CSC1015F Assignment 8B (70 Marks)

Testing

Learning Objectives

By the end of this assignment, you should be able to:

- Understand the need for Testing and Debugging.
- Identify the different types of errors.
- Understand the different testing methods.
- Understand how to trace through a program.
- Use a debugger to check for errors.

Assignment Instructions

This assignment involves constructing tests, assembling automated '`doctest`' test scripts for statement and path coverage.

READ the APPENDIX on pages 6 to 9 before attempting the questions in this Assignment.

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 assignment, up to 10 marks will be deducted for deficiencies.

Question 1 [20 marks]

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

The Amathuba page for this assignment provides a Python module called '`dateutil.py`' as an attachment. The module contains a function called '`format_date`'. The function accepts day, month and year as parameters. It formats the date into a string like "15 April 2025" and returns the date.

For example, `format_date(15, 3, 2024)` returns the string`'15 March 2024'`.

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 (see the appendix).
3. Save your `doctest` script as '`testdateutil.py`'.

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

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

Question 2 [20 marks]

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

The Amathuba page for this assignment provides a Python module called '`romanutil.py`' as an attachment. The module contains a function called '`from_roman()`'. The function accepts a Roman numeral string value (e.g. 'XIV') as a parameter and return the corresponding integer value.

For example, `from_roman('IX')` returns 9.

Your task:

4. Develop a set of 6 test cases that achieve ***path 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 '`testromanutil.py`'.

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

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

Question 3 [15 marks]

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

The Amathuba page for this assignment provides a Python module called '`palindrome.py`' as an attachment. The module contains a function called '`is_palindrome()`'. The purpose of this function is to accept a string value as a parameter and determine whether it is a palindrome or not.

A palindrome is a word, phrase, number, or other sequence of symbols or elements, whose meaning may be interpreted the same way in either forward or reverse direction (see Wikipedia).

Examples of palindromes are (note that white spaces are ignored – see the last examples):

```
989
a
aba
racecar
a santa at nasa
```

Examples of strings that are not palindromes (note case sensitivity – last two examples):

```
34563
ab
Racecar
A Santa at NASA
```

Your task:

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

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

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

Question 4 [15 marks]

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

The Amathuba 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 '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.

Submission

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

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 Amathuba 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.

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

**************************************************************
********
 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:
    0
ok
1 items passed all tests:
   5 tests in __main__
5 tests in 1 items.
5 passed and 0 failed.
Test passed.
```

APPENDIX ENDS