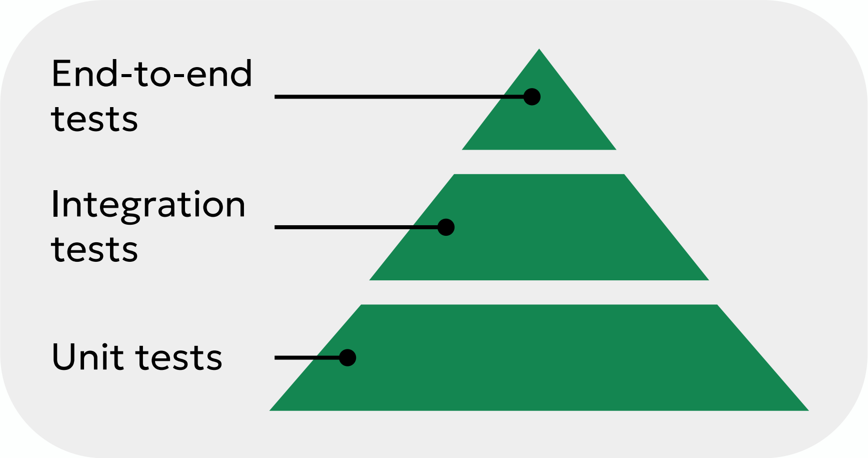
# Software Testing

Kayla Meistrell, Chris Pappelis, Matthew Smith

# Background

- **Software testing** - the process of evaluating whether a program does what it is supposed to do
- Used very often in software development
- Types of tests:
    - Unit tests - components tested in isolation
    - Integration tests - components tested in groups
    - End-to-end tests - system tested as a whole


End-to-end tests

Integration tests

Unit tests

# Motivation

- Write more robust, testable code
- Save time, money, and effort by fixing bugs early
  - Poorly written software cost the U.S. at least $2.4 trillion in 2022[1]
- Improve user experience
- Ensure critical software is safe

[1] H. Krasner, “Cost of Poor Software Quality in the U.S.: A 2022 Report,” Consortium for Information and Software Quality, Dec. 2022. [Online]. Available: https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/

# Outline

1. Setting up pytest
2. Walkthrough:
   - Creating unit tests
   - Parameterizing tests
   - Using fixtures
3. Activities
   - Writing your own unit tests
   - Writing your own tests with fixtures
   - Testing multiple interacting functions
4. Exercises
   - Locating bugs using testing
   - Test-driven development
   - Debugging using testing

# Installing pytest

1. If you don't have Python installed, download it here:
   ```
   https://www.python.org
   /downloads/
   ```
2. Open command line and clone git repository:
   ```
   git clone https://github.com
   /msmithp/software-testing/
   ```
3. Navigate to folder:
   ```
   cd software-testing
   ```
4. Create a virtual environment:
   ```
   python -m venv env
   ```
5. Activate Python virtual environment in command line:
   Windows:
   ```
   env\Scripts\activate
   ```
   MacOS:
   ```
   source env/bin/activate
   ```
6. Install pytest:
   ```
   pip install pytest
   ```

# Walkthrough: Creating a Unit Test

1. Navigate to the `walkthrough` folder:
   `cd walkthrough`

2. Open `test_functions.py` in an IDE or text editor

3. Write a pytest function in `test_functions.py` to test `add()` (Note that pytest test functions **must** start with `test`):

```
def test_add():
    assert add(2, 2) == 4, "Should be 4"
```

`assert add(2, 2) == 4,` — assertion that some Boolean expression is true

`"Should be 4"` — optional message to display if test fails

4. With virtual environment active, run test in command line:
   `pytest test_functions.py`

# Walkthrough: Creating a Failing Test

1. In `test_functions.py`, write another test:

```
def test_add_failing():
    assert add(3, 5) == 7, "Should be 7"
```

2. Run tests in command line:

```
pytest test_functions.py
```

3. This test will fail

# Walkthrough: Parameterized Tests

Using multiple inputs and outputs with the same test

1. In `test_functions.py`, define a list of tuples `(a, b, c)` such that `a + b = c`:
   ```
   test_cases = [(1, 1, 2), (-1, -2, -3),
                 (0, -5, -5), (5, -2, 3)]
   ```

2. Create a parameterized* test that tests many cases in one function:
   ```
   @pytest.mark.parametrize("a, b, expected", test_cases)
   def test_add_many(a, b, expected):
       assert add(a, b) == expected
   ```
   *Note that pytest spells it as *parametrize* rather than *parameterize*.

