Lecture 5: Verification

CEE 690

Verification

Does my code do what I think it is doing?

What never to do

What never to do

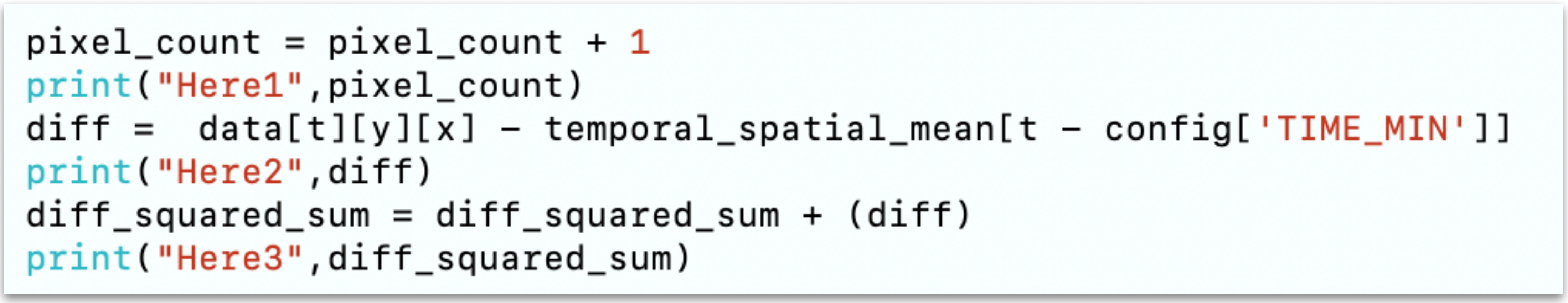
The code runs… so it works!!!

What you probably will do

It runs but the results are either weird or a bunch of NaNs. So….

What you probably will do

It runs but the results are either weird or a bunch of NaNs. So….



```
pixel_count = pixel_count + 1
print("Here1",pixel_count)
diff =  data[t][y][x] - temporal_spatial_mean[t - config['TIME_MIN']]
print("Here2",diff)
diff_squared_sum = diff_squared_sum + (diff)
print("Here3",diff_squared_sum)
```

What you should do

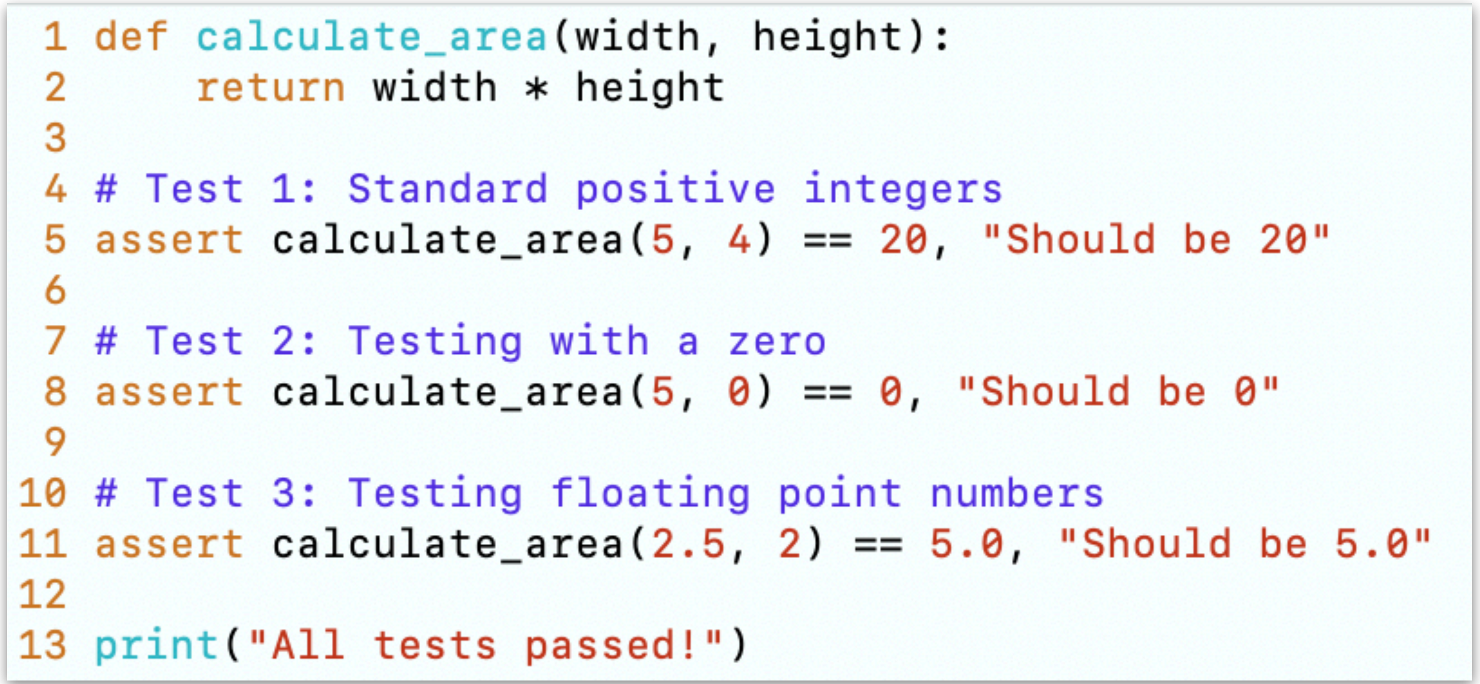
What you should do

Unit tests

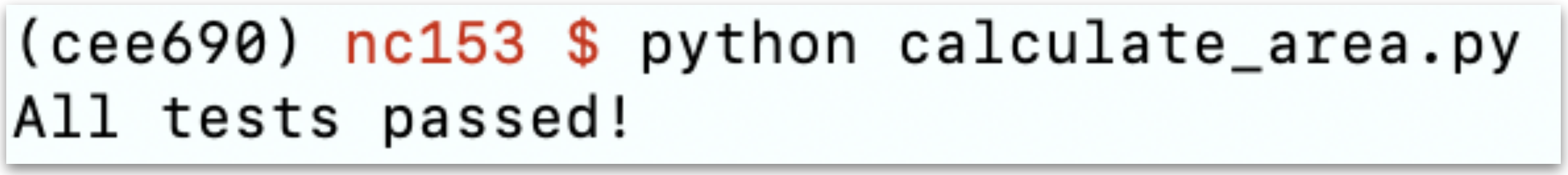
Small data testing

Never test a large piece of code as an entire piece of software. Instead we should test parts.

Example: Calculate area

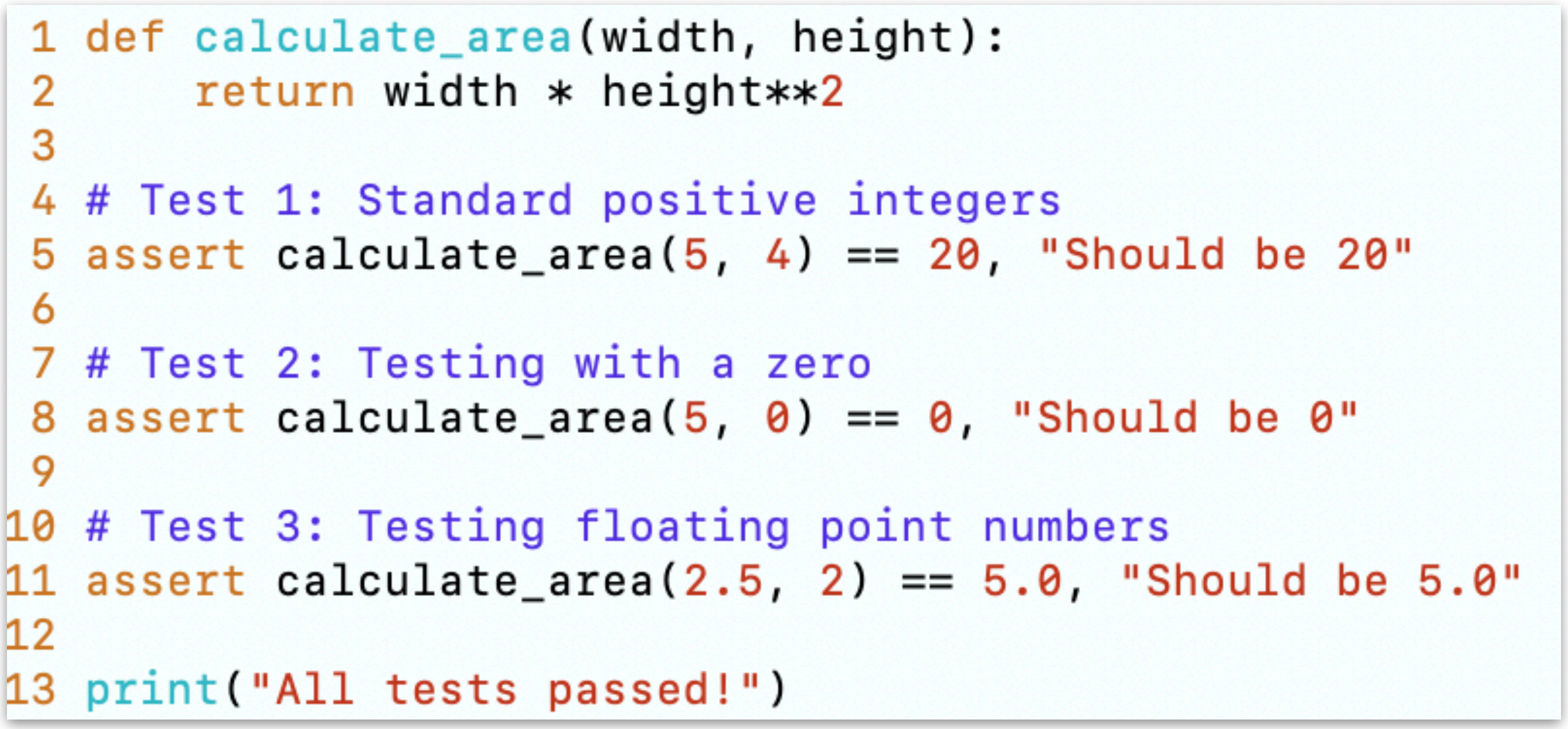


```python
def calculate_area(width, height):
    return width * height

# Test 1: Standard positive integers
assert calculate_area(5, 4) == 20, "Should be 20"

# Test 2: Testing with a zero
assert calculate_area(5, 0) == 0, "Should be 0"

# Test 3: Testing floating point numbers
assert calculate_area(2.5, 2) == 5.0, "Should be 5.0"

print("All tests passed!")
```

