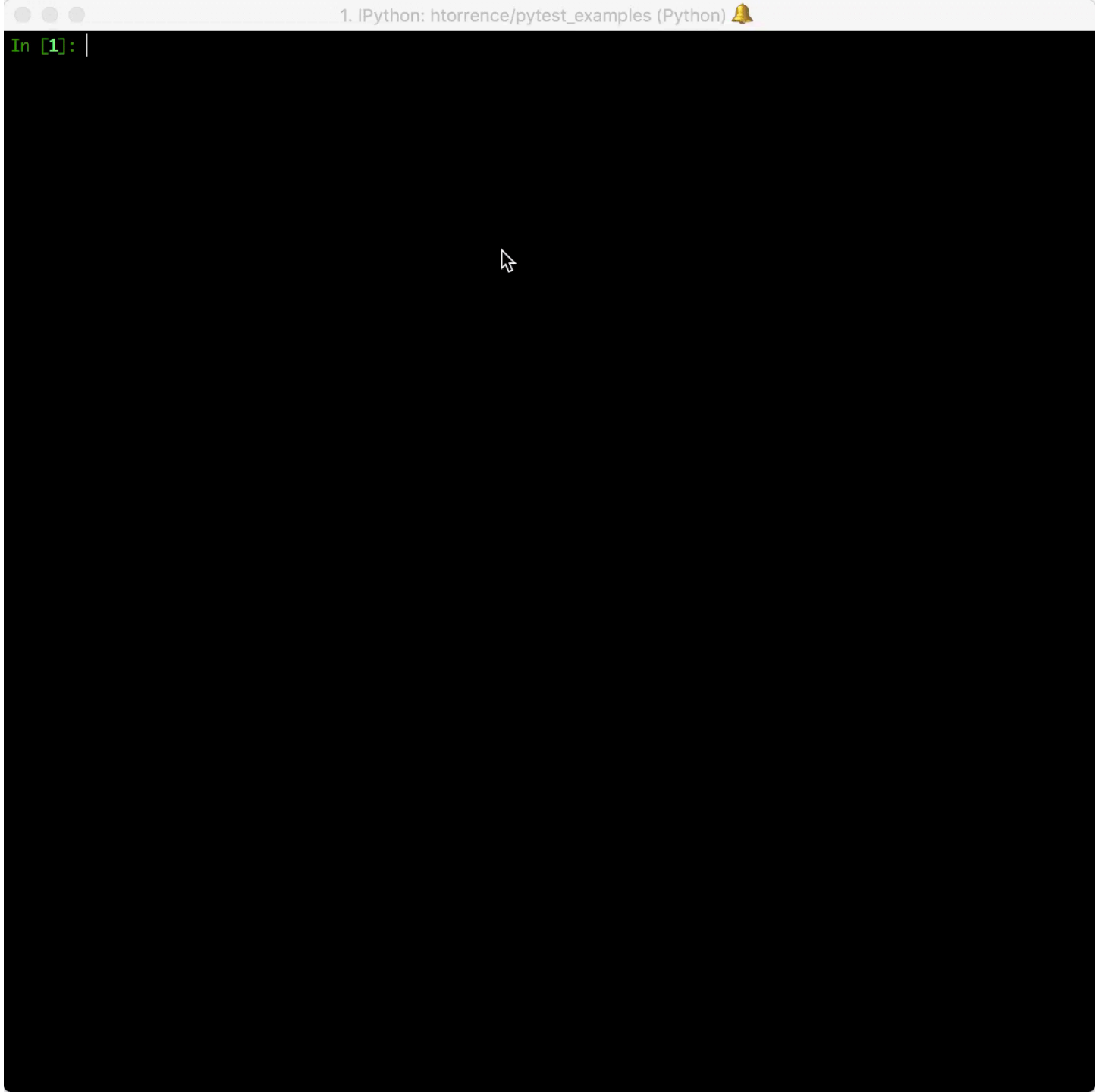
- Data reads or writes
- API calls
- External functions you don’t want to test