3. Run tests in the command line:
   ```
   pytest test_functions.py
   ```

4. In the output, each case in `test_cases` will be counted as its own test

# Walkthrough: Using Fixtures

Reusing the same data in multiple tests

1. Open `test_rational.py` in an IDE or text editor

2. Create a fixture:

   ```
   @pytest.fixture
   def example_rational():
       return Rational(1, 3)
   ```

3. Write multiple tests that use the fixture:

   ```
   def test_multiply(example_rational):
       assert example_rational * 2 == Rational(2, 3)

   def test_add(example_rational):
       assert example_rational + Rational(2, 3) == Rational(1, 1)
   ```

4. Run test in command line:

   ```
   pytest test_rational.py
   ```

# Activity 1: Basic Unit Test

1. Navigate to the `activity1` folder
2. Create functions to test `calc_grades()`
3. Test the function with multiple different grade percentages in order to determine whether or not there is an error in the code

| Input | Expected Output |
|---|---|
| `x >= 90.0` | “A” |
| `90.0 > x >= 80.0` | “B” |
| `80.0 > x >= 70.0` | “C” |
| `70.0 > x >= 60.0` | “D” |
| `60.0 > x` | “F” |
| `otherwise` | “Not a valid grade!” |

# Activity 2: Fixtures

1. Navigate to the `activity2` folder.

2. Create a fixture to initialize a queue with some data:

   ```
   @pytest.fixture
   def example_queue():
       queue = Queue()
       queue.enqueue(1)
       queue.enqueue(2)
       ...
       return queue
   ```

3. Write tests in `test_queue.py` for the functions given in the table to the right. Are there any bugs?

| Function | Input | Expected Output |
|---|---|---|
| `enqueue()` | Item to be queued | None |
| `dequeue()` | None | First item in queue, throws `Exception` if empty |
| `is_empty()` | None | `True` if empty, `False` otherwise |
| `size()` | None | Length of queue |
| `to_string()` | None | String of items delimited by spaces (e.g., “`1 2 3`”); “” if empty |

# Activity 3: Testing Interacting Functions

1. Navigate to `activity3` folder
2. Create unit tests for the `Triangle` Class functions in `test_triangle.py`
3. Some notes: `perimeter()` relies on `pythagorean()`, and you may need to use the provided `equals_float()` function to compare floating-point values.

| Function | Formula | Input | Expected Output |
|---|---|---|---|
| `area()` | `0.5 * base * height` | `base >= 0`<br>`height >= 0` | Area |
| `pythagorean()` | `sqrt(base^2 * height^2)` | `base >= 0`<br>`height >= 0` | Hypotenuse |
| `perimeter()` | `base + height + sqrt(base^2 * height^2)` | `base >= 0`<br>`height >= 0`<br>`hypotenuse >= 0` | Perimeter |

# Exercise 1: Find the Broken Function

You are tasked with using tests to determine the broken function in a pipeline that processes a list of dictionaries containing movie data. Out of all the functions given, one is causing an error. Use the given list of dictionaries to find and fix the error.

1. Navigate to the `exercise1` folder.
2. Write tests that test each of the functions.
3. Determine where the error is and fix it.

| Function | Input | Expected Output |
|---|---|---|
| `get_avg()` | List of movie dictionaries | The average revenue of all movies, rounded to the nearest million, as a string like “`$x.00`” |
| `get_min()` | List of movie dictionaries | The movie title that has the lowest revenue, and its revenue, formatted as a string like “`Title: <title>, Revenue: $x.00`” |
| `get_max()` | List of movie dictionaries | The movie title that has the highest revenue, and its revenue, formatted as a string like “`Title: <title>, Revenue: $x.00`” |
| `rounding()` | Integer | The integer rounded to the nearest million. |
| `format_revenue()` | Integer | Revenue formatted as a string in dollars, like “`$x.00`” |
| `format_movie()` | String and integer | Title and revenue formatted as a string, like “`Title: <title>, Revenue: $x.00`” |

# Exercise 2: Test-Driven Development

You are tasked with writing a function that will check the validity of a user's password. It will take a string as input and return an **ordered list** of violations, denoted as error codes. The violations are listed in the table to the right. Examples:

```
“pass” → [1, 2, 3]
“qwertyuiop” → [2]
“softwareTesting” → []
```