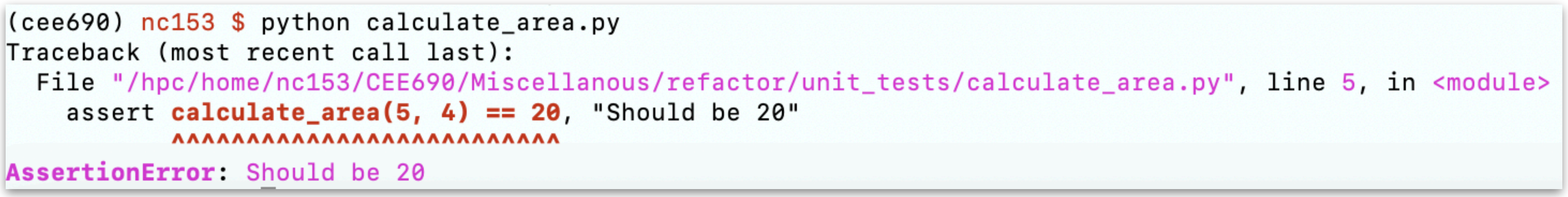


```
(cee690) nc153 $ python calculate_area.py
All tests passed!
```

Example: Calculate area

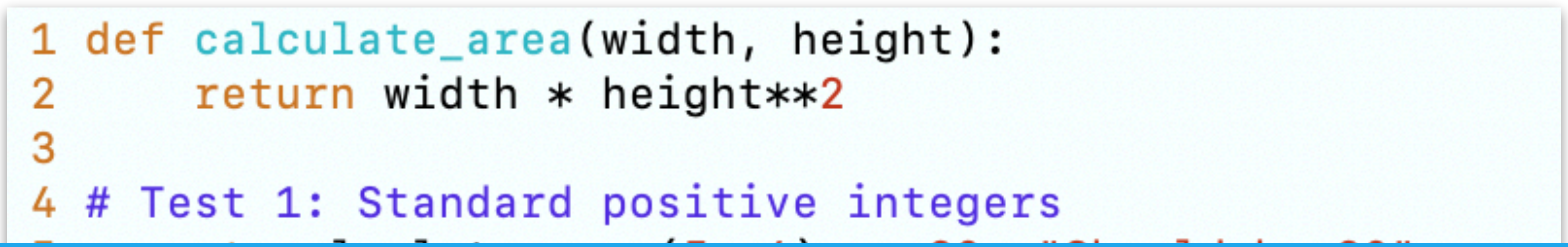


```python
def calculate_area(width, height):
    return width * height**2

# Test 1: Standard positive integers
assert calculate_area(5, 4) == 20, "Should be 20"

# Test 2: Testing with a zero
assert calculate_area(5, 0) == 0, "Should be 0"

# Test 3: Testing floating point numbers
assert calculate_area(2.5, 2) == 5.0, "Should be 5.0"

print("All tests passed!")
```



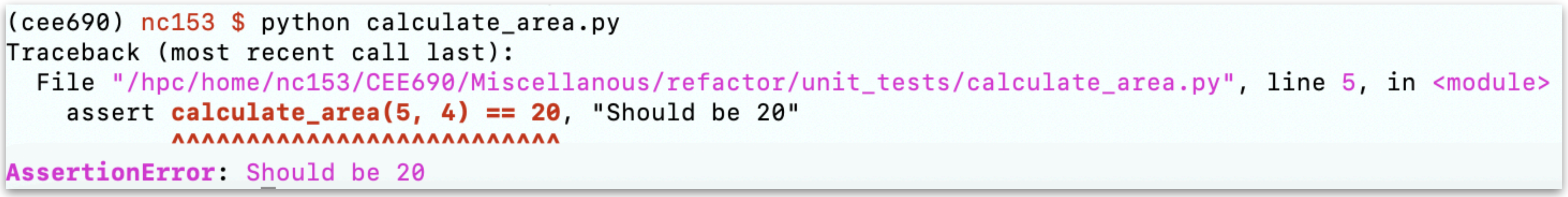
```
(cee690) nc153 $ python calculate_area.py
Traceback (most recent call last):
  File "/hpc/home/nc153/CEE690/Miscellanous/refactor/unit_tests/calculate_area.py", line 5, in <module>
    assert calculate_area(5, 4) == 20, "Should be 20"
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: Should be 20
```

Example: Calculate area



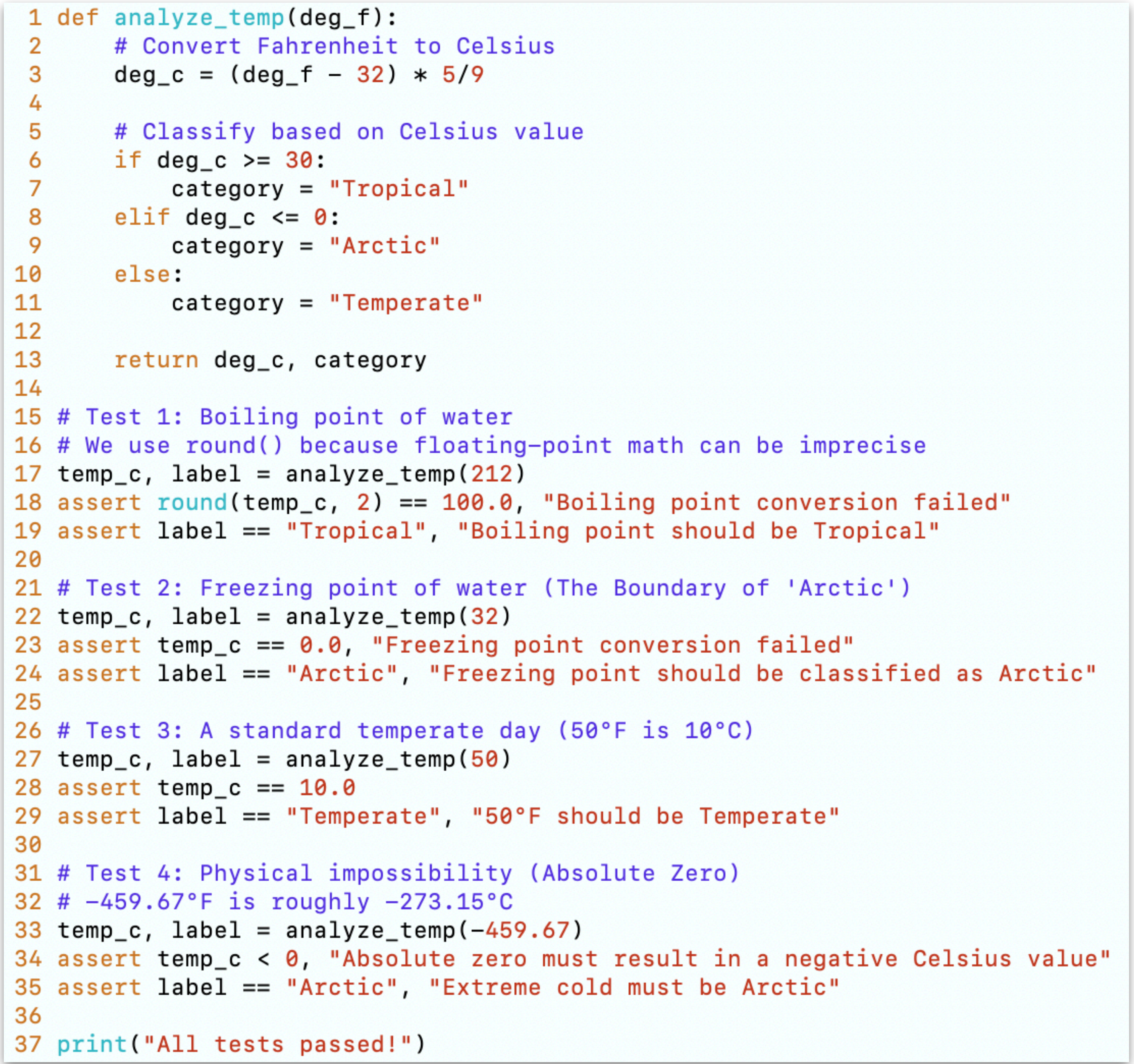
```python
def calculate_area(width, height):
    return width * height**2

# Test 1: Standard positive integers
```

If you already know what the behavior of your function should be (and you are not purposefully changing what the function is supposed to do) then it should always pass the same checks



```
(cee690) nc153 $ python calculate_area.py
Traceback (most recent call last):
  File "/hpc/home/nc153/CEE690/Miscellanous/refactor/unit_tests/calculate_area.py", line 5, in <module>
    assert calculate_area(5, 4) == 20, "Should be 20"
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: Should be 20
```

Example: Analyze temperature



```python
def analyze_temp(deg_f):
    # Convert Fahrenheit to Celsius
    deg_c = (deg_f - 32) * 5/9

    # Classify based on Celsius value
    if deg_c >= 30:
        category = "Tropical"
    elif deg_c <= 0:
        category = "Arctic"
    else:
        category = "Temperate"

    return deg_c, category

# Test 1: Boiling point of water
# We use round() because floating-point math can be imprecise
temp_c, label = analyze_temp(212)
assert round(temp_c, 2) == 100.0, "Boiling point conversion failed"
assert label == "Tropical", "Boiling point should be Tropical"

# Test 2: Freezing point of water (The Boundary of 'Arctic')
temp_c, label = analyze_temp(32)
assert temp_c == 0.0, "Freezing point conversion failed"
assert label == "Arctic", "Freezing point should be classified as Arctic"

# Test 3: A standard temperate day (50°F is 10°C)
temp_c, label = analyze_temp(50)
assert temp_c == 10.0
assert label == "Temperate", "50°F should be Temperate"

# Test 4: Physical impossibility (Absolute Zero)
# -459.67°F is roughly -273.15°C
temp_c, label = analyze_temp(-459.67)
assert temp_c < 0, "Absolute zero must result in a negative Celsius value"
assert label == "Arctic", "Extreme cold must be Arctic"

print("All tests passed!")
```

“Assert” give us the idea but is not the WAY to do this.