Mocks

Python's `unittest.mock` module uses MagicMock objects

Their behavior can be altered in many ways, but by default they:

- accept any call and any attribute access
- return a new MagicMock object for all call return values and all accessed attributes
- record all calls made to them


1. IPython: htorrence/pytest_examples (Python)
In [1]:

Mocks

Actions taken by mock object:

To return the same value for each call

```
mock_obj.return_value = object_to_return
```

To return different values for each call

```
mock_obj.side_effect = [first_object, second_object]
```

To throw an exception

```
mock_obj.side_effect = Exception()
```

To return different values depending on input

```
mock_obj.side_effect = lambda x: 'odd' if x%2 else 'even'
```

Mocks

Assertions on mock objects:

Called at least one time

```
mock_obj.assert_called()
```

Called exactly one time

```
mock_obj.assert_called_once()
```

Last called with certain arguments

```
mock_obj.assert_called_with(arg1, arg2)
```

Called a several times with certain arguments

```
assert mock_obj.call_args_list == [mock.call(arg1, arg2)]
```

Mocks

Be wary of assertion typos!

Throws an exception if the mock object was not called:

```
mock_obj.assert_called_once()
```

Never throws an exception:

```
mock_obj.called_once()  # does not exist!
```

However, nonexistent methods starting with `assert` do throw an exception:

```
mock_obj.assert_caled()  # typo
```

Gotchas for Mocks

Mock an object where it is used, not where it is defined

```
@mock.patch('pytest_examples.functions_to_test.pd.read_csv')
```

```
@mock.patch('pandas.read_csv')
```

Gotchas for Mocks

Mock an object where it is used, not where it is defined

```
@mock.patch('pytest_examples.functions_to_test.pd.read_csv')
```

```
@mock.patch('pandas.read_csv')
```

Pay attention to decorator order

```
@mock.patch('pytest_examples.functions_to_test.function_1')
@mock.patch('pytest_examples.functions_to_test.function_2')
@mock.patch('pytest_examples.functions_to_test.function_3')
def test_multiple_mocks(function_3_mock, function_2_mock, function_1_mock, df):
```

Gotchas for Mocks

Mock an object where it is used, not where it is defined

```
@mock.patch('pytest_examples.functions_to_test.pd.read_csv')
```

```
@mock.patch('pandas.read_csv')
```

Pay attention to decorator order

```
@mock.patch('pytest_examples.functions_to_test.function_1')
@mock.patch('pytest_examples.functions_to_test.function_2')
@mock.patch('pytest_examples.functions_to_test.function_3')
def test_multiple_mocks(function_3_mock, function_2_mock, function_1_mock, df):

test_multiple_mocks()
```

Gotchas for Mocks

Mock an object where it is used, not where it is defined

```
@mock.patch('pytest_examples.functions_to_test.pd.read_csv')
```

```
@mock.patch('pandas.read_csv')
```

Pay attention to decorator order

```
@mock.patch('pytest_examples.functions_to_test.function_1')
@mock.patch('pytest_examples.functions_to_test.function_2')
@mock.patch('pytest_examples.functions_to_test.function_3')
def test_multiple_mocks(function_3_mock, function_2_mock, function_1_mock, df):

patch_3(test_multiple_mocks())
```

Gotchas for Mocks

Mock an object where it is used, not where it is defined

```
@mock.patch('pytest_examples.functions_to_test.pd.read_csv')
```

```
@mock.patch('pandas.read_csv')
```

Pay attention to decorator order

```
@mock.patch('pytest_examples.functions_to_test.function_1')
@mock.patch('pytest_examples.functions_to_test.function_2')
@mock.patch('pytest_examples.functions_to_test.function_3')
def test_multiple_mocks(function_3_mock, function_2_mock, function_1_mock, df):
```

```
patch_2(patch_3(test_multiple_mocks()))
```

Gotchas for Mocks

Mock an object where it is used, not where it is defined

```
@mock.patch('pytest_examples.functions_to_test.pd.read_csv')
```

```
@mock.patch('pandas.read_csv')
```

