6.7 Testing Functions III: Testing Mutation

The ability to mutate objects means that we have to be careful when writing functions that accept mutable types as parameters. In general, if a function's documentation does not specify that an object will be mutated, then it **must not** be mutated. How can we test that no mutation occured? And, for functions that intend to mutate an object, how can we test that the correct change occured? In this section, we will extend our study of writing tests to answer both of these questions.

Identifying mutable parameters

Consider the `squares` function we introduced at the beginning of the chapter:

```
def squares(numbers: list[int]) -> list[int]:
    """Return a list of the squares of the given numbers."
    squares_so_far = []

    for number in numbers:
        list.append(squares_so_far, number * number)

    return squares_so_far
```

There are two lists in `squares`: the `numbers` parameter, which is an input to the function; and the `squares_so_far` variable, which is an output of the function. Because `squares_so_far` is created by the function `squares`, it is okay that it is mutated (i.e., the call to `list.append` inside the for loop). However, the `numbers` list is passed as an argument to `squares`. Because the docstring does not indicate that `numbers` will be mutated, it is expected that the `squares` function will not mutate the list object referred to by `numbers`.

We can contrast this with how we would document and implement a similar function that *does* mutate its input:

```
def square_all(nums: list[int]) -> None:
    """Modify nums by squaring each of its elements."""
    for i in range(0, len(nums)):
        nums[i] = nums[i] * nums[i]
```

Testing for no mutation

Let us write a test that ensures the `squares` function does not mutate the list referred to by `numbers`:

```
def test_squares_no_mutation() -> None:
    """Test that squares does not mutate the list it is given.
    """
    lst = [1, 2, 3]
    squares(lst)

    # TODO: complete the test
```

In order to test that a list is not mutated, we first create a list `lst`. Second, we call the `squares` function on `lst`; note that this function call returns a list of squares, but we do not assign the result to a variable because we don't actually care about the returned value for the purpose of this test.[1] We can now add an assertion that ensures `lst` has not been mutated:

[1] This might seem a bit strange, as all of our tests so far have been about checking the return value of the function being tested. In practice, we would have such unit/property-based tests for `squares` as well, we just aren't showing them here.

```
def test_squares_no_mutation() -> None:
    """Test that squares does not mutate the list it is given.
    """
    lst = [1, 2, 3]
    squares(lst)

    assert lst == [1, 2, 3]
```

The variable `lst` originally had value `[1, 2, 3]`. So our assertion checks that *after* the call to `squares`, `lst` still has value `[1, 2, 3]`. Another way to accomplish this, without re-typing the list value, is by creating a copy of `lst` before the call to `squares`. We can do this using the `list.copy` method:

```
def test_squares_no_mutation() -> None:
    """Test that squares does not mutate the list it is given.
    """
    lst = [1, 2, 3]
    lst_copy = list.copy(lst)  # Create a copy of lst (not an alias!)
    squares(lst)

    assert lst == lst_copy
```

Note that the order of statements is very important when testing for mutation. We need to create the list and its copy before the call to `squares`. And we need to test for mutation (i.e., the assertion) after the call to `squares`.

Generalizing this test

You might notice that the above `test_squares_no_mutation` test function doesn't actually use the specific elements of the list `lst`. That is, if we replaced `lst`'s value with another list, the test would behave in the exact same way. That makes this test very suitable to be generalized into a *property-based test*, representing the following property:

> For all lists of integers `lst`, calling `squares(lst)` does not mutate `lst`.

Here is how we could implement such a property-based test using the technique we learned in 4.4 Testing Functions II: `hypothesis`.[2]

[2] We've included the import statements to remind you about the ones from `hypothesis` you need for property-based tests.

```
from hypothesis import given
from hypothesis.strategies import lists, integers


@given(lst=lists(integers()))
def test_squares_no_mutation_general(lst: list[int]) -> None:
    """Test that squares does not mutate the list it is given.
    """
    lst_copy = list.copy(lst)  # Create a copy of lst (not an alias!)
    squares(lst)

    assert lst == lst_copy
```

Testing for mutation

Now let's consider testing the `square_all` function from above. One common error students make when writing tests for mutating functions is to check the return value of the function.

```
def test_square_all() -> None:
    """Test that square_all mutates the list it is given correctly.
    """
    lst = [1, 2, 3]
    result = square_all(lst)

    assert result == [1, 4, 9]
```

This test fails because `square_all` returns `None`, and `None == [1, 4, 9]` is False. Using `result` in our assertion is not useful for testing if `lst` was mutated. Instead, we must test if the value of `lst` has changed:[3]

[3] Like `test_squares_no_mutation`, this test does not store the return value of the function being tested. But the reason is quite different!

```
def test_square_all_mutation() -> None:
    """Test that square_all mutates the list it is given correctly.
    """
    lst = [1, 2, 3]
    square_all(lst)

    assert lst == [1, 4, 9]
```

We can again generalize this test into a property-based test by storing a copy of the original list and verifying the relationship between corresponding elements. We'll leave it as an exercise for you to read through and understand the following property-based test:

```
@given(lst=lists(integers()))
def test_square_all_mutation_general(lst: list[int]) -> None:
    """Test that square_all mutates the list it is given correctly.
    """
    lst_copy = list.copy(lst)
    square_all(lst)

    assert all({lst[i] == lst_copy[i] ** 2 for i in range(0, len(lst))})
```