Unit test

Program that verifies that a given “unit” of your software (e.g., functions or class methods) performs the task as expected

What does a unit test look like?

- **Arrange** - Setup data/objects (e.g., mock input and known output)
- **Act** - Run the given function with the mock data
- **Assert** - Verify the outcome

What defines a good unit test?

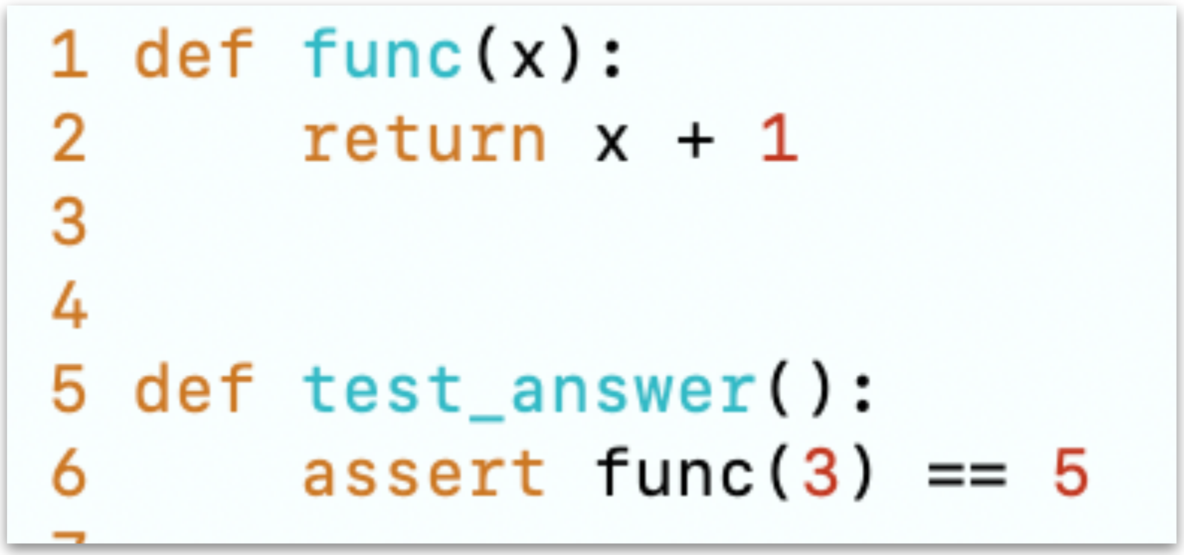
- **Isolated** - It only tests one or a very small number of things at a time
- **Deterministic** - If you prescribe a given input, you will always get the same output
- **Fast** - It needs to be nearly instantaneous; if not, no one will want to run them

What defines a good unit test?

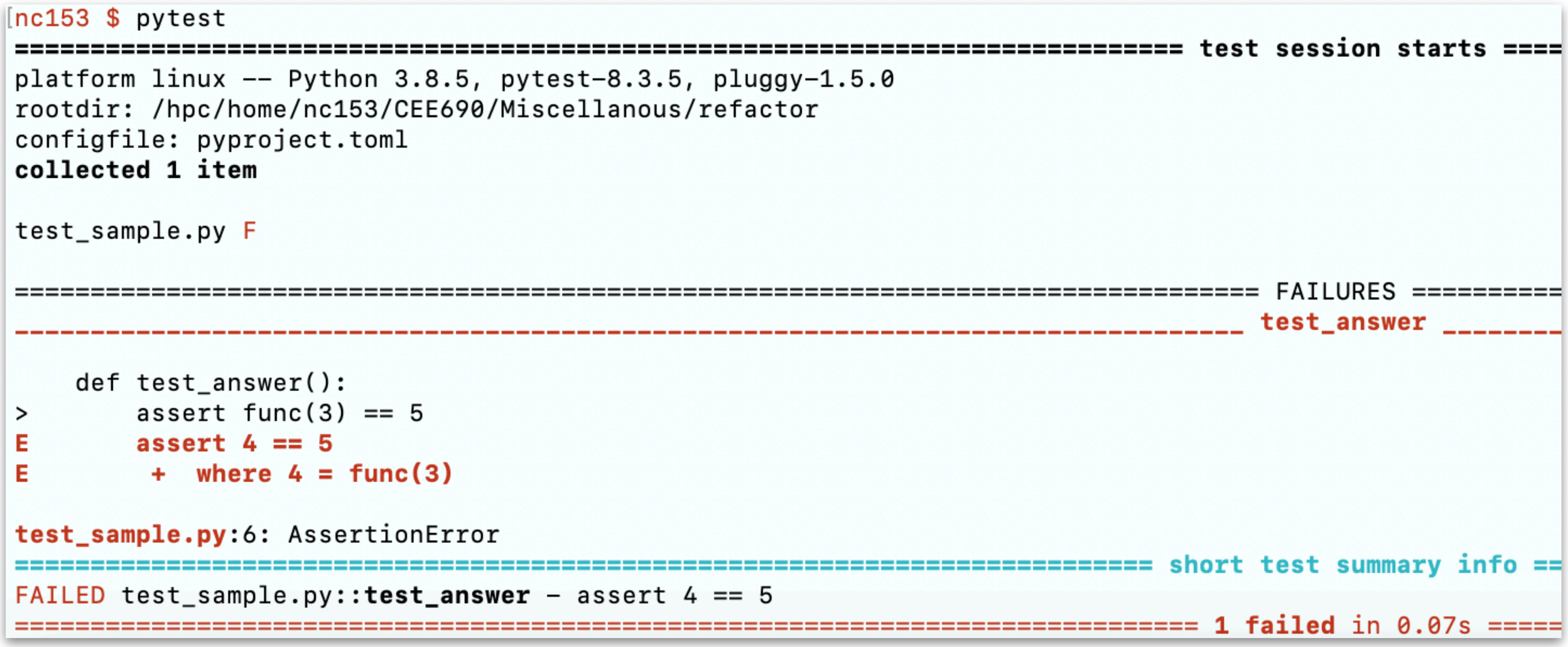
- **Isolated** - It only tests one or a very small number of things at a time
- **Deterministic** - If you prescribe a given input, you will always get the same output
- **Fast** - It needs to be nearly instantaneous; if not, no one will want to run them

pytest

Pytest example

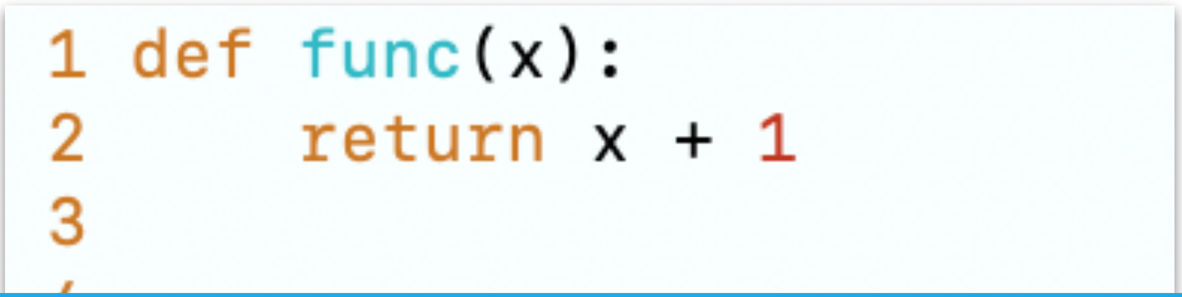


```python
def func(x):
    return x + 1


def test_answer():
    assert func(3) == 5
```



