CSC1015F Assignment 6: Testing and Arrays

Assignment Instructions

This assignment involves constructing tests, assembling automated 'doctest' test scripts, and developing and reasoning about Python programs that use arrays (lists), input and output statements, 'if' and 'if-else' control flow statements, 'while' statements, 'for' statements, and statements that perform numerical manipulation.

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'.

You will find a tutorial on testing using `doctest` scripts in the appendix.

NOTE:

- Make sure the docstring in your script contains a blank line before the closing """. (The automarker requires it.)
- the `aswords()` function is believed to be error free.

Question 2 [20 marks]

Write a Python program called 'right-align.py' where the user can enter a list of strings followed by the sentinel "DONE" and the list of strings is then printed out right aligned with the longest string.

Sample I/O (user input is in bold):

Enter strings (end with DONE):

Stuart
Masixole
Milan
Joachim
Hanan
Caitlin
Molefe
Jason
Jacob
Mbongeni

CONTINUED

DONE

```
Right-aligned list:
  Stuart
Masixole
   Milan
 Joachim
   Hanan
 Caitlin
  Molefe
   Jason
   Jacob
Mbongeni
```

Question 3 [25 marks]

Write a program called '`pascal.py`' that uses arrays to print out Pascal's triangle efficiently.

Pascal's triangle contains the binomial coefficients for a sequence of binomial powers. An example triangle is the following:

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1

Each value is equal to the sum of the values immediately above it and above to the left.

HINTS:

- One possible solution is to start with the first row as the 'current row', your program should print it, then calculate and print the next row, which then becomes the current row for the next row after that!
- Consider making a function that accepts a row, *n*, as a parameter, and that calculates and returns row *n+1*.

Sample I/O (user input is in bold):

```
Enter the height of the triangle:
6
The triangle is:
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

The values on a row are separated by a single space.

CHALLENGE (not for submissions or marks): Can you make a new version of the program that will, for any height triangle, right-justify values such that columns are aligned? For example:

```
Enter the height of the triangle:
6
The triangle is:
 1
 1  1
 1  2  1
 1  3  3  1
 1  4  6  4  1
 1  5 10 10  5  1
```

Question 4 [25 marks]

Write a program called '`vector.py`' that uses a Python module called '`vectormaths.py`' to do basic vector calculations in 3 dimensions: addition, dot product and normalization.

A vector has 3 component values, such as *(1, 3, 2)* and is naturally storable as an array.

- Addition of vectors requires addition of the corresponding elements.
- A dot product is the sum of the products of corresponding elements.
- The norm of a single vector is the square root of the sum of the squares of the elements.

Suppose that we have 2 vectors: *A=(1, 3, 2)* and *B=(2, 3, 0)*:

- Addition:
  A+B = (1+2, 3+3, 2+0) = (3, 6, 2)
- Dot product:
  A.B = 1.2 + 3.3 + 2.0 = 2 + 9 = 11
- Norm (of *A*):
  |A| = Sqrt(1^2 + 3^2 + 2^2) = Sqrt(1+9+4) = Sqrt(14) = 3.74
- Norm (of B):
  |B| = Sqrt(2^2 + 3^2 + 0^2) = Sqrt(4+9+0) = Sqrt(13) = 3.61

The `vectormaths.py` module must contain the following functions:

- `vector_sum (A, B)`
  Accepts as parameters, two three-element arrays, A and B, that represent vectors. Calculates and returns a three-element array representing the vector that is the sum of vectors A and B.
- `vector_product (A, B)`
  Accepts as parameters, two three-element arrays, A and B, that represent vectors. Calculates and return the value that is the dot product of vectors A and B.
- `vector_norm (A)`
  Accepts as a parameter a three-element array, A, representing a vector. Calculates and return the norm of vector A.

Examples

- `vector_sum([1, 3, 2], [2, 3, 0])` returns the array `[3, 6, 2]`.
- `vector_product([1, 2, 3], [2, 3, 0])` returns the value `11`.
- `vector_norm([1, 2, 3])` returns the value `3.7416573867739413`.

The `vector.py` program will ask the user to enter vector A, vector B and to choose a calculation to perform. It will use the appropriate `vectormaths.py` function to perform the calculation and will then print the result.

Sample I/O (user input is in bold):

```
Enter vector A:
1 3 2
Enter vector B:
2 3 0
Select a calculation to perform. Enter '+', '.' or '|':
+
A+B = [3, 6, 2]
```

Sample I/O (user input is in bold):

```
Enter vector A:
1 3 2
Enter vector B:
2 3 0
Select a calculation to perform. Enter '+', '.' or '|':
.
A.B = 11
```

Sample I/O (user input is in bold):

```
Enter vector A:
1 3 2
Enter vector B:
2 3 0
Select a calculation to perform. Enter '+', '.' or '|':
|
|A| = 3.74
|B| = 3.61
```

Sample I/O (user input is in bold):

```
Enter vector A:
1 3 2
Enter vector B:
2 3 0
Select a calculation to perform. Enter '+', '.' or '|':
sldkhf
Selection not recognised.
```

HINT: The expression `"{:.2f}".format(3.7416573867739413)` produces the value `3.74`.

Submission

Create and submit a Zip file called '`ABCXYZ123.zip`' (where ABCXYZ123 is YOUR student number) containing `testnumberutil.py`, `right-align.py`, `pascal.py`, and `vector.py` and `vectormaths.py`.

END

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`.

CONTINUED

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):

```
# 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:
```

CONTINUED

```
     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:
```

CONTINUED

```
    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