1 Motivation

Ideal scenario for writing a function:

- design the header and write the docstring
- choose the test cases
- write the code
- implement and run the test cases

It takes a huge amount of discipline to choose test cases before writing the code, but it pays off.

- forces you to think through situations you might not have considered
- increases your chance of writing correct code in the first place

Testing skills that we'll focus on improving:

- ability to choose test cases that will find errors efficiently
- ability to implement those test cases efficiently

2 Choosing test cases

Let's choose some test cases for this function:

```
def insert_after (L, n1, n2):
      '''(list of ints, int, int)
      After each occurrence of n1 in L, insert the n2.'''
```

Q. How many possible ways can we call this function?

A. too many

Q. How many tests cases would it take to convince you that the function works?

A. As many as possible.

The approach:

1. Divide all possible inputs into meaningful categories (based on features of the input values)
2. Choose a representative test case from each

2.1 Properties of the inputs

For **`insert_after`**, what are the properties of the inputs:

- length of **`L`**
- value of **`n1`**
- value of n2
- number of occurrences of n1 in L
- the position of n1 in L
- the contents of L

Q. Which of the above properties are relevant?

A. length of L, num occurrences of n1, position of n1 in L

Inputs	Property of the inputs	Values to try
L	length	0, 1, longer
L, n1	num occurrences of n1 in L	0, 1, longer
L, n1	position of n1 in L	front , back, somewhere else

2.2 Test cases

`L`	**`n1`**	**`n2`**	Purpose
[]	1	2	length 0, no n1
[0]	0	3	length of 1, n1 occurs
[0]	2	3	length of 1, no n1 occurs
[0, 1, 2, 3, 4, 5]	6	7	longer, no n1
[0, 1, 2, 3, 4, 5]	0	7	longer, n1 at front
[0, 1, 2, 3, 4, 5]	2	7	longer, n1 in "middle"
[0, 1, 2, 3, 4, 5]	5	7	longer, n1 at back
[0, 2, 2, 3, 4, 5, 2]	2	7	longer, several n1s

3 Implementing test cases

Testing (also called verification) involves:

- choosing test cases, as we just did
- implementing and executing them to see if results match expectations

We must re-verify code whenever it changes. Testing done in the shell cannot be reused.

Better: make a main that has testing code. This can be reused. (e.g., a2_type_checks.py)

The **nose**s module is even better.

4 Testing with `nose`

Suppose you are testing module called **mod**. Then create a new module called **test_mod** and in it:

- Import **nose** and **mod**.
- Write a function for each test case. In each function:
  - Set up variables, if necessary.
  - Call the function being tested.
  - Make one assertion: **assert value1 == value2**, description where
    - **value1** tries a test case on the demo code,
    - **value2** is the expected result, and
    - **description** is a string that will be printed if the assertion fails (make it useful!).
  - Often, everything can be done in one line.
- Name each function **test_condition**, where condition describes the test case. Eg, **test_empty_dict**
- In the main, have the single line:**nose.runmodule()** It will cause each test function to be called.

5 A `nose` example

[**functions.py**]

```
def our_max(num1, num2):
    '''(number, number) -> number
    Return the larger of the numbers num1 and num2.'''

    max = num1
    if num2 > num1:
        max = num2
    return max
```

```
def same_string(str1, str2):
    '''(str, str) -> bool
    Return True if strings str1 and str2 have the same contents ignoring case,
    and return False otherwise.'''

    return str1.lower() == str2.lower()
```

Write the test module. [**test_functions.py**]

```
import nose # testing module
import functions # module to be tested

def test_our_max_first_bigger():

    assert function.our_max(8, 4) == 8, \
           'First number is larger.'

def test_our_max_second_bigger():

    assert function.our_max(4, 7) == 7, \
           'Second number is larger.'

def test_our_max_same():

   assert functions.our_max(5, 5) == 5, \
       'The numbers are the same.'

def test_same_string_empty():

   assert functions.same_string('', ''), \
       'Empty strings.'

def test_same_string_diff_contents():

   assert not functions.same_string('abc', 'efg'), \
       'Different string contents.'

def test_same_string_diff_case():

   assert functions.same_string('abc', 'ABC'), \
       'Same thing, different case.'
```

```
x = 8
assert x == 8
assert x == 9
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
AssertionError:
# if the stuffs after assert is true, nothing happens, if false, there will be an error.
assert x == 7, 'x is not 7'
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
AssertionError: x is not 7
```

```
if __name__ == '__main__':
    nose.runmodule()
```

6 Testing with dictionaries

The module to test is [**dict_functions.py**].

```
def increment_count(d, k):
    '''(dict, number) -> number
    Increment the value associated with key k in d.
    If k is not a key in d, add it with value 1.'''

    if k in d:
        d[k] += 1
    else:
        d[k] = 1

def invert(table):
    '''(dict) -> dict
    Return a new dict that is dict table inverted.'''

    index = {}
    for key in table.keys():
        value = table[key]
        if not index.has_key(value):
            index[value] = []
        index[value].append(key)
    return index
```

Q. How would you test **increment_count**? Do you see a problem?

A.

Comparing dictionaries using the equality operator (**==**):

[**test_dict_functions.py**]

```
import nose
import dict_functions

def test_increment_count_empty_dict():

    d = {}
    dict_functions.increment_count(d, 'a')
    assert d == {'a' : 1}
```

```
    def test_increment_count_key_present():

        d = {1 : 3, 2 : 1, 7 : 4}
        dict_functions.increment_count(d, 2)
        assert d == {1 : 3, 2 : 2, 7 : 4}

    def test_increment_count_key_absent():

        d = {1 : 3, 2 : 1, 7 : 4}
        dict_functions.increment_count(d, 8)
        assert d == {1 : 3, 2 : 1, 7 : 4, 8 : 1}
```

```
if __name__ == '__main__':
    nose.runmodule()
```

7 Testing with pictures

Testing with pictures is a special case, because we can't just write a simple assert that will capture correctness.
A few options:

- eyeball the output
  - sometimes looking visually good is good enough
  - but if you have many test cases, this will take a lot of time
- test specific pixels
  - create pictures for input that are as small and simple as possible
  - extract individual pixels and assert what they RGB values should be
  - may be okay to verify only some pixels
  - makes it easy to verify many test cases
- test every pixel
  - extremely tedious, so only would do this if absolutely necessary