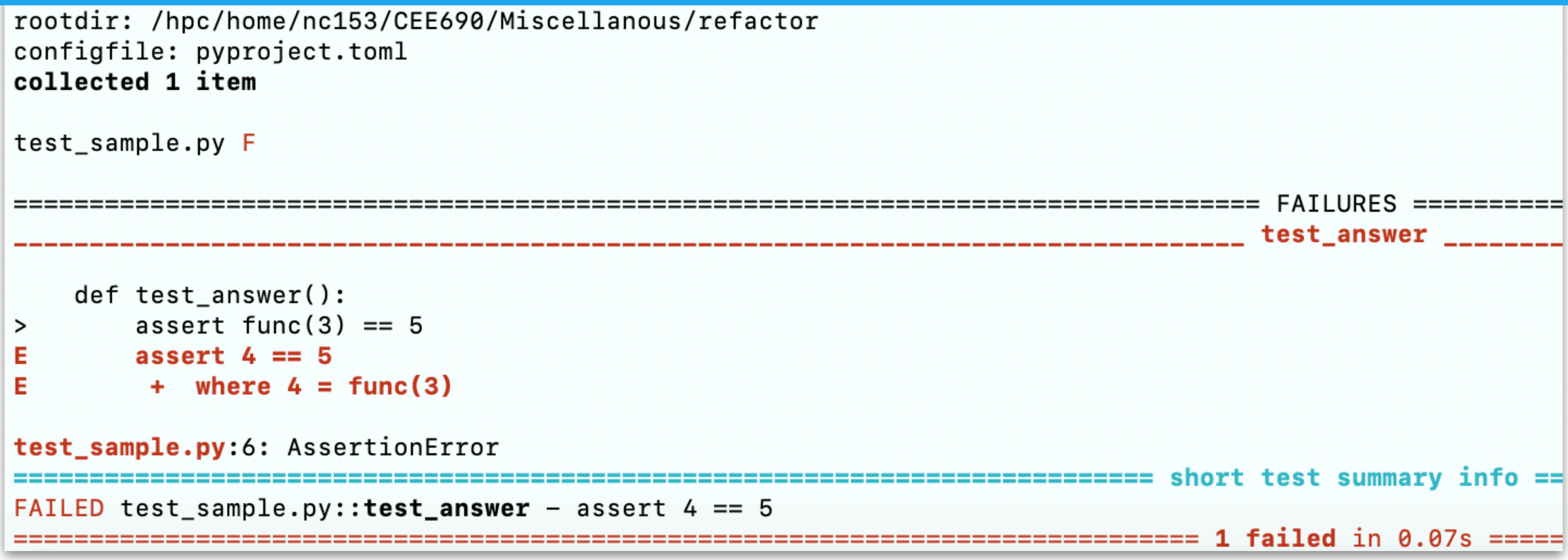
```
[nc153 $ pytest
============================================================================ test session starts ====
platform linux -- Python 3.8.5, pytest-8.3.5, pluggy-1.5.0
rootdir: /hpc/home/nc153/CEE690/Miscellanous/refactor
configfile: pyproject.toml
collected 1 item

test_sample.py F

================================================================================= FAILURES ==========
________________________________________________________________________________ test_answer ________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:6: AssertionError
=========================================================================== short test summary info ==
FAILED test_sample.py::test_answer - assert 4 == 5
============================================================================== 1 failed in 0.07s =====
```

Pytest example



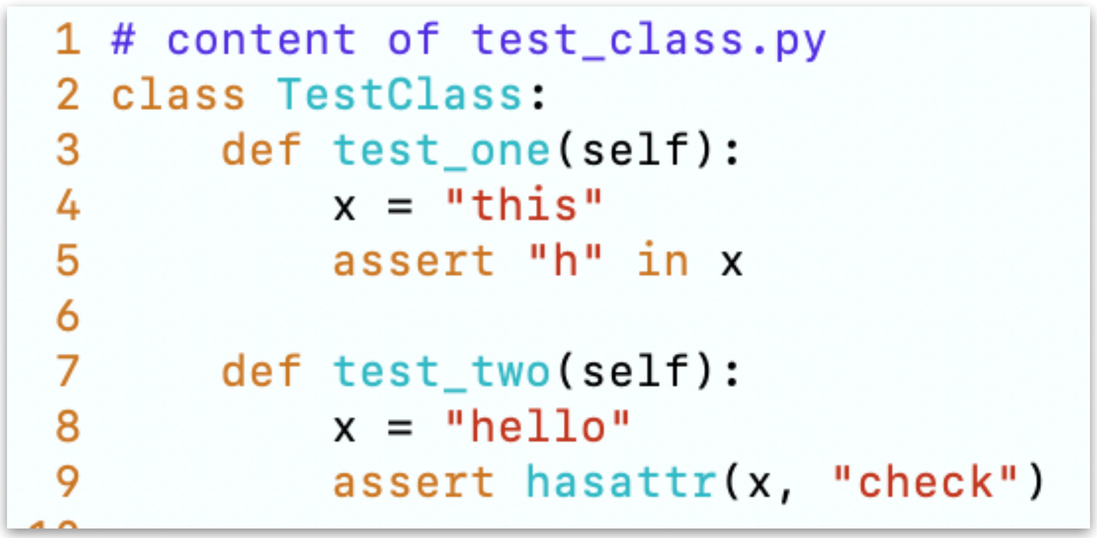
```python
def func(x):
    return x + 1

```

Pytest can run all test_*.py or *_test.py files in a directory and its subdirectories

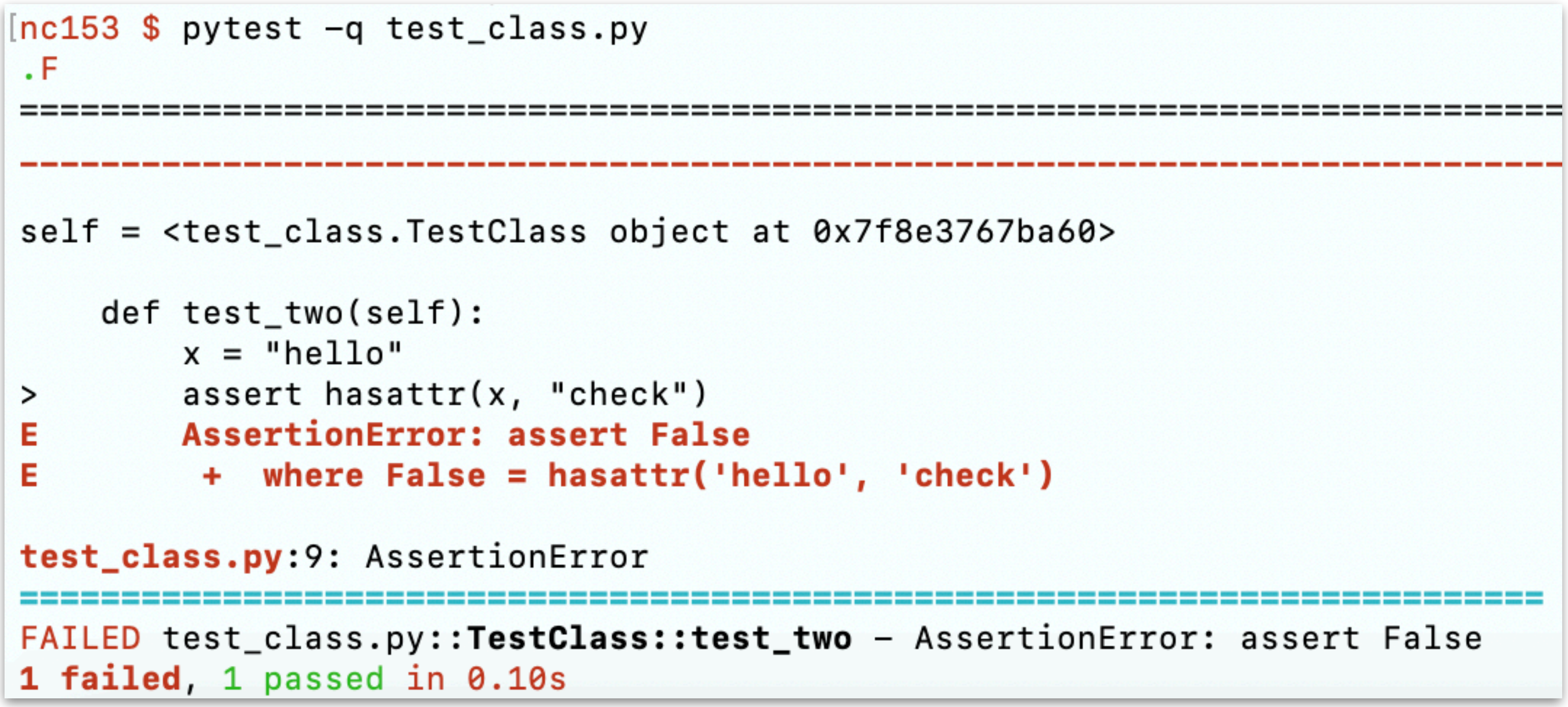


```
rootdir: /hpc/home/nc153/CEE690/Miscellanous/refactor
configfile: pyproject.toml
collected 1 item

test_sample.py F

================================================================================ FAILURES ==========
_______________________________________________________________________________ test_answer ________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:6: AssertionError
========================================================================== short test summary info ==
FAILED test_sample.py::test_answer - assert 4 == 5
============================================================================= 1 failed in 0.07s =====
```

Group multiple tests in a class

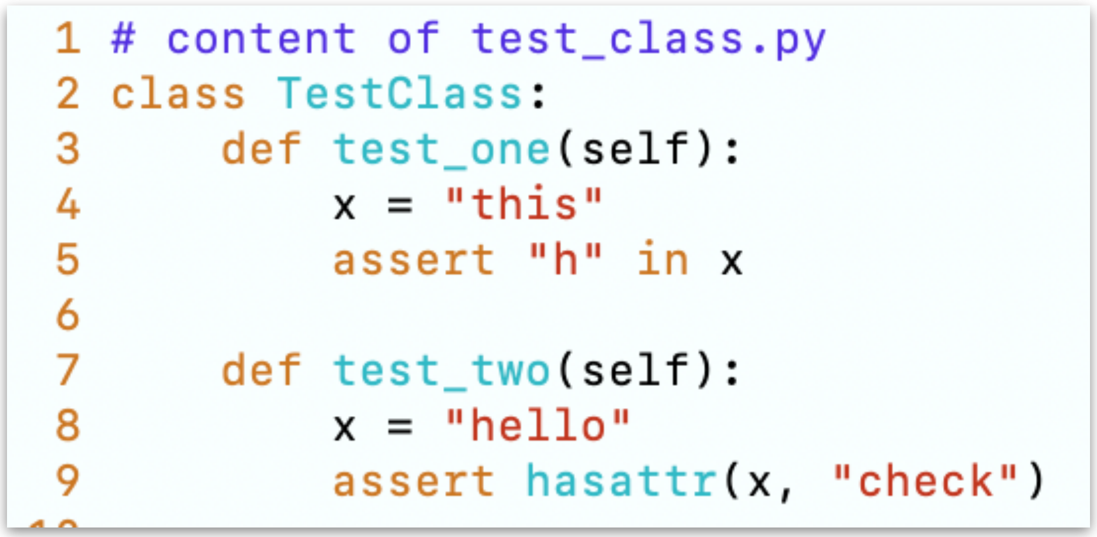


```python
# content of test_class.py
class TestClass:
    def test_one(self):
        x = "this"
        assert "h" in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, "check")
```

