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.

**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.

**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).

example: 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\_\_'

answer: C