CS4 Python Testing Guidelines

1 Why Test?

Failure is inevitable. We all make mistakes. We need to test so that the code we write works as intended. With solid testing, we can prove that our code works. When our code doesn't work, testing also helps us identify what went wrong, where something went wrong, and how we might fix our mistakes.

2 What is Test-Driven Development?

Writing our tests before we start to code forces us to think about the different and often complex things we need to handle. We should have a clear idea of what can be taken as input and have firm expectations about how different inputs should be handled.

Test-Driven Development is the practice of writing these appropriate test cases before coding. Once we have carefully thought about testing, writing robust code should follow.

3 CS4 Testing Guidelines

3.1 An initial approach to testing in Python

So you want to write some tests? Let's take a look at this example of a **test case**, or an attempt to ask and answer a single question about your code.

Assume we have the function written `fibonacci(n)` which returns the n^{th} **Fibonacci number**, with `fibonacci(1) = 0`, `fibonacci(2) = 1`.

How can we test that our `fibonacci(5)` returns the actual 5^{th} Fibonacci number, 3? After our fibonacci function, we might write:

```
# Testing fibonacci(n) general cases
 assert fibonacci(5) == 3, "Fibonacci general case test 1 failed."
```

Each test is made up of several components. Let's break it down:

- **Comment**: A brief comment describing the test. (In the above example - `Testing fibonacci(n) general cases`)
- **`assert` Statement**: Signals the beginning of a test.
- **Test**: The test statement. This should contain some sort of comparison operator – in most cases ==. (In the above example - `fibonacci(5) == 3`)
- **Error Message**: The message to be displayed if the test evaluates to `false`. (In the above example - `"Fibonacci general case test 1 failed."`)

If a test fails, the code stops on the first assertion that it did not pass, and outputs the corresponding error message. We can then find that piece of code and identify what might have gone wrong. If all assertions pass, nothing will be printed and the rest of our code will continue to run normally.

3.2 Test Coverage

Now that we know how tests work and how to write a test, how can we make sure that the tests we write are good? Here we introduce the concept of **coverage**. At a high level, test coverage is the proportion of requirements of a function/application that our tests address. We'll examine this concept in this section.

How can we make sure that our tests are **good**? We can break down tests into three different groups: **valid input**, **invalid input**, and **edge cases** that we keep in mind in writing a robust collection of tests.

Let's come back to our `fibonacci(n)` function from before.

Valid input tests check that given input that you would expect, our code gives the expected result. These tests check the meat of what our functions aim to do.

These might be some examples of valid input tests:

```
# Testing fibonacci general cases
assert fibonacci(4) == 2, "Fibonacci general case test 1 failed."
assert fibonacci(5) == 3, "Fibonacci general case test 2 failed."
assert fibonacci(10) == 34, "Fibonacci general case test 3 failed."
```

Invalid input tests checks for input that can't be used given the specifications of the function. We make sure that if the input is not valid, we address that properly using exceptions, asserts, or other checks. These tests are likely outside the scope of what we might ask for in implementing a function and writing tests, but these are good to keep in mind!

These might be some examples of invalid inputs that we would need to address.

```
fibonacci(-3.1415)
fibonacci("CS4")
fibonacci(new Date(3,14,2015))
```

Since these examples will throw exceptions, we can not simply use the `assert` statement that we have been using to test valid input. Testing exceptions requires looking into more sophisticated ways of assertions. For more information, you can read about the `unittest` module and `assertRaises`.

Edge case tests are often where the inklings of 2 a.m. lazy and/or exhausted coding are caught. Edge cases technically are a part of valid input tests, but we identify them because our code might deal with these inputs differently than most inputs, or have unique behavior.

These cases test situations when the function may not be acting as it generally does. Some examples include:

- empty or None
- base cases
- initial value cases (as in the fibonacci example)
- negative/zero values
- inequalities (i.e. test 4 when function contains <= 4)

Edge cases for `fibonacci(n)` might look like:

```
# Testing prescribed initial Fibonacci numbers
assert fibonacci(1) == 0, "Failed to return 1st initial Fibonacci value."
assert fibonacci(2) == 1, "Failed to return 2nd initial Fibonacci value."
```

In testing in each of these cases, we aim for complete **code coverage**. This is the idea that our tests execute or *cover* every line of code that we wrote.

3.3 Test Functions

For each non-trivial function that you write, you should write a **test function** containing all of the necessary tests. This test function and its execution should immediately follow the function you are testing. The name of this function must be in the form `my_function_test`. To run your tests, execute the `cs4Tester.py` file, which should be installed as part of your homework stencils; again, the test function's name **must** end in `_test`! A complete example for writing and testing the `fibonacci(n)` function with all valid input cases can be seen below:

```
def fibonacci(n):
    # Calculates the nth fibonacci number.
    ...
    ...
    return fibonacci_number

def fibonacci_test():
    # Testing fibonacci general cases
    assert fibonacci(4) == 2, "Fibonacci general case test 1 failed."
    assert fibonacci(5) == 3, "Fibonacci general case test 2 failed."
    assert fibonacci(10) == 34, "Fibonacci general case test 3 failed."

    # Testing prescribed initial Fibonacci numbers
    assert fibonacci(1) == 0, "Failed to return 1st initial Fibonacci value."
    assert fibonacci(2) == 1, "Failed to return 2nd initial Fibonacci value."
```

Below is another example containing a complete set of valid input tests for the function `sort(nums)`, which sorts a list of integers in ascending order:

```
def sort(nums):
    # Sorts a given list of integers in ascending order.
    ...
    ...
    return sorted_list

def sort_test()
    # Testing general list of integers
    assert sort([2, 5, 4, 1, 3, 7, 6]) == [1, 2, 3, 4, 5, 6, 7], "Sort general
    case failed."

    # Testing already sorted list
    assert sort([1, 2, 3, 4, 5]) == [1, 2, 3, 4, 5], "Failed to sort pre-sorted
    list"

    # Testing list in descending order
    assert sort([5, 4, 3, 2, 1]) == [1, 2, 3, 4, 5], "Failed to sort list in
    descending order"

    # Testing list with negative/zero values
    assert sort([5, 0, -4, 3, -12, -1, 100]) == [-12, -4, -1, 0, 3, 5, 100],
    "Failed to sort list with negative/zero values"

    # Testing list with a single element
    assert sort([4]) == [4], "Failed to sort single element list"

    # Testing empty list
    assert sort([]) == [], "Failed to sort empty list"
```

Ultimately, testing our code and functions well isn't exactly about **how many** tests we write, but how **comprehensive** our test suite is. Having a clear understanding of the specifications of the function and strictly enforcing expectations based on the variety of inputs our function might encounter will allow us to better understand the nature of the function and in turn how to test it. Just like playing the piano or mastering video games, with time and practice we too can "git gud" at writing tests.

4 References

- `http://cs.brown.edu/courses/cs016/static/files/docs/PythonTesting.pdf`
- `https://en.wikipedia.org/wiki/Test-driven_development`
- `http://istqbexamcertification.com/why-is-testing-necessary/`