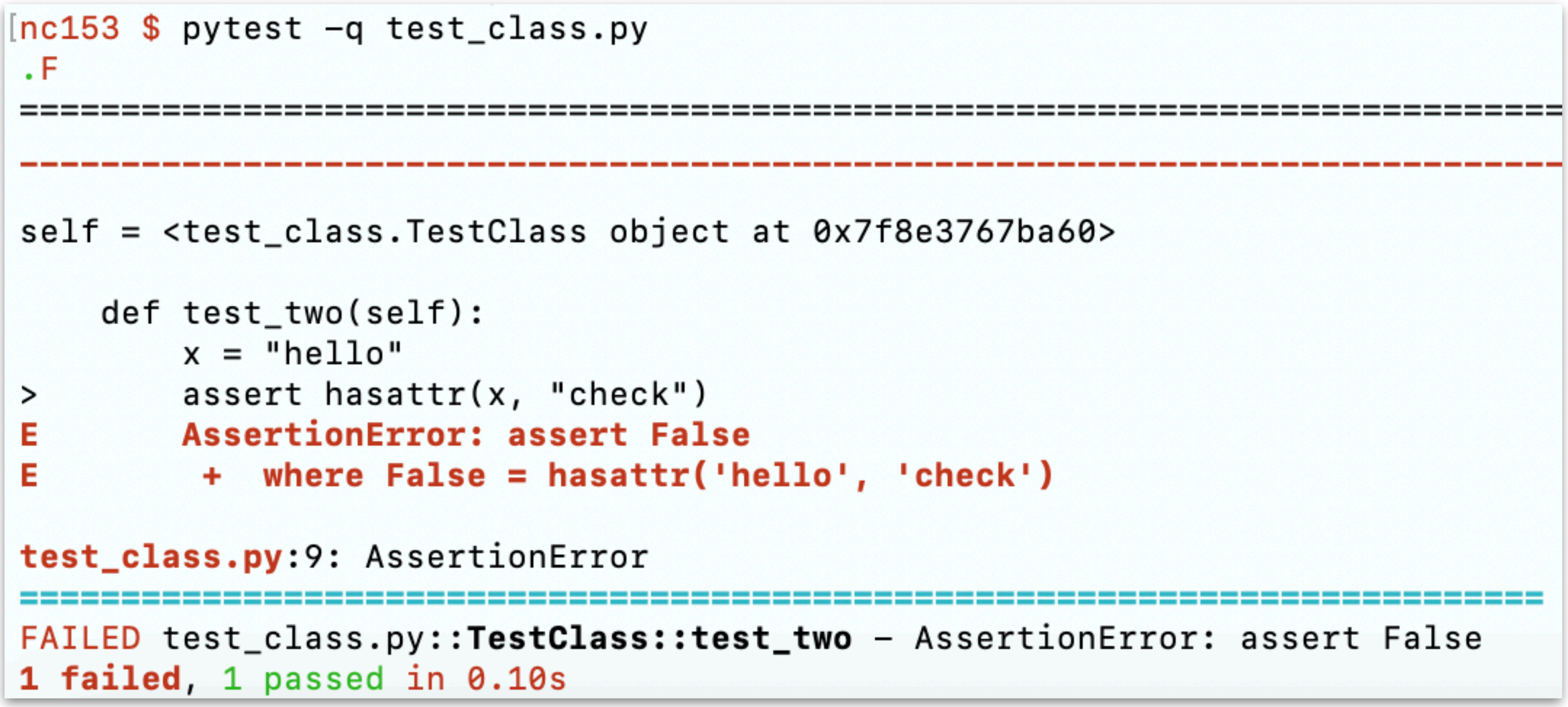


```
nc153 $ pytest -q test_class.py
.F
============================================================================
----------------------------------------------------------------------------

self = <test_class.TestClass object at 0x7f8e3767ba60>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, "check")
E       AssertionError: assert False
E        +  where False = hasattr('hello', 'check')

test_class.py:9: AssertionError
===========================================================================
FAILED test_class.py::TestClass::test_two - AssertionError: assert False
1 failed, 1 passed in 0.10s
```

Group multiple tests in a class

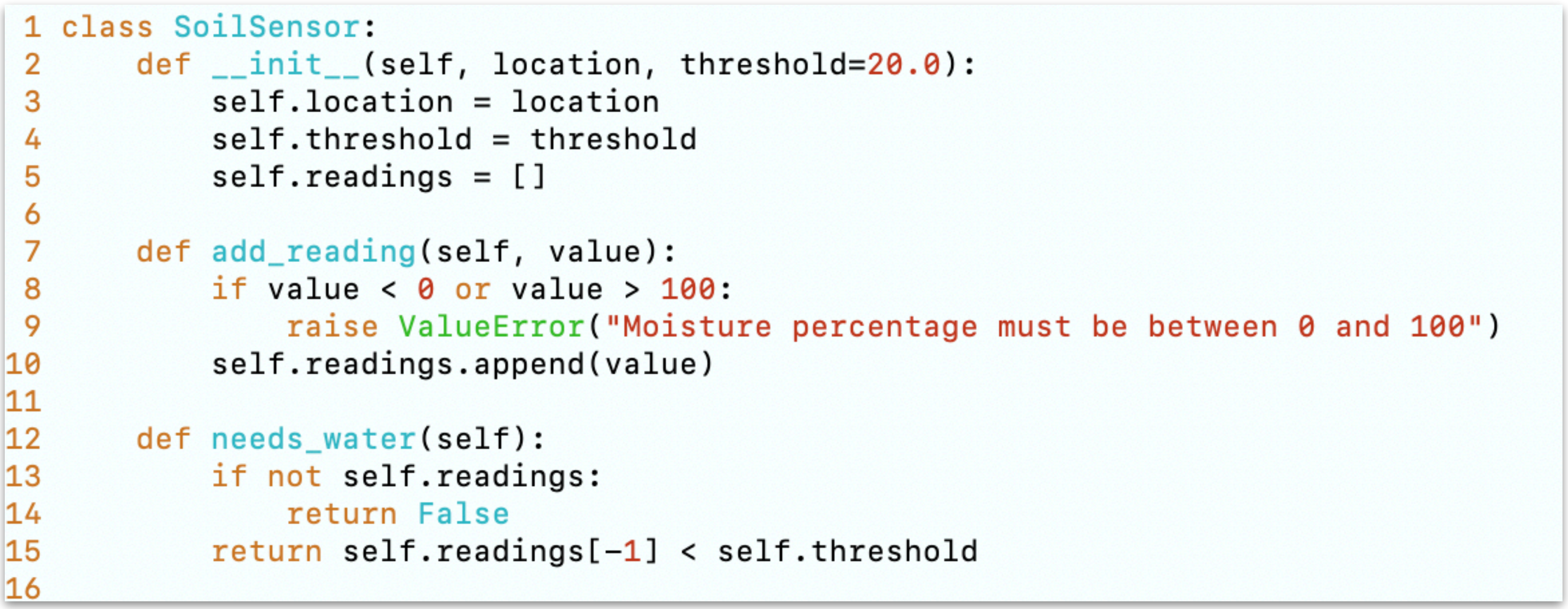


```python
# content of test_class.py
class TestClass:
    def test_one(self):
        x = "this"
        assert "h" in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, "check")
```



