Version 14/05/2021 14:32

CSC1015F Assignment 7: Testing

Assignment Instructions

This assignment involves constructing tests, assembling automated '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.

NOTE Your solutions to this assignment will be evaluated for correctness and for the following qualities:

- Documentation
  - Use of comments at the top of your code to identify program purpose, author and date.
  - Use of comments within your code to explain each non-obvious functional unit of code.
- General style/readability
  - The use of meaningful names for variables and functions.
- Algorithmic qualities
  - Efficiency, simplicity

These criteria will be manually assessed by a tutor and commented upon. In this assignments, up to 10 marks will be deducted for deficiencies.

Question 1 [30 marks]

This question concerns devising a set of tests for a Python function that cumulatively achieve path coverage.

The Vula page for this assignment provides a Python module called 'numberutil.py' as an attachment. The module contains a function called 'aswords'. The function accepts an integer (0-999) as a parameter and returns the English language equivalent.

For example, `aswords(905)` returns the string `'Nine hundred and five'`.

Your task:

1. Develop a set of 8 test cases that achieve ***path coverage.***
2. Code your tests as a `doctest` script suitable for execution within Wing IDE.
3. Save your `doctest` script as 'testnumberutil.py'.

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

NOTE: the `aswords()` function is believed to be error free.

Question 2 [30 marks]

This question concerns devising a set of tests for a Python function that cumulatively achieve statement coverage.

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.

Continued

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 'a.m.' or 'p.m.'.

Leading and trailing whitespace is ignored.

```
1:15 p.m.
12:59 a.m.
11:01 p.m.
```

Examples of invalid strings:

```
01:1 a.m.
21:15 p.m.
12:61 am
111:01 p.m.
```

Your task:

4. Develop a set of 6 test cases that achieve ***statement coverage.***
5. Code your tests as a `doctest` script suitable for execution within Wing IDE (like the `testchecker.py` module described in the appendix).
6. 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.

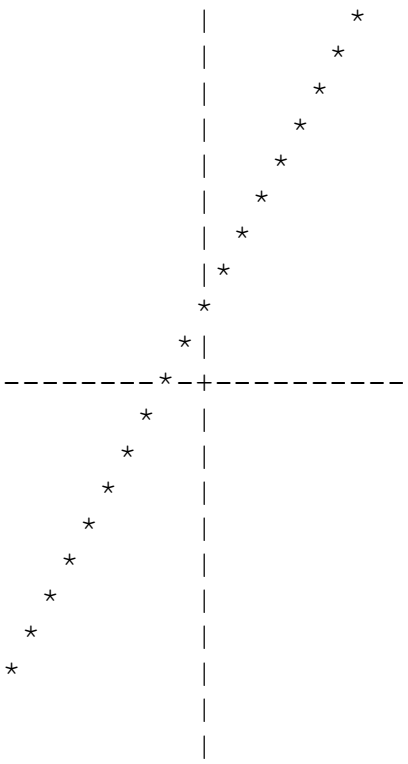
Continued

Question 3 [40 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
```



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 "*" (an asterisk), 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`, `testnumberutil.py` and `plot.py`.

END

(*Appendix starts on the next page*)

Continued

Appendix: Testing using the `doctest` module

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

Say, for instance, we have a function called '`check`' in a module called '`checker.py`':

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

We've numbered the lines for convenience.

The function is supposed to behave as follows: if *a* is less than 25 then add 1 to the *result*. If *b* is less than 25 then add 2 to the *result*. Possible outcomes are 0, 1, 2, or 3.

For the sake of realism, that there's an error in the code. Line 6 should be '`result=result+1`'.)

Here are a set of tests devised to achieve path coverage:

test #	*Path(lines executed)*	*Inputs (a,b)*	*Expected Output*
1	3, 4, 5, 6, 7, 8, 9	(20, 20)	3
2	3, 4, 5, 6, 7, 9	(20, 30)	1
3	3, 4, 5, 7, 8, 9	(30, 20)	2
4	3, 4, 5, 7, 9	(30, 30)	0

Note that we analyse the code to identify paths and select inputs that will cause that path to be followed. Given the inputs for a path, we study the function *specification* (the description of its intended behaviour) to determine the expected output.

Here is a `doctest` script for running these tests:

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

The text looks much like the transcript of an interactive session in the Python shell.

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.

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):

Continued

```
# 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.

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)
```

Continued

```
 Expected:
     1
 Got:
     3
 Trying:
     checker.check(30, 20)
 Expecting:
     2
 ok
 Trying:
     checker.check(30, 30)
 Expecting:
     0
 ok

**********************************************************************
********
 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.
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:
```

Continued

```
    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