Pay attention to decorator order

```
@mock.patch('pytest_examples.functions_to_test.function_1')
@mock.patch('pytest_examples.functions_to_test.function_2')
@mock.patch('pytest_examples.functions_to_test.function_3')
def test_multiple_mocks(function_3_mock, function_2_mock, function_1_mock, df):
```

```
patch_1(patch_2(patch_3(test_multiple_mocks())))
```

1. Getting Started with Unit Tests
2. Pytest
3. Fixtures
4. Mocks
5. **Helpful Libraries**
6. Resources

Helpful Libraries

- Responses
  - Provides `@responses.activate` decorator
  - Add API responses with

```
@responses.activate
def test_cat_api_404():
    responses.add(
        responses.GET,
        'https://cat.reactjsgirls.com/cats',
        json={'whoops, there are no cats!'},
        status=404,
    )
```

Helpful Libraries

- Responses
- Click CliRunner
  - Test command line interface tools built with Click

```
runner = CliRunner()
result = runner.invoke(
    configure,
    input=( )
)
assert result.output == expected_stdout
```

Helpful Libraries

- Responses
- Click CliRunner
- Pandas.testing
  - Testing utilities for pandas dataframes
  - `pd.testing.assert_frame_equal(df1, df2)`

Helpful Libraries

- Responses
- Click CliRunner
- Pandas.testing
- Numpy.testing
  - Testing utilities for numpy arrays
  - `np.testing.assert_almost_equal(arr1, arr2)`

Helpful Libraries

- Responses
- Click CliRunner
- Pandas.testing
- Numpy.testing
- Pytest-cov
  - Plugin for pytest to assess code coverage

Helpful Libraries

- Responses
- Click CliRunner
- Pandas.testing
- Numpy.testing
- Pytest-cov

- For specialized tools, check if someone has written testing helpers

Helpful Libraries

- Responses
- Click CliRunner
- Pandas.testing
- Numpy.testing
- Pytest-cov

- For specialized tools, check if someone has written testing helpers
- If not, write your own!