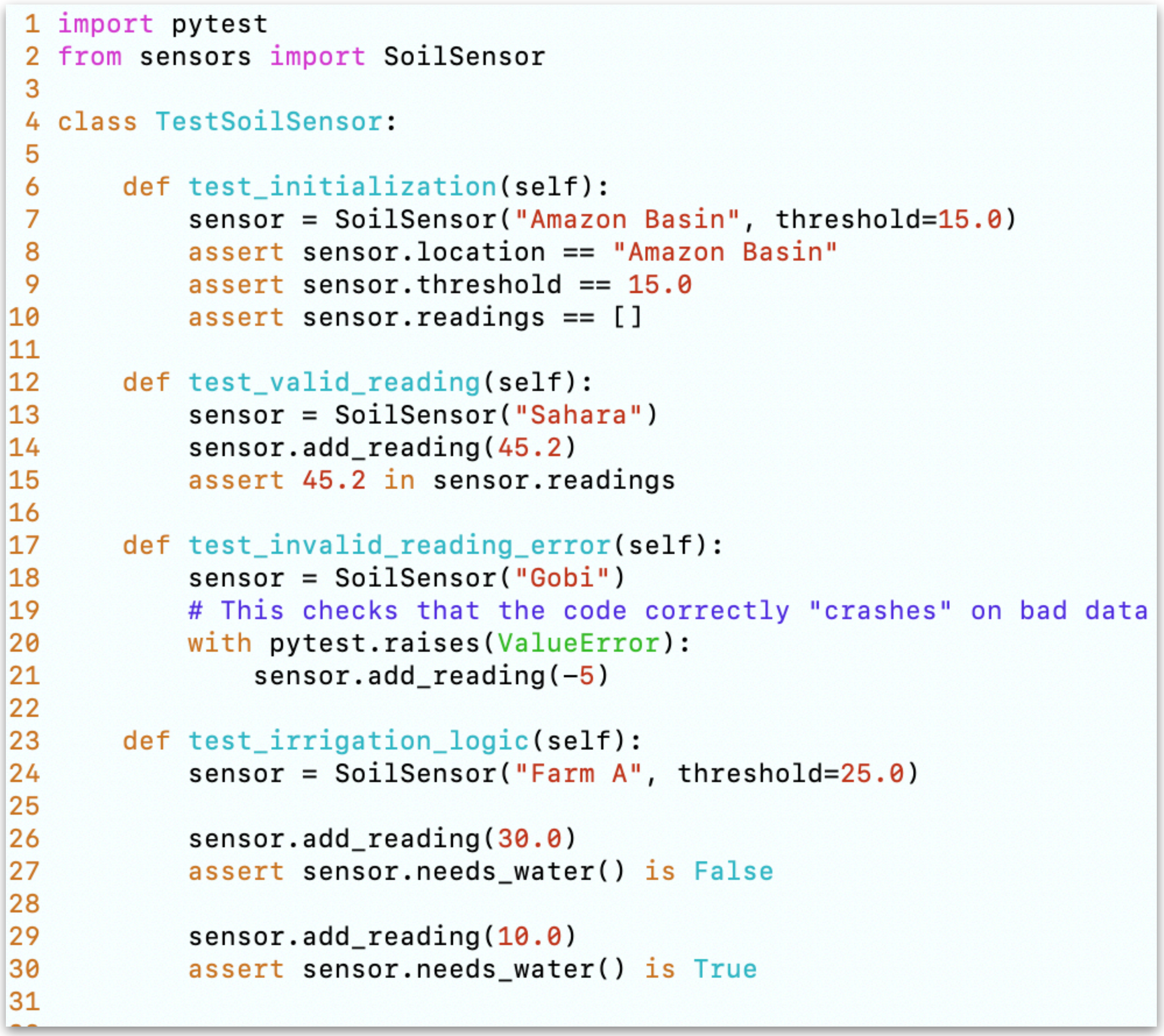
```
[nc153 $ pytest -q test_class.py
.F
============================================================================
----------------------------------------------------------------------------

self = <test_class.TestClass object at 0x7f8e3767ba60>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, "check")
E       AssertionError: assert False
E        +  where False = hasattr('hello', 'check')

test_class.py:9: AssertionError
===========================================================================
FAILED test_class.py::TestClass::test_two - AssertionError: assert False
1 failed, 1 passed in 0.10s
```

Example: Sensors class



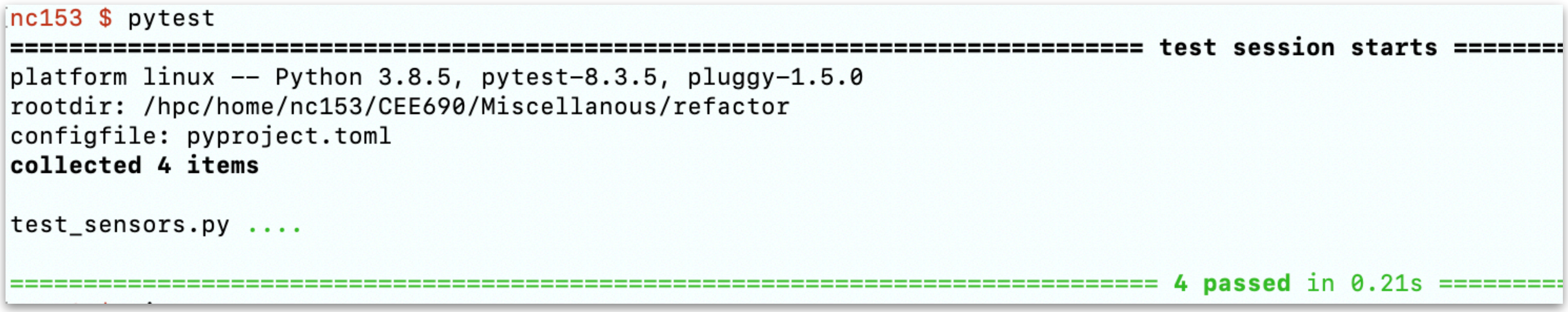
```python
class SoilSensor:
    def __init__(self, location, threshold=20.0):
        self.location = location
        self.threshold = threshold
        self.readings = []

    def add_reading(self, value):
        if value < 0 or value > 100:
            raise ValueError("Moisture percentage must be between 0 and 100")
        self.readings.append(value)

    def needs_water(self):
        if not self.readings:
            return False
        return self.readings[-1] < self.threshold

```

Example: Pytest + sensors



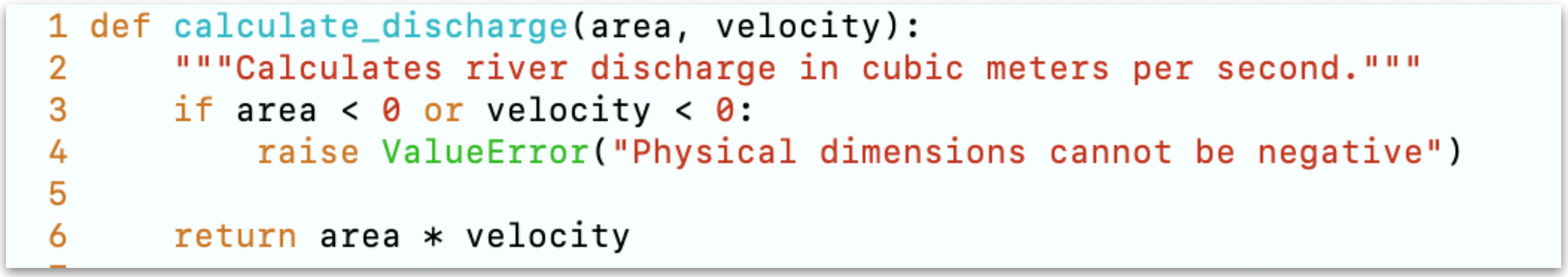
```python
import pytest
from sensors import SoilSensor

class TestSoilSensor:

    def test_initialization(self):
        sensor = SoilSensor("Amazon Basin", threshold=15.0)
        assert sensor.location == "Amazon Basin"
        assert sensor.threshold == 15.0
        assert sensor.readings == []

    def test_valid_reading(self):
        sensor = SoilSensor("Sahara")
        sensor.add_reading(45.2)
        assert 45.2 in sensor.readings

    def test_invalid_reading_error(self):
        sensor = SoilSensor("Gobi")
        # This checks that the code correctly "crashes" on bad data
        with pytest.raises(ValueError):
            sensor.add_reading(-5)

    def test_irrigation_logic(self):
        sensor = SoilSensor("Farm A", threshold=25.0)

        sensor.add_reading(30.0)
        assert sensor.needs_water() is False

        sensor.add_reading(10.0)
        assert sensor.needs_water() is True

```

Both the class and the methods need to start with “Test” and test_ respectively

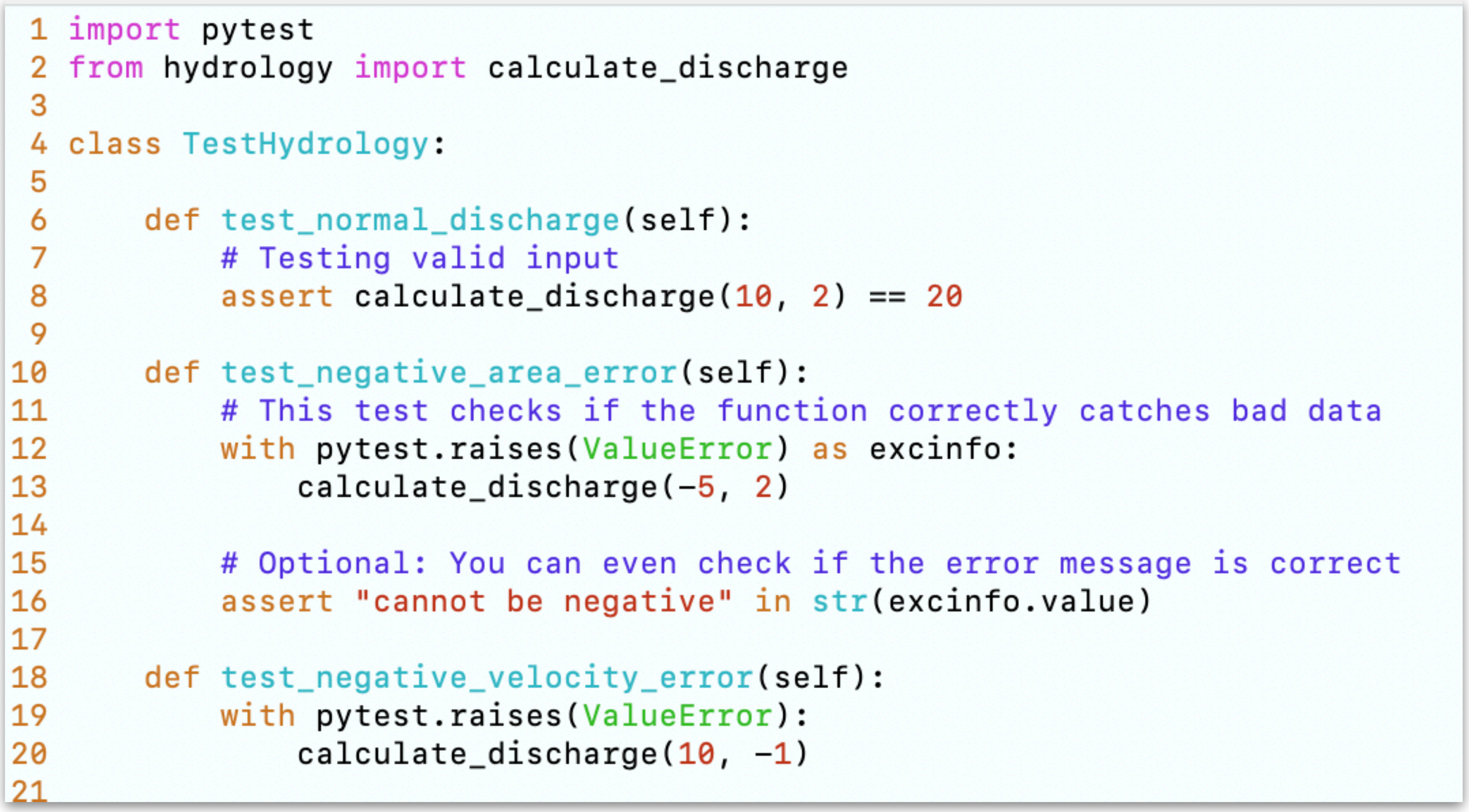


```
nc153 $ pytest
================================================================== test session starts ========
platform linux -- Python 3.8.5, pytest-8.3.5, pluggy-1.5.0
rootdir: /hpc/home/nc153/CEE690/Miscellanous/refactor
configfile: pyproject.toml
collected 4 items

test_sensors.py ....

=================================================================== 4 passed in 0.21s ========
```

