CSC108: Testing

Daniel Zingaro
University of Toronto
daniel.zingaro@utoronto.ca

November 2014

1 Choosing Good Test Cases

Suppose you wanted to write a function that returns the absolute value of its parameter. Python already has this built-in, of course:

```
>>> abs(-8)
8
>>> abs(0)
0
>>> abs(9)
9
>>> abs(0.5)
0.5
>>> abs(-0.5)
0.5
>>> abs(-1)
1
>>>
```

but let's say you'd like to write your own version of this function.

As shown, the absolute value function returns the number itself if the number is at least 0, or returns the positive version of the number if the number is less than 0.

Now assume you've written a function called `abs1` that is supposed to calculate the absolute value. For now, let's assume the type contract is

`(int) -> int`

so that all we have to consider are integer inputs. How can you be sure that your function is correct?

Correct means that your function does the proper thing for every single allowable value of its parameters. If any valid call of your function produces an incorrect result, your function is not correct. However, the important thing here is that there are an infinite number of ways to call your function. I could call `abs1(3)`, or `abs1(4)`, or `abs1(5)`, or `abs1(6)`, or `abs1(7)`, and so on, forever and ever. When should I stop? If I test all the way

from `abs1(1)` to `abs1(8000)` and then stop, with those 8000 calls working correctly, can I stop there? If I do stop, then what if `abs1(8001)` were the test that finally failed?

Unfortunately, testing with every possible value is impossible. There are an infinite number of integers. Even if the function took a string instead of a number, the problem is no easier: there are an infinite number of strings, and an infinite number of lists, etc. So, given that a correct function requires it to work for every possible call, how can we ever say that our function is correct if it's impossible to test with every possible parameter value?

The way around the problem of infinite testing is to observe that many tests, though they appear to be different, are actually testing your function in the same way. For example, consider calling your `abs1` function with `abs1(4)` and getting a return value of `4`. Since 4 is a positive integer, what we might conclude from this is that your function works correctly for **any** positive integer that is greater than 1. That is, it is highly unlikely that your function would be able to work for 4, but somehow not work for 5 or 50 or 8000. It should also work for 2 or 3 if it works for 4. Why might it not work for 1? Why did I say "positive integer that is greater than 1" earlier in this paragraph?

To see why 1 is "different" from 4 or 50 or 8000, let's take a look at one incorrect but seemingly-reasonable absolute value function.

```
def abs1(val):
  '''(int) -> int

  Return absolute value of val.

  >>> abs1(-4)
  4
  '''

  if val > 1:
    return val
  else:
    return -val
```

This function does work for 2, 3, 4, and higher and higher integers. But it does not work for 1. For an input of `1`, `abs1` returns `-1` which is incorrect.

Can you see the mistake?

The reason that `1` does not work but higher integers do work is that `1` is close to a **boundary value** for the function. The absolute value function does the same thing for all positive integers including 1, but 1 is close to where the function starts doing something else. In particular, 1 is the next-highest integer above 0, and an absolute value function might try to make a distinction between 1 and 0. 0 is also a boundary condition for absolute value, because 0 is close to -1, and the function starts doing something different when we get to negative numbers.

You should use two sources of information to decide on the test cases for a function.

First, use typical values of data types that cause trouble in many functions:

Data Type	**Typical Test Values**
int	-1, 0, 1, positive int, negative int
str	empty string, string of length 1, string of positive length
list	empty list, list with one element, list with multiple elements
dict	empty dict, dict with one entry, dict with multiple entries

Second, combine these "typical values" with the particulars of the function that you are trying to test. Use what you know about writing functions to anticipate likely places where the function could fail.

Let's return briefly to the absolute value function `abs1` above. The type contract indicated `(int) -> int`, meaning that it is required to work only for ints. In particular, this means we shouldn't test with floats, because the type contract does not allow a float parameter. With that in mind, I would test the function with the following set of test cases. I also provide the **category** to which each test case belongs. For example, one of the test cases is the number 2, and that category name is "positive int". I therefore assume that other positive ints will work, since they are in the same category as 2.

