Testing Automatically Using doctest

As part of the Function Design Recipe, we include one or two example calls on the function in the docstring. For example, consider the function `collect_vowels`:

```
def collect_vowels(s):
    """ (str) -> str

    Return the vowels (a, e, i, o, and u) from s.

    >>> collect_vowels('Happy Anniversary!')
    'aAiea'
    >>> collect_vowels('xyz')
    ''
    """

    vowels = ''
    for char in s:
        if char in 'aeiouAEIOU':
            vowels = vowels + char
    return vowels
```

The function above has two examples calls in its docstring. You can execute those calls one at a time in the Python shell:

```
>>> collect_vowels('Happy Anniversary!')
'aAiea'
>>> collect_vowels('xyz')
''
```

Using `doctest`

Using `doctest`, you can run all of the tests from the docstring at once. After running the module containing `collect_vowels`, in the Python shell, import the `doctest` module, then call `doctest.testmod()`:

```
>>> import doctest
>>> doctest.testmod()
TestResults(failed=0, attempted=2)
```

The number of tests that failed and the number of tests that were attempted are reported. In this case, 2 tests were attemped, and 0 failed. `doctest.testmod()` automatically compares the actual value returned by the function call with the value we expect to be returned.

More Examples

Consider this code:

```
def get_divisors(num, possible_divisors):
    """ (int, list of int) -> list of int

    Return a list of the values from possible_divisors
    that are divisors of num.

    >>> get_divisors(8, [1, 2, 3])
```

```
    [1, 2]
    >>> get_divisors(4, [-2, 0, 2])
    [2]
    """

    divisors = []

    for item in possible_divisors:
        if item != 0 and num % item == 0:
            divisors.append(item)

    return divisors
```

The above function, `get_divisors` has a `docstring` with two examples calls. After loading the module that contains the above function, we will test it by executing the following:

```
>>> import doctest
>>> doctest.testmod()
**********************************************************************
File "__main__", line 7, in __main__.get_divisors
Failed example:
    get_divisors(4, [-2, 0, 2])

Exception raised:
    Traceback (most recent call last):
      File "/local/packages/python-2.7/lib/python2.7/doctest.py", line 1254, in __run
        compileflags, 1) in test.globs
      File "", line 1, in
        get_divisors(4, [-2, 0, 2])
      File "", line 12, in get_divisors
    ZeroDivisionError: integer division or modulo by zero
**********************************************************************
1 items had failures:
   1 of   2 in __main__.get_divisors
***Test Failed*** 1 failures.
TestResults(failed=1, attempted=2)
```

From the report above, we see that 2 tests were attempted, but one failed. The failure was due to a `ZeroDivisionError`. Looking at the error report further, we find out that the error happened at line 15 (`if num % item == 0:`).

For one of the function calls, when line 15 is executed, `item` refers to `0`. To prevent this division by zero, we add another condition to the `if` statement: `if item != 0 and num % item == 0:`

Now, the error is avoided. This is true because of *lazy evaluation* (if the first operand in an `and` expression is `False`, the `and` expression evaluates to `False`, and the second operand is not evaluated). Now, if we run `doctest.testmod()` again, we get the following report:

```
>>> doctest.testmod()
**********************************************************************
File "__main__", line 7, in __main__.get_divisors
Failed example:
    get_divisors(4, [-2, 0, 2])
Expected:
    [2]
Got:
    [-2, 2]
**********************************************************************
1 items had failures:
   1 of   2 in __main__.get_divisors
***Test Failed*** 1 failures.
TestResults(failed=1, attempted=2)
```

An error still occurs, but it is different from the previous one. For the function call `get_divisors(4, [-2,0,2])` we expect `[2]` to be returned, but `[-2,2]` is actually returned.

In this case, there is no problem with our function definition, but rather, with our test case. `-2` is in fact a divisor of `8`, and should have been included in our returned value. So, if we change the `docstring` to include `-2`, all tests pass.

Automatically run `doctest`

It is also possible to automate the running of doctest when loading a module into Python, by including the following code at the end of the module:

```
import doctest
doctest.testmod()
```

In this case, every time you run the module, the doctest is also executed, and if there are no errors, then nothing will be reported to the console output.

Jennifer Campbell • Paul Gries
University of Toronto
