CSC1015F Assignment 5: Testing, Functions, Control (if, for, while)

Assignment Instructions

This assignment involves constructing doctest test scripts, and developing and reasoning about Python programs that use input and output statements, '`if`' and '`if-else`' control flow statements, '`while`' statements, '`for`' statements, and statements that perform numerical manipulation.

Question 1 [40 marks]

This question concerns the construction of a doctest test script for a Python function by using execution path analysis. Read through the question and then consult the appendix on page 3 for a detailed explanation of what is required.

The Vula page for this assignment provides a Python module called '`timeutil.py`' as an attachment. The module contains a function called '`validate`'. The purpose of this function is to accept a string value as a parameter and determine whether it is a valid representation of a 12 hour clock reading.

The string is a valid representation if the following applies:

- It comprises 1 or 2 leading digits followed by a colon followed by 2 digits followed by a space followed by a two letter suffix.
- The leading digit(s) form an integer value in the range 1 to 12.
- The 2 digits after the colon form an integer value in the range 0 to 59.
- The suffix is 'am' or 'pm'.

Leading and trailing whitespace is ignored.

Examples of valid strings:

```
01:10 am
1:15 pm
12:59 am
11:01 pm
```

Your task:

1. Develop a set of test cases for testing every possible execution path.
2. Code your tests as a doctest script suitable for execution within Wing IDE (like the `testchecker.py` module described in the appendix).
3. Save your doctest script as '`testtimeutil.py`'.

NOTE: make sure the docstring in your script contains a blank line before the closing """. (The automarker requires it.)

NOTE: the validate function is believed to be error free.

CONTINUED

Question 2 [60 marks]

Write a program to draw a text-based graph of a mathematical function f(x). Use axis limits of -10 to 10 and only plot discrete points i.e. points with integer value ordinates.

Sample I/O:

```
Enter a function f(x):
x+2
           |        o
           |       o
           |      o
           |     o
           |    o
           |   o
           |  o
           | o
           |o
          o
         o|
--------o-+----------
        o  |
       o   |
      o    |
     o     |
    o      |
   o       |
  o        |
 o         |
o          |
           |
           |
           |
```

Use nested loops to print the entire area of the graph (i.e. an outer loop for rows and an inner one for columns), keeping track of the current x and y values. Whenever the (rounded) value of the function f(x), entered by the user, is equal to the current y value, output "o" (small letter Oh), otherwise, output either the appropriate axis character or a space.

NOTE: Remember to import math to enable the use of some mathematical functions.

How should the program support the entering of arbitrary functions?

- Obtain user input in the form of a string, then within the inner loop,
- whenever f(x) is to be calculated, use the Python 'eval' function on that string.

Save your program as plot.py.

Submission

Create and submit a Zip file called '`ABCXYZ123.zip`' (where ABCXYZ123 is YOUR student number) containing `testtimeutil.py` and `plot.py`.

Appendix: Path testing using the doctest module

Question one concerns path testing. Your task is to create a 'doctest' test script that may be used to evaluate every execution path within a given Python function. In this section we explain what is required.

1. Example function

We'll use an example. Say we have a function called '`check`' that evaluates two parameters '*a*' and '*b*', and returns a value indicating whether one, the other, or both are invalid (possessing an incorrect value). If *a* is invalid but not *b* then the result is 1. If *b* is invalid but not *a* then the result is 2. If both are invalid then the result is 3. If neither is invalid then the result is 0.

Here is an implementation of the function in Python (available on the Vula assignment page) :

```
1 # checker.py
2
3 def check(a, b):
4     errors=0
5     if a<25:
6         errors=errors+3
7     if b<25:
8         errors=errors+2
9     return errors
```

We've numbered the lines for convenience. (You can probably also see, for the sake of realism, that there's an error in the code. Line 6 should be '`errors=errors+1`'.)

2. Devising path tests

Now let's say that we wish to test the function and we're going to do so by evaluating all possible execution paths. Our first step is to identify the paths. Here they are:

1. <3, 4, 5, 6, 7, 8, 9>

2. <3, 4, 5, 6, 7, 9>

3. <3, 4, 5, 7, 8, 9>

4. <3, 4, 5, 7, 8, 9>

(We've chosen to describe each path in terms of the line numbers of the statements executed.)

We obtain the paths by looking at all combinations of '`if`' statement outcomes. For example, for path 1 both '`if`' statement conditionals must evaluate to `True`, while for path 2 the second conditional evaluates to `False` (and hence line 8 does not appear).

The next step is to devise a test for each path:

path #	*Input(a,b)*	*Expected Output*
1	(20, 20)	3
2	(20, 30)	1
3	(30, 20)	2
4	(30, 30)	0

CONTINUED

How are these tests devised?