Pytest example: Error handling

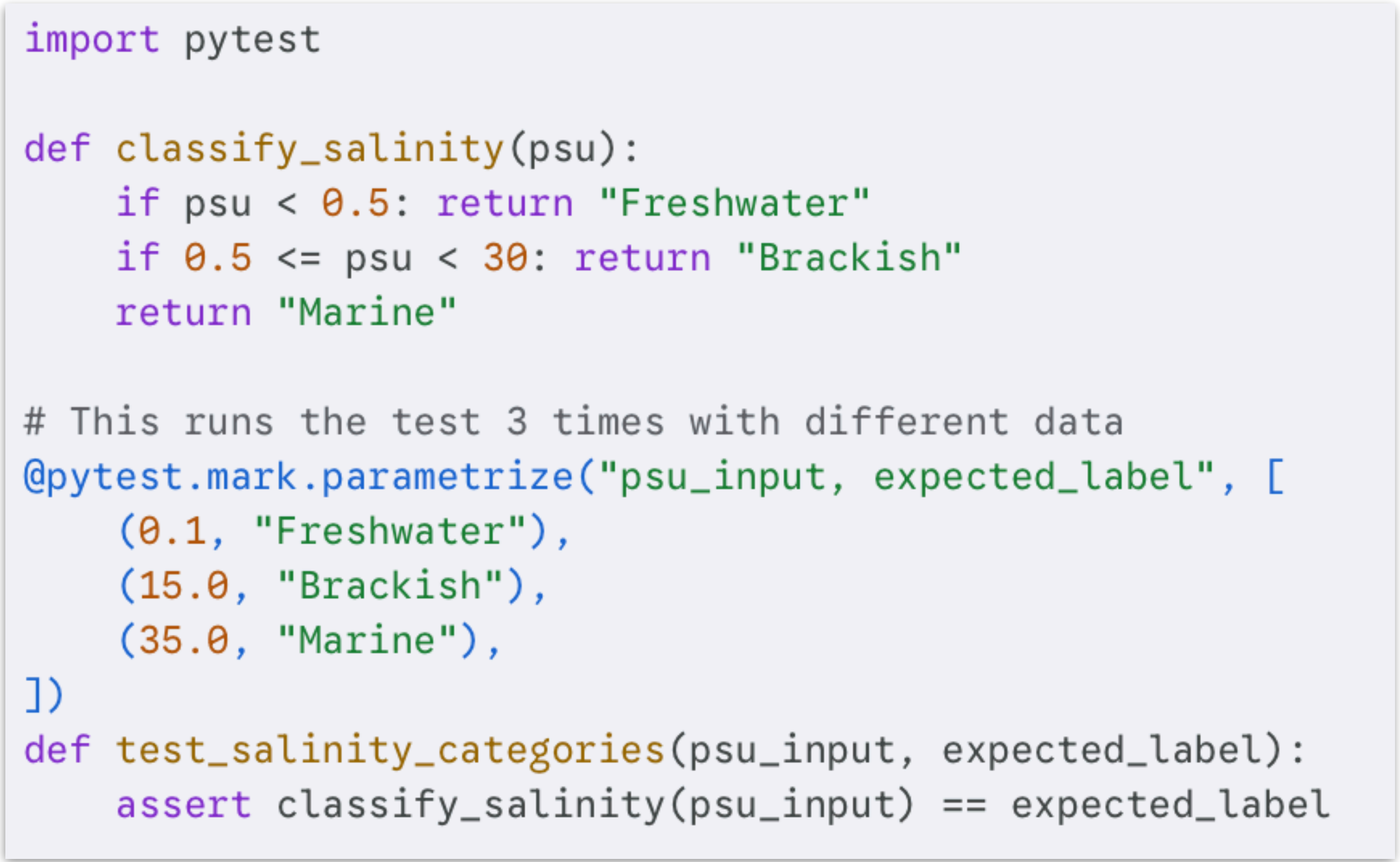


```python
def calculate_discharge(area, velocity):
    """Calculates river discharge in cubic meters per second."""
    if area < 0 or velocity < 0:
        raise ValueError("Physical dimensions cannot be negative")

    return area * velocity
```



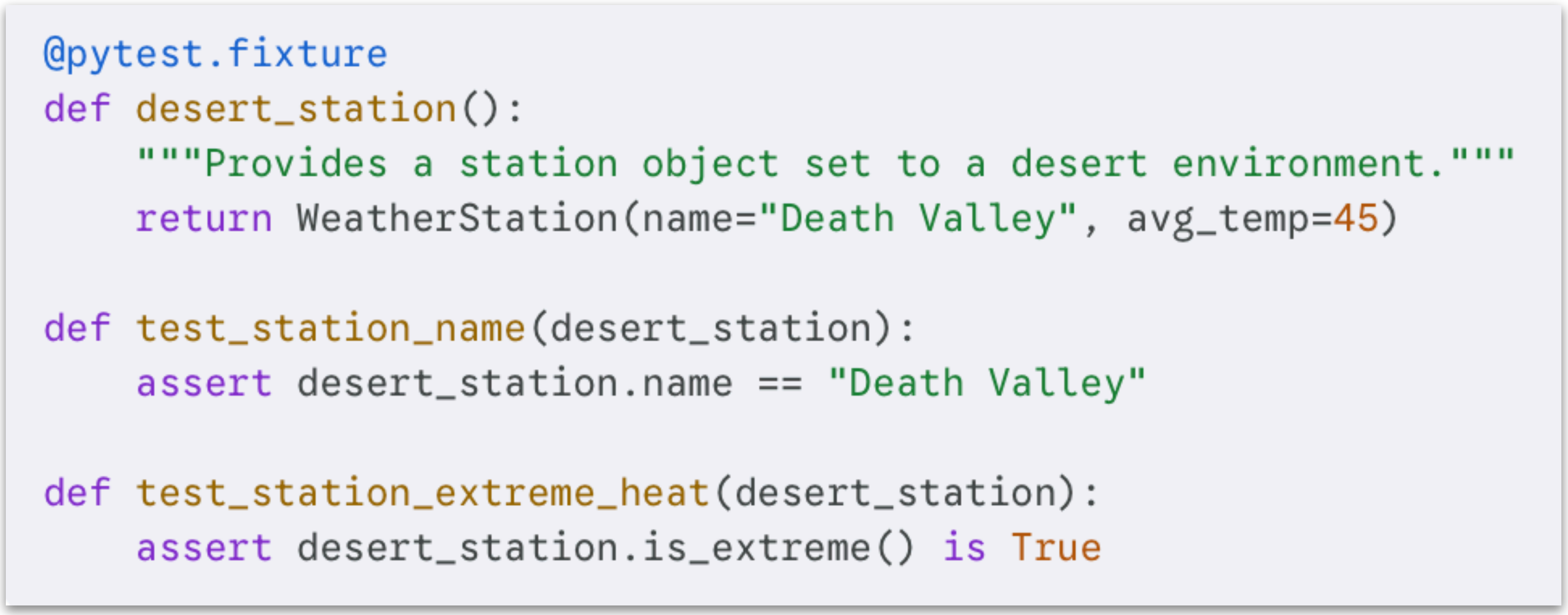
```python
import pytest
from hydrology import calculate_discharge

class TestHydrology:

    def test_normal_discharge(self):
        # Testing valid input
        assert calculate_discharge(10, 2) == 20

    def test_negative_area_error(self):
        # This test checks if the function correctly catches bad data
        with pytest.raises(ValueError) as excinfo:
            calculate_discharge(-5, 2)

        # Optional: You can even check if the error message is correct
        assert "cannot be negative" in str(excinfo.value)

    def test_negative_velocity_error(self):
        with pytest.raises(ValueError):
            calculate_discharge(10, -1)
```

Other pytest features: Multiple datasets at once



```python
import pytest

def classify_salinity(psu):
    if psu < 0.5: return "Freshwater"
    if 0.5 <= psu < 30: return "Brackish"
    return "Marine"

# This runs the test 3 times with different data
@pytest.mark.parametrize("psu_input, expected_label", [
    (0.1, "Freshwater"),
    (15.0, "Brackish"),
    (35.0, "Marine"),
])
def test_salinity_categories(psu_input, expected_label):
    assert classify_salinity(psu_input) == expected_label
```

