CSP1150/CSP5110: Programming Principles
Workshop 10: Testing and Quality Assurance

Task 1 – Testing Boundaries and Edge Cases

Here is the code to define a small function named "`countNum()`":

```python
def countNum(start, end, num):
    count = 0

    for i in range(start, end):
        if str(num) in str(i):
            count = count + 1

    return count
```

The "`countNum()`" function tries to implement the following behaviour:

- It requires 3 parameters, all of which should be integers: `start`, `end`, `num`.
- The function aims to count (and return) how many times `num` appears in the numbers between (and including) `start` and `end`.
    - e.g. With a `start` of `1`, an `end` of `25` and a `num` of `4`, the function should return 3, since the number 4 appears in 4, 14 and 24.
    - A simple test confirms that "`print(countNum(1, 25, 4))`" results in 3.

Come up with some inputs that you feel will test how well the function works and try to identify any bugs it may contain. For now, only test *boundaries and edge cases*, not invalid or unexpected input.

Be sure to include test cases that check the following:

- Inputs *resulting in 0*, e.g. start = 1, end = 5, num = 8 should return 0
- Inputs *resulting in a large result*, e.g. start = 1, end = 500, num = 2 should return 176
- Inputs *involving a negative start or end*, e.g. start = -50, end = -5, num = 6 should return 5
- Inputs *where num is larger than 9*, e.g. start = 100, end = 1000, num = 42 should return 19
- Inputs *where num is in start or end*, e.g. start = 1, end = 12 , num = 1 should return 4
- Inputs *where all parameters are the same*, e.g. start = 5, end = 5 , num = 5 should return 1

Test a few of your test cases manually, then write some basic code to automate it – see slides 17 and 18 of Lecture 10 for an idea of how to do this.

Remember to determine the expected output of your tests yourself – you can't rely upon what the function returns at this stage since it is untested and would defeat the purpose of the testing!

Task 2 – Debugging

Hopefully your testing revealed a bug in the "`countNum()`" function. It relates to how the "`range()`" function works, and it can be fixed by adding just two characters to the code. Fix the bug, and perform the tests again to confirm that the function is now working correctly.

Task 3 –Testing Invalid and Unexpected Input

Now let's decide how the function should behave when given invalid or unexpected input and test for that. I have provided a list of test cases and expected outputs for invalid or unexpected input:

Test Case	Inputs	Expected Outputs
Non-Integer Numeric Inputs	`start = 1.5, end = 9, num = 5`	**`TypeError`**
	`start = 1, end = 9.5, num = 5`	**`TypeError`**
	`start = 1, end = 9, num = 5.5`	**`TypeError`**
Non-Numeric Inputs	`start = 'A', end = 9, num = 5`	**`TypeError`**
	`start = 1, end = 'A', num = 5`	**`TypeError`**
	`start = 1, end = 9, num = 'A'`	**`TypeError`**
Start > End	`start = 9, end = 1, num = 5`	**`ValueError`**
Negative "num" Parameter	`start = 1, end = 9, num = -5`	**`ValueError`**

I've specified that if input of an inappropriate type (whether it be float, string, or any other type) is provided for any of the parameters, a `TypeError` exception should be raised. If the parameters are all integers but the `start` parameter is greater than the `end` parameter or the `num` is negative, a `ValueError` should be raised – since the data types are correct, but the *values* are not.

Your task is to **enhance your basic automated testing code** from Task 1 so that it performs these seven invalid and unexpected input tests as well as the previous tests for boundaries and edges. See slide 22 of Lecture 10 for an example of this.

Run the tests and try to determine which ones pass and which ones fail.

Task 4 – More Debugging

Hopefully your testing revealed the "`countNum()`" function behaves as desired for some of the tests, but resulted in 0 instead of raising an exception for some of the other tests.

Examine the test results and the function code until you are able to determine *why* the test results are as they are, and then try to **modify the function so that it behaves as desired for all test cases**.

One way to implement this is to add a series of "`if`" statements to the start of the function that test a condition and raise an exception if needed, e.g:

```
    if type(start) != int:                                    Python
        raise TypeError('start parameter must be an integer')
```

Similar code can be used to check the type of the other parameters, and to check whether `start` is greater than `end` and whether `num` is less than 0, and raise the appropriate exception (with an appropriate error message) for each case.

Task 5 – Testing with the "unittest" Module

Similar to how the example in Lecture 10 progressed, try to now **use the "unittest" module to test the function**. Use the code from slide 26 of Lecture 10 as an example.

Follow these steps to do this:

1. Copy the code of the "`countNum()`" function into a new file and save it as "countnum.py".
2. Create a new Python program and use the code from slide 26 of Lecture 10 as an example:
    - Import "unittest" and your "countnum" file.
    - Create a class that expands the "unittest.TestCase" class.
    - Create methods inside it with names starting in "test", that use assertions to test the test cases. You should only need "`assertEqual()` and "`assertRaises()`".

Note that "`assertRaises()`" requires the inputs to be specified as separate parameters, e.g. "`self.assertRaises(TypeError, countnum.countNum, 1.5, 9, 5)`".

Test your code and make sure that it is working.

That's all for this workshop. If you haven't completed the workshop or readings, find time to do so before next week's class.