Code Warriors: The Art of Testing

Catch Bugs Early, Write Better Code

Michael Borck

Table of contents

Objectives

- Learn the importance of testing in software development.
- Understand how to write tests using `doctest`.
- Run tests using `doctest`

Why Test?

- Ensure code correctness
- Improve software quality
- Detect bugs early
- Validate functionality
- Facilitate maintenance

Types of Testing

Test Type	Description
Unit	Test an individual isolated component
Integration	Test multiple units work together
End-to-End	Act as user, test entire stack
Acceptance Test	Verify user story works as expected

Understanding Different Types of Errors

Error Type	Description	Example
Syntax Error	Occurs when the code violates the syntax rules of the programming language.	`print("Hello World` (missing closing parenthesis)
Runtime Error	Occurs during the execution of the program, causing it to terminate abruptly.	Division by zero: `1 / 0`
Logical Error	Occurs when the code runs without crashing but produces incorrect results.	Using `=` instead of `==` in a conditional statement

Example Runtime Errors

Exception	Description	Example
Type Error	Occurs when an operation or function is applied to an object of inappropriate type.	Trying to add a string and an integer: `"2" + 2`
Name Error	Occurs when a variable or function name is not found.	Using an undefined variable: `print(x)` (where `x` is not defined)
Index Error	Occurs when trying to access an element outside the bounds of a list.	Accessing a non-existent list index: `my_list[10]` (for a list of length < 11)
Attribute Error	Occurs when an invalid attribute reference or assignment is made.	Accessing a non-existent attribute: `my_obj.non_existent_attribute`
Value Error	Occurs when a function receives an argument of the right type but inappropriate value.	Converting an invalid string to an integer: `int("abc")`

Tools for Testing

- `assert`
- `unittest`
- `pytest`
- `doctest`

Certainly! Here are two slides on using `assert` outside of any testing framework:

What is `assert`? - A built-in statement used to test conditions - Raises an `AssertionError` if the condition is `False` - Quick and easy way to test code - Useful for catching bugs early - Immediate feedback on failed conditions - Enhances code reliability and correctness

Basic Usage

```
def add(a, b):
    return a + b

result = add(2, 3)
assert result == 5, f"Expected 5, got {result}"
```

Example: Using `assert` for Validation

```
def multiply(a, b):
    return a * b

# Test cases
assert multiply(2, 3) == 6, "Test case 1 failed"
assert multiply(-1, 5) == -5, "Test case 2 failed"
assert multiply(0, 10) == 0, "Test case 3 failed"
```

Example: Checking conditions

```
def divide(a, b):
    assert b != 0, "Denominator cannot be zero"
    return a / b

# Test cases
assert divide(10, 2) == 5, "Test case 1 failed"
assert divide(9, 3) == 3, "Test case 2 failed"
```

Example: Using `unittest`

```python
import unittest

def add(a, b):
    return a + b

class TestAdd(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)
        self.assertEqual(add(-1, 1), 0)

if __name__ == '__main__':
    unittest.main()
```

Example: Using `pytest`

```python
def add(a, b):
    return a + b

def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
```

What is `doctest`?

- `doctest` allows writing tests within docstrings.
- Verify code functionality directly in documentation.

Example of `doctest`

```python
def add(a, b):
    """
    Add two numbers and return the result.

    Args:
        a (int or float): The first number.
```

```
        b (int or float): The second number.

    Returns:
        int or float: The sum of `a` and `b`.

    Examples:
    >>> add(2, 3)
    5
    >>> add(1.5, 2.5)
    4.0
    """
    return a + b
```

Running `doctest`

- Run `doctest` using the following command:

```
!python -m doctest -v your_script.py
```

Adding `doctest` to Functions

- Add `doctest` to the `fetch_weather_data` function in `fetch_data.py`.

fetch_data.py with `doctest`

```
def fetch_weather_data(api_key, location):
    """
    Fetch weather data from the OpenWeatherMap API.
    ...

    Examples:
    >>> fetch_weather_data('invalid_api_key', 'London') is None
    True
    >>> isinstance(fetch_weather_data('valid_api_key', 'London'), dict)
    True
    """
```

```
    url = ....
    ...
```

Running `doctests`

- Run `doctests` using the following command:

```
!python -m doctest -v scripts/fetch_data.py
```

Testing Data Processing Functions

- Add `doctest` to data processing functions to ensure correctness.

```
def convert_temp_kelvin_to_celsius(kelvin):
    """
    Convert temperature from Kelvin to Celsius.
    ...

    >>> convert_temp_kelvin_to_celsius(273.15)
    0.0
    >>> convert_temp_kelvin_to_celsius(0)
    -273.15
    >>> convert_temp_kelvin_to_celsius(373.15)
    100.0
    """
    return kelvin - 273.15
```

Running All `doctests`

- Run all `doctests` in the project using:

```
!python -m doctest -v scripts/*.py
```

Integrating `doctest` into Your Workflow

- Ensure all functions have appropriate `doctests`.
- Run `doctests` regularly to verify functionality.

Homework

- Add more `doctest` cases to cover edge cases and different scenarios.
- Explore the official `doctest` documentation for more advanced usage.

Summary

- Learned the importance of testing.
- Wrote and ran tests using `doctest`.
- Tested weather fetching and data processing functions.

Next Sessions

- Focus on debugging
- Add documentation
- Distribution methods