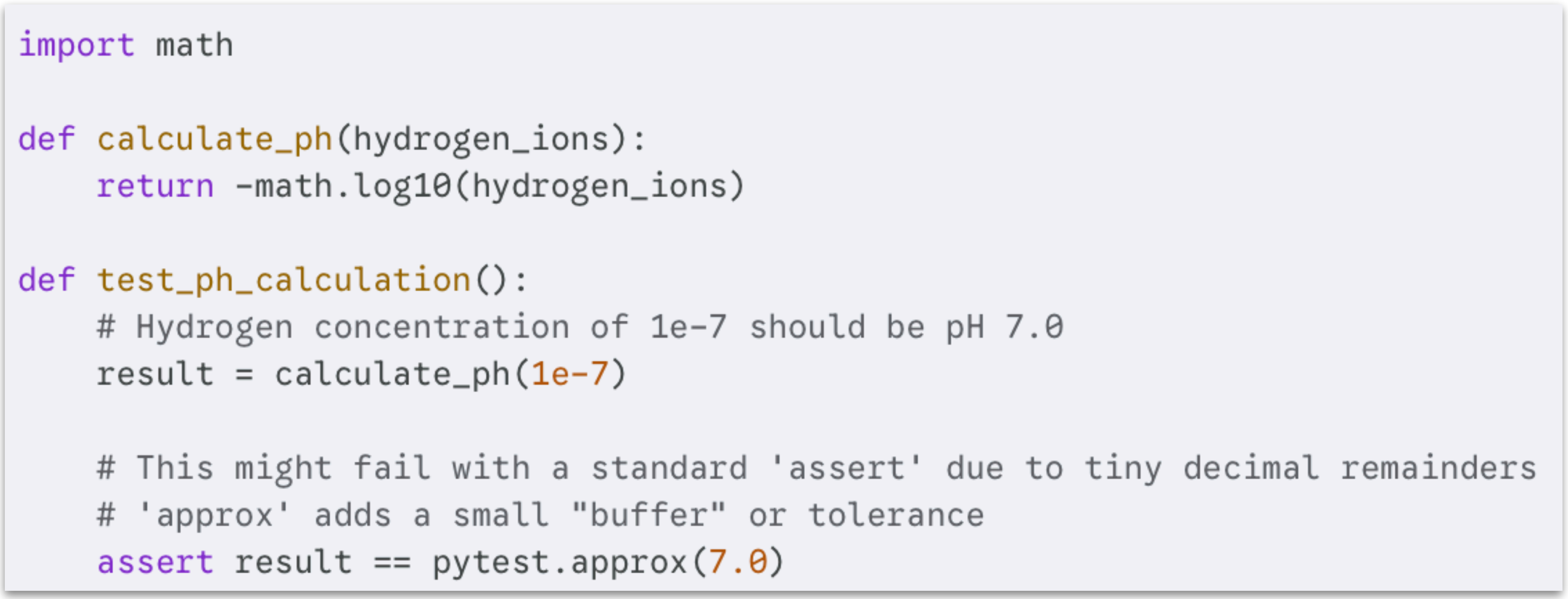
Other pytest features: Fixtures



```python
@pytest.fixture
def desert_station():
    """Provides a station object set to a desert environment."""
    return WeatherStation(name="Death Valley", avg_temp=45)

def test_station_name(desert_station):
    assert desert_station.name == "Death Valley"

def test_station_extreme_heat(desert_station):
    assert desert_station.is_extreme() is True
```

Setup a “clean” instantiation that can be used by all the tests without have to initialize over and over again

Other pytest features: Approximate comparisons



```python
import math

def calculate_ph(hydrogen_ions):
    return -math.log10(hydrogen_ions)

def test_ph_calculation():
    # Hydrogen concentration of 1e-7 should be pH 7.0
    result = calculate_ph(1e-7)

    # This might fail with a standard 'assert' due to tiny decimal remainders
    # 'approx' adds a small "buffer" or tolerance
    assert result == pytest.approx(7.0)
```

Find the bugs

Assume a function called “calculate_fire_danger” follows these following rules:

Inputs: temp (celsius), humidity (0-100 %), wind speed (km/h)

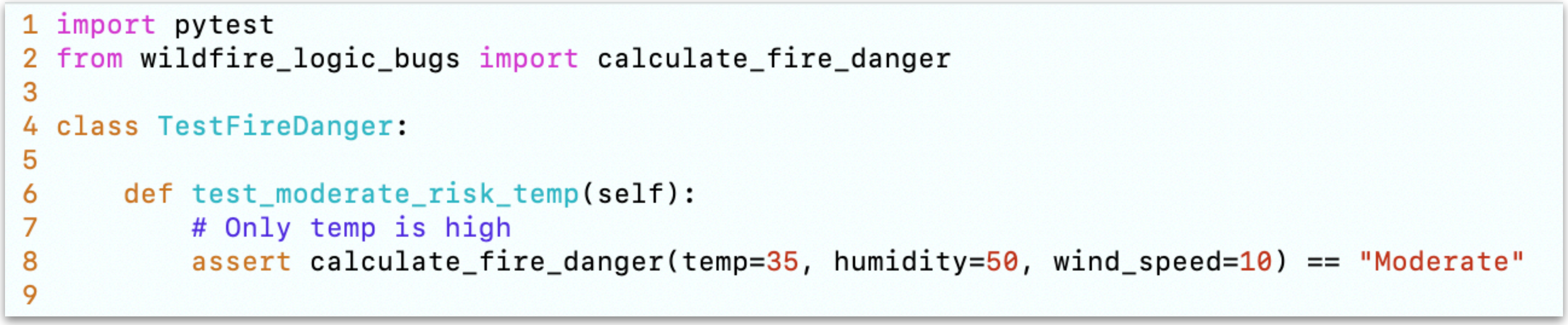
Outputs: “Low”, “Moderate”, and “High”

Logic rules:

- **High -** humidity is < 20% and wind speed is > 30 km/h
- **Moderate -** temp > 30% or humidity < 30 %
- **Low -** Everything else
- **Errors -** Must raise a ValueError if humidity is outside 0-100 or wind/temp are physically impossible

Design some unit tests that are able to ensure this logic is met

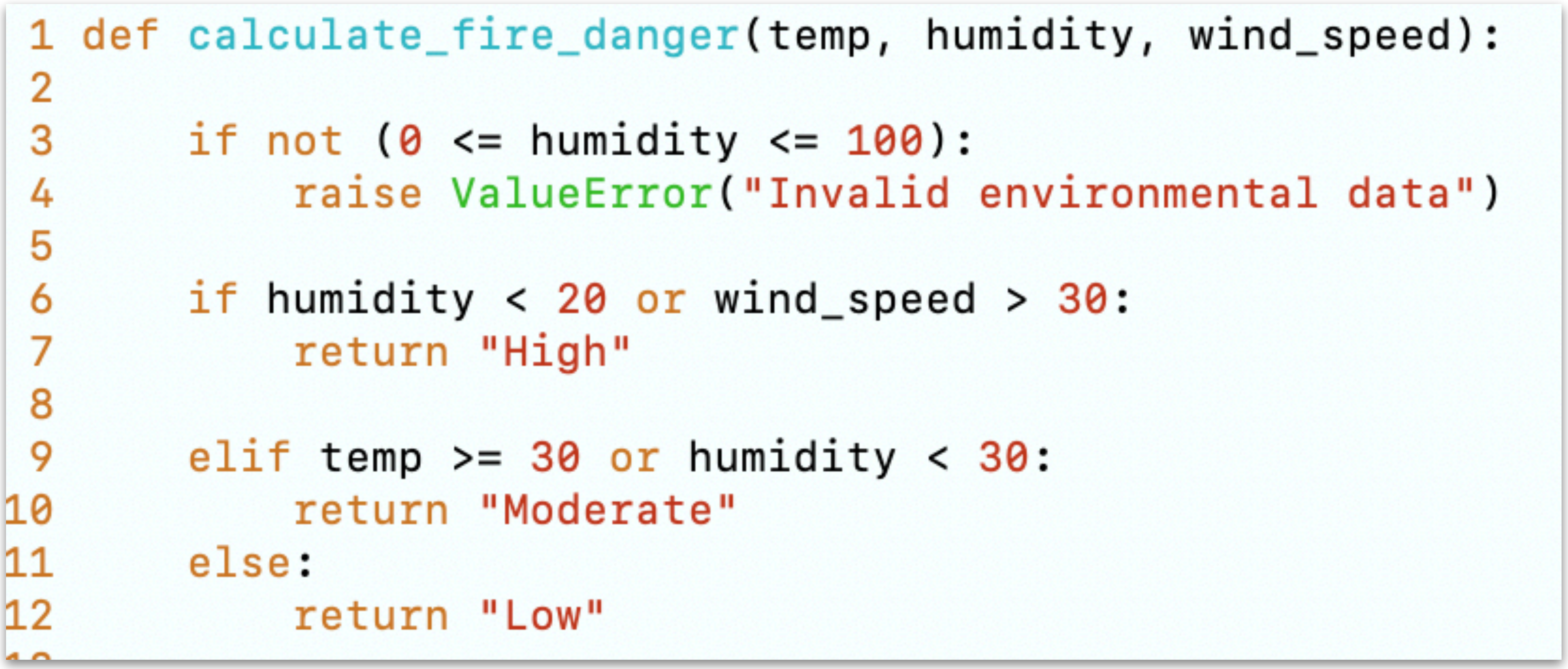
test_wildfires_bugs.py



```python
import pytest
from wildfire_logic_bugs import calculate_fire_danger

class TestFireDanger:

    def test_moderate_risk_temp(self):
        # Only temp is high
        assert calculate_fire_danger(temp=35, humidity=50, wind_speed=10) == "Moderate"

```

Let’s design more unit tests within this class that will find errors in our code

wildfire_logic_bugs.py



```python
def calculate_fire_danger(temp, humidity, wind_speed):

    if not (0 <= humidity <= 100):
        raise ValueError("Invalid environmental data")

    if humidity < 20 or wind_speed > 30:
        return "High"

    elif temp >= 30 or humidity < 30:
        return "Moderate"
    else:
        return "Low"
```