Value of `val`	Return value	Purpose of this test case
`0`	`0`	0 boundary
`1`	`1`	1 boundary
`2`	`2`	positive integer
`-1`	`1`	-1 boundary
`-2`	`2`	negative integer

On the other hand, let's say that the type contract for our function were as follows:

`(number) -> number`

and that we wanted to allow both ints and floats. If an int is provided, we want to return an int; if a float is provided, we want to return a float. Now, more test cases are required because we have to check that the function works properly in more situations (the new ones involve floats). My new set of test cases might be as follows:

Value of `val`	Return value	Purpose of this test case
`0`	`0`	0 boundary
`1`	`1`	1 boundary
`2`	`2`	positive integer
`-1`	`1`	-1 boundary
`-2`	`2`	negative integer
`0.5`	`0.5`	between 0 and 1
`4.2`	`4.2`	positive float
`-0.5`	`0.5`	between 0 and -1
`-4.2`	`4.2`	negative float

If an absolute value function works for all of these cases, it is likely (but not certainly) correct. It tests a variety of situations where mistakes like < instead of <=, or `1` instead of `0`, can lead to incorrect behavior.

The assumption of this set of test cases — and indeed all test cases — is that the function will be implemented in accordance with our expectations. If someone really does write an absolute value function where they hard-code only a few values, like:

```
def abs1(val):
  if val == 4.2 or val == -4.2:
    return 4.2
  ...
```

Then it can be made to pass our tests but is still incorrect. All we can do in our tests is test a **reasonable** implementation of the function and hope that our understanding of common errors will catch any errors that do actually exist in the code.

2 Exercise: A String Example

Here is the type contract and description for a function. Do not write the code for this function.

```
def same_strings(str1, str2):
  '''(str, str) -> bool
  Return True iff str1 and str2 are the same ignoring case.
  '''
```

Fill-in the following table. Provide three test cases, each of which is representative of a separate category of test cases. (For example, don't test both `('a', 'b')` and `('c', 'd')`. They are in the same category, and are not testing different aspects of the function.)

Value of `str1` and `str2`	Return value	Purpose of this test case

3 Examples in Docstrings

One of the key steps in our design recipe this semester is the inclusion of examples in our functions' docstrings. Typically, we include one or two typical examples in the docstring. This is not the same as testing as described above: docstrings are intended to serve as documentation for people reading your code or using your functions, not as comprehensive test suites.

That said, it is important that the examples in the docstrings faithfully represent what actually happens when the function is called. If docstring examples say one thing but the function does another, the docstrings will not be useful as documentation.

One way to test the examples in your docstrings is to `import` your module and then type the examples at the shell, manually verifying that the function returns what is indicated by each example. However, there is an automated way by which these examples can be run and checked. Let's reconsider an early example from this semester. Save this code in file `vowels.py`.

```
def num_vowels(s):
  '''(str) -> int
  Return number of vowels in s.

  >>> num_vowels('Europe')
  4
  '''
  counter  = 0
  for char in s:
    if char in 'aeiouAEIOU':
      counter = counter + 1
  return counter
```

There is only one example in this docstring, but the following applies even when there is more than one. To automatically check this example, type the following at the Python shell:

```
>>> import vowels
>>> import doctest
>>> doctest.testmod(vowels)
TestResults(failed=0, attempted=1)
```

doctest is the module that finds the tests in docstrings and executes them. It finds tests by searching for the >>> string in your docstrings. When you call `testmod`, you will be told the number of tests that failed. Here, the single example did not fail, but it is a good exercise for you to change the `num_vowels` function so that it works incorrectly and then run this example again.

A couple of things:

- You can `import vowels` only once.
  If you make changes to `num_vowels` and `import vowels` again without restarting the shell, the `import` will do nothing and you will still be using and testing the old code. Instead, restart the Python shell and then `import vowels` again.
- Be very careful of trailing spaces in your docstring examples. They will cause doctest to report failure even though the output looks the same. For example, change the example in the docstring to:

  ```
  >>> num_vowels('Europe')
  4<space>
  ```

  (where `<space>` is a space character.) Running `testmod` on the function now yields:

  ```
  **********************************************************************
  File "vowels.py", line 5, in vowels.num_vowels
  Failed example:
  num_vowels('Europe')
  Expected:
  ```

```
4
Got:
4
**********************************************************************
1 items had failures:
1 of   1 in vowels.num_vowels
***Test Failed*** 1 failures.
TestResults(failed=1, attempted=1)
```