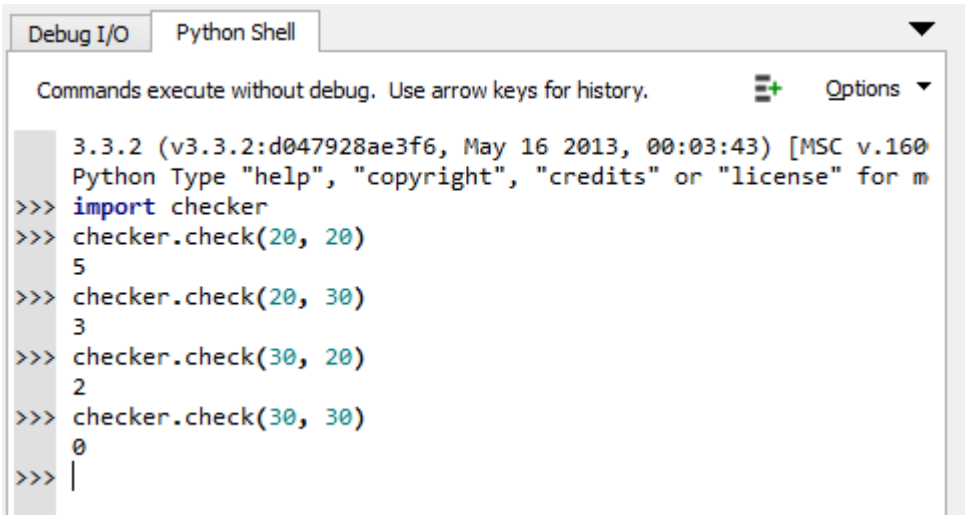
1. Having identified a path, we must figure out from the code what inputs to use to cause execution to go that way. (Not always easy.)
2. Having identified inputs, we must then examine the specification (the description of what the function is supposed to do) to determine what the expected output should be.

3. Running tests

The Python shell in the Wing IDE provides a quick and easy way of running tests.

- We import the module containing the function to be tested and then we issue a call for each set of test inputs.
- If the result of each execution is correct then our tests have passed. If not, then we must debug the code.

Let's say that the '`check`' function is in a module called '`checker.py`' and that we've opened it in Wing:



Our function contains a bug, so tests 1 and 2 have failed. Time to debug....

While this approach to running tests is simple, what if we want to run our tests again? (We certainly should, if we've found and fixed bugs.) We'd have to re-enter our commands (tedious).

We could alternatively write a Python program that contained the necessary function calls and that checked results; however, this may be more effort than it is worth. Another possibility is to use the Python `doctest` module.

CONTINUED

4. The Python doctest module

The doctest module utilises the convenience of the Python interpreter shell to define, run, and check tests in a repeatable manner.

The module does a lot of clever things. For our purposes, we just want to be able to reuse our series of '`checker.check`' tests.

We can do this by writing a doctest script such as follows:

```
>>> import checker
>>> checker.check(20, 20)
3
>>> checker.check(20, 30)
1
>>> checker.check(30, 20)
2
>>> checker.check(30, 30)
0
```

A line beginning with '>>>' is a statement that doctest must execute. If the statement is supposed to produce a result, then the expected value is given on the following line e.g. '`checker.check(20, 20)`' is expected to produce the value `3`.

NOTE: there must be a space between a '>>>' and the following statement.

The script looks much like the transcript of our interactive session in the Python shell (in the screenshot above). The difference is that, if we save this script, we can use it again and again.

It is possible to save the script just as a text file. However, because we're using Wing IDE, it's more convenient to package it up in a Python module (available on the Vula assignment page):

```
# testchecker.py
"""
>>> import checker
>>> checker.check(20, 20)
3
>>> checker.check(20, 30)
1
>>> checker.check(30, 20)
2
>>> checker.check(30, 30)
0

"""
import doctest
doctest.testmod(verbose=True)
```

The script is enclosed within a Python docstring. The docstring begins with three double quotation marks and ends with three double quotation marks.

NOTE: the blank line before the closing quotation marks is essential .

CONTINUED

Following the docstring is an instruction to import the doctest module, followed by an instruction to run the '`testmod()`' function. (The parameter '`verbose=True`' ensures that the function prints what it's doing.)

If we save this as say, '`testchecker.py`', and run it, here's the result:

```
 Trying:
     import checker
 Expecting nothing
 ok
 Trying:
     checker.check(20, 20)
 Expecting:
     3

**********************************************************************
**
 File "testchecker.py", line 3, in __main__
 Failed example:
     checker.check(20, 20)
 Expected:
     3
 Got:
     5
 Trying:
     checker.check(20, 30)
 Expecting:
     1

**********************************************************************
**
 File "testchecker.py", line 5, in __main__
 Failed example:
     checker.check(20, 30)
 Expected:
     1
 Got:
     3
 Trying:
     checker.check(30, 20)
 Expecting:
     2
 ok
 Trying:
     checker.check(30, 30)
 Expecting:
     0
 ok
```

CONTINUED

```
**********************************************************************
**
 1 items had failures:
    2 of   5 in __main__
 5 tests in 1 items.
 3 passed and 2 failed.
 ***Test Failed*** 2 failures.
```

As might be expected, we have two failures because of the bug at line 6.

What happens is that `doctest.testmod()` locates the docstring, and looks for lines within it that begin with '>>>'. Each that it finds, it executes. At each step it states what it is executing and what it expects the outcome to be. If all is well, ok, otherwise it reports on the failure.

The last section contains a summary of events. (Note that doctest thinks of the entire script as a test, whereas from our perspective it consists of four.)

If we correct the bug at line 6 in the check function and run the test script again, we get the following:

```
 Trying:
     import checker
 Expecting nothing
 ok
 Trying:
     checker.check(20, 20)
 Expecting:
     3
 ok
 Trying:
     checker.check(20, 30)
 Expecting:
     1
 ok
 Trying:
     checker.check(30, 20)
 Expecting:
     2
 ok
 Trying:
     checker.check(30, 30)
 Expecting:
     0
 ok
 1 items passed all tests:
    5 tests in __main__
 5 tests in 1 items.
 5 passed and 0 failed.
 Test passed.
```

END

CONTINUED