1. Navigate to the `exercise2` folder.
2. First, write tests that test different combinations of these cases.
3. Then, implement the function and use the tests to ensure that it works. If any test fails, modify your function.

| Violation | Error Code |
|---|---|
| Too short (<8 characters long) | **1** |
| Needs an uppercase character | **2** |
| Cannot use the same character twice in a row | **3** |
| Cannot be “`password`” | **4** |

# Exercise 3: Broken Clock Class

You are tasked with writing a test that will ensure the functionality of a clock. This simple 24-hour clock class will allow you to set the time and add time to progress the clock. For example, adding `03:40` to `10:00` should return `13:40`, and adding `01:30` to `23:00` should return `00:30`.

1. Navigate to `exercise3` folder
2. Create fixture(s) for the Clock class
3. Create tests to test the Clock class
4. Can you find the error? If so, fix the program error. *(Hint: it may be helpful to use the string representation of the clock)*

# Glossary

## Key terms

**Unit testing** - Class of software testing that tests individual components (e.g., functions, classes) in isolation. Typically the simplest and most numerous of tests.

**Integration testing** - Class of software testing in which multiple components of a system are tested together. In between unit testing and end-to-end testing in complexity.

**End-to-end/system testing** - Class of software testing that simulates a user's experience to test the functionality of the system as a whole. Typically the most complex and least numerous of tests.

**Test case** - An individual test which checks for a specific output given a particular set of inputs

**Test suite** - A collection of test cases that are executed together

**Fixture** - An initialization that provides a consistent context in which tests can run in order to ensure repeatable results

**Parameterized test** - A single test that tests multiple inputs and outputs using parameters

# Glossary

Other important terms

**Black-box testing** - Form of testing that focuses only on the inputs and outputs of a system without consideration of its implementation

**White-box testing** - Form of testing where the tester has access to source code and implementation details of the system to ensure that internal code of the software is correct

**Regression testing** - Type of testing that runs after every change to ensure that new additions have not broken existing code

**Functional testing** - Testing that ensures that the system does what it is designed to do (i.e., testing functional requirements)

**Non-functional testing** - Testing that focuses on performance, reliability, security, and other non-functional attributes

**Test-driven development** - Software development methodology wherein tests are written for expected features first, then code is written to pass these tests

# Pytest Cheat Sheet

## Defining a test:

```
def test_example():
     assert <bool_expr>, <message>
```

## Parameterizing a test:

```
test_cases = [(1, 1, 2), (-1, -2, -3),
              (0, -5, -5), (5, -2, 3)]

@pytest.mark.parametrize(
     "a, b, expected", test_cases)
def test_add_many(a, b, expected):
    assert add(a, b) == expected
```

## Defining and using a fixture:

```
@pytest.fixture
def example_fixture():
     return “example”

def test_fixture(example_fixture):
     assert example_fixture == “example”
```

## Running a test file:

```
pytest <filename>
```

Or, to discover test files automatically:

```
pytest
```

# Appendix A: Technical Tricks

- By using the command `pytest` with no arguments, pytest will automatically discover and run all tests in files named `test_*.py` or `*_test.py` in the current directory and all subdirectories.

- You can assert that a function will raise an exception as follows:

  ```
  def test_division():
      with pytest.raises(
          ZeroDivisionError):
           divide(1, 0)
  ```

  This test will pass only if `divide(1, 0)` raises a ZeroDivsionError.

- You can use `yield` instead of `return` in a fixture if you need to perform teardown after creating a fixture:

  ```
  @pytest.fixture
  def user():
      user = create_user()
      yield user
      delete_user(user)
  ```

# Appendix A: Technical Tricks

- You can test output to stdout (e.g., a print statement) using the built-in `capfd` fixture from pytest:

  ```
  def test_output(capfd):
      print("hello")
      captured = capfd.readouterr()
      assert captured.out == "hello\n"
  ```

  You do not need to manually declare the `capfd` fixture.

- You can print to standard output during a test, and pytest will automatically create a section in that test's output to show the printed contents.

- You can organize related tests into classes (class name must start with `Test`):

  ```
  class TestMath():
      def test_add(self):
          assert 1 + 1 == 2
      def test_multiply(self):
          assert 6 * 3 == 18
  ```

# Appendix B: Questions

**Questions?**