18.6. Writing Test Cases for Functions

It is a good idea to write one or more test cases for each function that you define.

A function defines an operation that can be performed. If the function takes one or more parameters, it is supposed to work properly on a variety of possible inputs. Each test case will check whether the function works properly on **one set of possible inputs**.

A useful function will do some combination of three things, given its input parameters:

- Return a value. For these, you will write **return value tests**.
- Modify the contents of some mutable object, like a list or dictionary. For these you will write **side effect tests**.
- Print something or write something to a file. Tests of whether a function generates the right printed output are beyond the scope of this testing framework; you won't write these tests.

18.6.1. Return Value Tests

Testing whether a function returns the correct value is the easiest test case to define. You simply check whether the result of invoking the function on a particular input produces the particular output that you expect. If `f` is your function, and you think that it should transform inputs `x` and `y` into output `z`, then you could write a test as `assert f(x, y) == z`. Or, to give a more concrete example, if you have a function `square`, you could have a test case `assert square(3) == 9`. Call this a **return value test**.

Because each test checks whether a function works properly on specific inputs, the test cases will never be complete: in principle, a function might work properly on all the inputs that are tested in the test cases, but still not work properly on some other inputs. That's where the art of defining test cases comes in: you try to find specific inputs that are representative of all the important kinds of inputs that might ever be passed to the function.

The first test case that you define for a function should be an "easy" case, one that is prototypical of the kinds of inputs the function is supposed to handle. Additional test cases should handle "extreme" or unusual inputs, sometimes called **edge cases**. For example, if you are defining the "square" function, the first, easy case, might be an input like 3. Additional extreme or unusual inputs around which you create test cases might be a negative number, 0, and a floating point number.

One way to think about how to generate edge cases is to think in terms of **equivalence classes** of the different kinds of inputs the function might get. For example, the input to the `square` function could be either positive or negative. We then choose an input from each of these classes. **It is important to have at least one test for each equivalence class of inputs.**

Semantic errors are often caused by improperly handling the boundaries between equivalence classes. The boundary for this problem is zero. **It is important to have a test at each boundary.**

Another way to think about edge cases is to imagine things that could go wrong in the implementation. For example, in the square function we might mistakenly use addition instead of multiplication. Thus, we shouldn't rely on a test that uses 2 as input, but we might be fooled into thinking it was working when it produced an output of 4, when it was really doubling rather than squaring.

Try adding one or two more test cases for the square function in the code below, based on the suggestions for edge cases.

Save & Run | Original - 1 of 1 | Show in CodeLens

```
1 def square(x):
2     return x*x
3
4 assert square(3) == 9
5
```

Activity: 1 -- ActiveCode (ac19_2_1)

18.6.2. Side Effect Tests

To test whether a function makes correct changes to a mutable object, you will need more than one line of code. You will first set the mutable object to some value, then run the function, then check whether the object has the expected value. Call this a **side effect test** because you are checking to see whether the function invocation has had the correct side effect on the mutable object.

An example follows, testing the `update_counts` function (which is deliberately implemented incorrectly…). This function takes a string called `letters` and updates the counts in `counts_diction` that are associated with each character in the string. To do a side effect test, we first create a dictionary with initial counts for some letters. Then we invoke the function. Then we test that the dictionary has the correct counts for some letters (those correct counts are computed manually when we write the test. We have to know what the correct answer should be in order to write a test). You can think of it like writing a small exam for your code – we would not give you an exam without knowing the answers ourselves.

Save & Run | Original - 1 of 1 | Show in CodeLens

```
1 def update_counts(letters, counts_d):
2     for c in letters:
3         counts_d[c] = 1
```

```
        if c in counts_d:
            counts_d[c] = counts_d[c] + 1


counts = {'a': 3, 'b': 2}
update_counts("aaab", counts)
# 3 more occurrences of a, so 6 in all
assert counts['a'] == 6
# 1 more occurrence of b, so 3 in all
assert counts['b'] == 3

```

Activity: 2 -- ActiveCode (ac19_2_2)

Error

```
AssertionError:  on line 11
```

Description

An assertion error happens when python encounters an assertion statement. Python evaluates the expression to the right of the word assert; if that expression isTrue everything is fine and the program continues. If the expression is False Python raises an error and stops.

To Fix

Check the expression to the right of assert. The expression is False and you will need to determine why that is. You may want to simply print out the indidividual parts of the expression to understand why it is evaluating to False.

test-2-1: If you write a complete set of tests and a function passes all the tests, you can be sure that it's working correctly.

- A. True
- B. False

Check me | Compare me

✔ The tests should cover as many edge cases as you can think of, but there's always a possibility that the function does badly on some input that you didn't include as a test case.

Activity: 3 -- Multiple Choice (question19_2_1)

test-2-2: For the hangman game, the blanked function takes a word and some letters that have been guessed, and returns a version of the word with _ for all the letters that haven't been guessed. Which of the following is the correct way to write a test to check that 'under' will be blanked as `'u_d__'` when the user has guessed letters d and u so far?

- A. assert blanked('under', 'du', 'u_d__') == True
- B. assert blanked('under', 'u_d__') == 'du'
- C. assert blanked('under', 'du') == 'u_d__'

Check me | Compare me

✔ This checks whether the value returned from the blanked function is 'u_d__'.

Activity: 4 -- Multiple Choice (question19_1_3)

You have attempted 5 of 4 activities on this page

18.5. Testing Loops">
sting Loops
18.7. Testing Optional Parameters">
✔ Completed. Well Done!
18.7. Testing Optional Parameters">Next Section - 18.7. Testing Optional Parameters

| Back to top