1. Getting Started with Unit Tests
2. Pytest
3. Fixtures
4. Mocks
5. Helpful Libraries
6. **Resources**

Useful Blog Posts

Getting Started With Pytest *Jacob Kaplan-Moss*

Surrender Python Mocking! I Have You Now *Matt Pease*

Python Mocking 101: Fake It Before You Make It *Mike Lin*

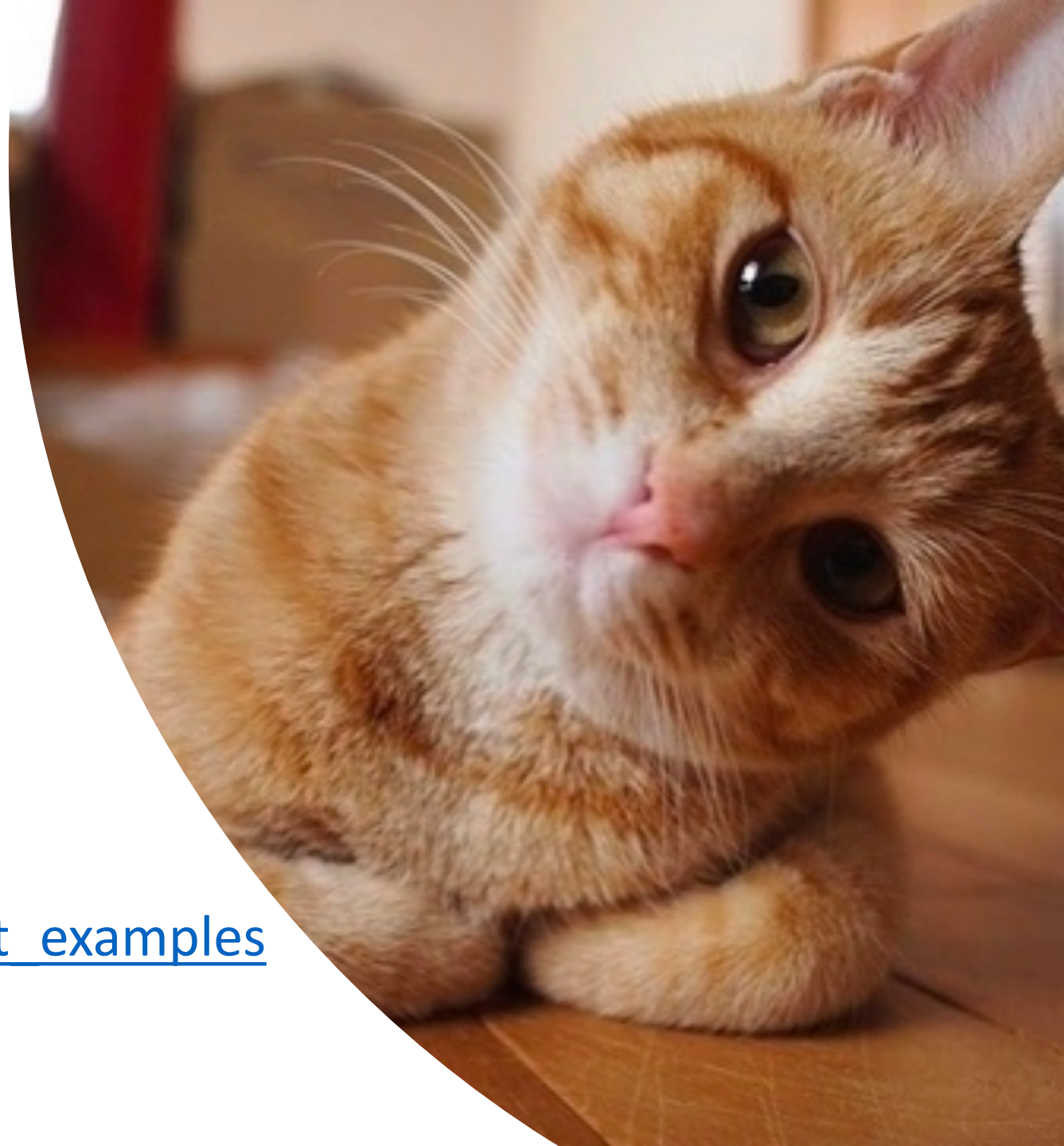
Full reading list in a gist at:

bit.ly/pytest_reading

Questions?

