Testing Exercises

This handout will give you experience with writing tests for functions. The ability to write tests will not only help you make sure that functions are correct – it will also help you predict the sorts of errors that you might make yourself when writing functions. In general, tests should be written before writing code. This practice allows a programmer to use the tests to guide how a function is written. To simulate this, you will write tests for functions without worrying about the code.

Each of the problems ramp up the amount of work that we leave up to you. You will start by writing tests for a function following a rigorous list of tests that we ask for. You will then both design and write the tests for a few example functions. After this, you will design and write both a function and its tests.

As we have not taught more formal tools for writing tests, use **assert** to test these functions. In general, your tests would exist in a separate file, which would import your the functions from your primary program before testing them.

An important note is that tests cannot be truly exhaustive, as (for most functions) it is not possible to write a test for every possible input. In that case, it is only possible to be very exhaustive by predicting what sort of errors a programmer might make.

Problems

1. Consider the following function:

```
def max_even(lst):
    """
    Returns the maximum even valued integer in lst.

    Keyword arguments: lst -- a list of integers
    If lst has no maximum even value, returns None
    """
    # Implementation not shown
```

Write tests to test the following cases for possiblities of the argument's, lst's, value. Each test should be about one line long. (Note that there are additional cases on the following page.)

- Empty list
- List of one element with even value
- List of one element with odd value
- List of two elements (first odd, then even)
- List of two elements (first even, then odd)
- List of elements where the max value is negative
- Larger list where first even encountered is max
- Larger list where last even encountered is max
- Larger list where some even that is not the first or the last is the max

2. Write tests for the following function:

```
def multiply(a, b):
    """
    Computes the product of the numbers a and b
    """
```

3. Write tests for the following function:

```
def mode(lst):
    """
    Returns the mode, defined as the most common value, in lst

    Keyword arguments:
    lst -- a list of integers

    If len(lst) == 0, returns None
    If multiple modes are possible, returns one of the potential values.
        Which value returned is not explicitly defined.
    """
```

4. Consider the following CSV file, *2012-electoral-college.csv*

```
State,Name,Electors,Population
AK,Alaska,3,710000
AL,Alabama,9,4780000
AR,Arkansas,6,2916000
AZ,Arizona,11,6392000
CA,California,55,37254000
CO,Colorado,9,5029000
CT,Connecticut,7,3574000
```

You have been given the following function, read_csv, which returns a list of dictionaries.

```
def read_csv(path):
    """
    Reads the CSV file at path, and returns a list of rows. Each row is a
    dictionary that maps a column name to a value in that column, as a string.
    """
    output = []
    for row in csv.DictReader(open(path)):
        output.append(row)
    return output
```

We would like to know for each quantity of electors, the average population. For instance, both Alabama and Colorado have 9 electors. If they were the only states that had 9 electors, then the average population for a state of 9 electors would be:

```
(4780000 + 5029000) / 2 = 4904500.0
```

Because you are designing the solution, the format of the output is up to you. Here is a general guideline for approaching a problem and designing a function.

- Specify the function
  - Generally describe its purpose
  - Decide on input/output parameters, and/or side effects. What information does the function require to produce its result?
  - Identify assumptions that you make (e.g. a sorted input parameter).
  - Use this information to write a function definition and docstring.
- Test the function (sometimes optional, but always a good idea)
  - Write some tests solely based on the docstring specification.
- Implement the function (sometimes optional, if stubbing a method to implement later)
  - Decide on an implementation, and write it
  - Be aware of your specification when implementing - the spec may change as you learn more about the problem.
  - Re-evaluate your tests. Do you require any additional tests to address implementation-specific edge conditions?

Answers

In general, there should exist tests to handle

- Small cases
- Normal cases
- Variations on normal cases
- Edge cases

1.

```
assert max_even([])  == None
assert max_even([2]) == 2
assert max_even([1]) == None
assert max_even([1, 2]) == 2
assert max_even([2, 1]) == 2
assert max_even([-2, 1, -4]) == -2 assert max_even([6, 4, 2]) == 6
assert max_even([2, 6, 4]) == 6
assert max_even([4, 2, 6]) == 6
```

2.

```
assert multiply(0, 0)   == 0
assert multiply(1, 0)   == 0
assert multiply(0, 1)   == 0
assert multiply(1, 1)   == 1
assert multiply(-1, 1)  == -1
assert multiply(1, -1)  == -1
assert multiply(-1, -1) == 1
assert multiply(3, 4)   == 12
assert multiply(-3, 4)  == -12
assert multiply(3, -4)  == -12
assert multiply(3, 4)   == 12
```

3.

```
assert mode([])  == None
assert mode([1]) == 1
assert mode([1, 1, 2]) == 1
assert mode([1, 2, 1]) == 1
assert mode([2, 1, 1]) == 1
assert(mode([1, 1, 2, 2]) == 1 or mode([1, 1, 2, 2]) == 2)
assert mode([1, 2, 3, 4, 1, 2, 3, 3]) == 3
```

4.

```
Overall purpose of our function:
To iterate through a given list of dictionaries and calculate the average population
for each states with the same number of electors

Inputs/outputs:
The function needs to work with some data to do the necessary calculations.
For this example, we would want to take a list of dictionaries as an input. Specifically,
dictionaries with the following keys: State,Name,Electors,Population

Assume:
We are given a dictionary with the following keys: State, Name, Electors, Population

Function name/docstring:
name: calculate_avg_population_per_numb_electors(data)
docstring: Given a list of election data, returns a dictionary mapping the
number of electors to the average population for a state with that many electors.

Tests based on function:
(Test cases will vary from person to person, a few examples are shown below)

assert calculate_avg_population_per_numb_electors([]) == {}

test_case = [{"State": "WA", "Name":"Washington", "Electors": 9, "Population": 6000000}]
second_test = [{"State": "WA", "Name":"Washington", "Electors": 9, "Population": 6000000},
               {"State": "CA", "Name":"California", "Electors": 9, "Population": 8000000}]
assert calculate_avg_population_per_numb_electors(test_case) == {9:6000000}
assert calculate_avg_population_per_numb_electors(second_test) == {9:7000000}

Decide on an implementation:
One implementation is shown below:

def calculate_avg_population_per_numb_electors(data):
    internal_dict = {}
    for entries in data:
        electors = entries['Electors']
        if electors in internal_dict.keys():
            internal_dict[electors].append(entries['Population'])
        else:
            internal_dict[electors] = [entries['Population']]
    output_dict = {}
    for values in internal_dict:
        output_dict[values] = sum(internal_dict[values])/len(internal_dict[values])
    return output_dict


print calculate_avg_population_per_numb_electors(read_csv("2012-electoral-college.csv"))
```