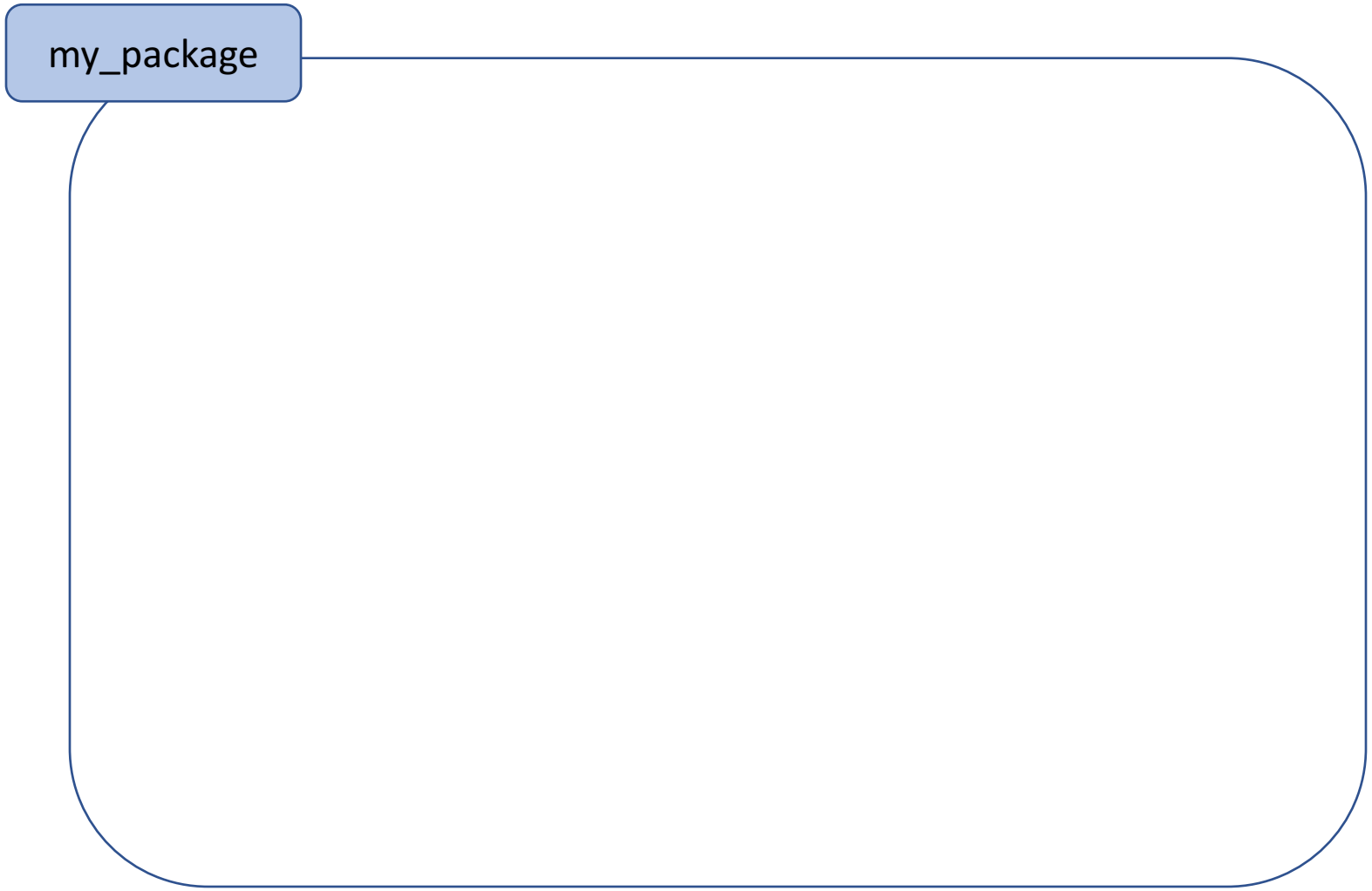
@HannaTorrence
htorrence@shoprunner.com

Extended Code Examples:
https://github.com/htorrence/pytest_examples



Appendix

Imports



Imports

my_package

functions_to_test.py

```
5    import pandas as pd

19   def df_from_csv(filename):
20       """Return a dataframe read from a csv"""
21       return pd.read_csv(filename)
22
```

Imports

my_package

functions_to_test.py

1. Go find pandas module
2. Add module to namespace as `my_package.functions_to_test.pd`

```
5    import pandas as pd


19   def df_from_csv(filename):
20       """Return a dataframe read from a csv"""
21       return pd.read_csv(filename)
22
```

Imports

my_package

functions_to_test.py

1. Go find pandas module
2. Add module to namespace as `my_package.functions_to_test.pd`

```
5    import pandas as pd

19   def df_from_csv(filename):
20       """Return a dataframe read from a csv"""
21       return pd.read_csv(filename)
22
```

Call `my_package.functions_to_test.pd.read_csv`

Imports

Replace `my_package.functions_to_test.pd.read_csv` with a mock object

my_package

test_functions_to_test.py

```
13    @pytest.mark.usefixtures('df')
14    @mock.patch('pytest_examples.functions_to_test.pd.read_csv')
15    def test_df_from_csv_mock_function(read_csv_mock, df):
16        # setup
17        read_csv_mock.return_value = df
18
19        # call function
20        actual = df_from_csv('fake_file_name.csv')
```

With Mocking

my_package

functions_to_test.py

1. Go find pandas module
2. Add module to namespace as `my_package.functions_to_test.pd`

```
5    import pandas as pd

19   def df_from_csv(filename):
20       """Return a dataframe read from a csv"""
21       return pd.read_csv(filename)
22
```

Call Mock object called `my_package.functions_to_test.pd.read_csv`