Worksheet 8: White-Box Testing and Test Fixtures

Thi Van Anh DUONG Student ID: 90023112

Diploma of Information Technology, Curtin College

ISEN1000: Introduction to Software Engineering

Coordinator: Khurram Hameed

9 May 2022

Student Declaration of Originality

☒	This assignment is my own original work, and no part has been copied from another student's work or source, except where it is clearly attributed.
☒	All facts/ideas/claims are from academic sources are paraphrased and cited/referenced correctly.
☒	I have not previously submitted this work in any form for THIS or for any other unit; or published it elsewhere before.
☒	No part of this work has been written for me by another person.
☒	I recognise that should this declaration be found to be false, disciplinary action could be taken and penalties imposed in accordance with Curtin College policy.

Electronic Submission:

☒	I accept that it is my responsibility to check that the submitted file is the correct file, is readable and has not been corrupted.
☒	I acknowledge that a submitted file that is unable to be read cannot be marked and will be treated as a non-submission.
☒	I hold a copy of this work if the original is damaged, and will retain a copy of the Turnitin receipt as evidence of submission.

1. printCoordinates()

`printCoordinates()` takes in *x, y* and *z* coordinates, and prints them out in the format **(*x, y, z*)**, with two decimal places each. The output will end in a new line.

(This is a trivial case for test design. Since there are no conditional statements, there is only one path, and hence one test case.)

```
def printCoordinates(x, y, z):
    print(”({:.2f}, {:.2f}, {:.2f})”.format(x, y, z))
```

(a) Design your test cases:

- Identify the paths through the production code.
- Select test data for each test case. In other words, for each path, select inputs (parameters, console input, and/or input files) that will cause the production code to follow that path.
- For each test case, determine the expected results. This includes return values, exceptions thrown, console output, and output files.

➔ Number of paths: 1

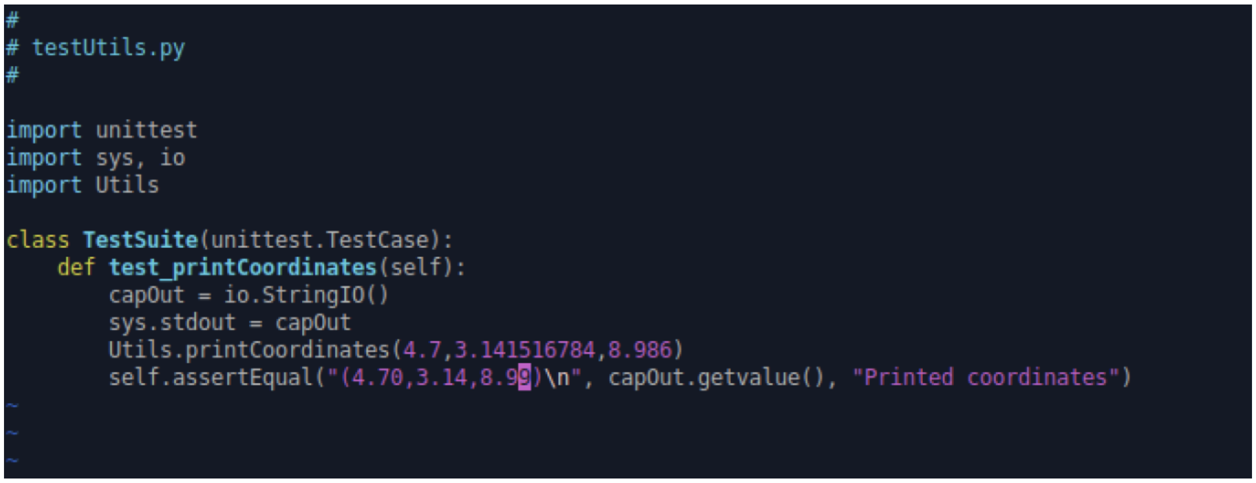
```
def printCoordinates(x, y, z):
    print("({:.2f},{:.2f},{:.2f})".format(x, y, z))
```

➔

Path	Test Data	Expected result
Print statement	x = 4.7, y = 3.141516784, z = 8.986	Output: “(4.70, 3.14, 8.99)\n”

(b) Implement your test cases.

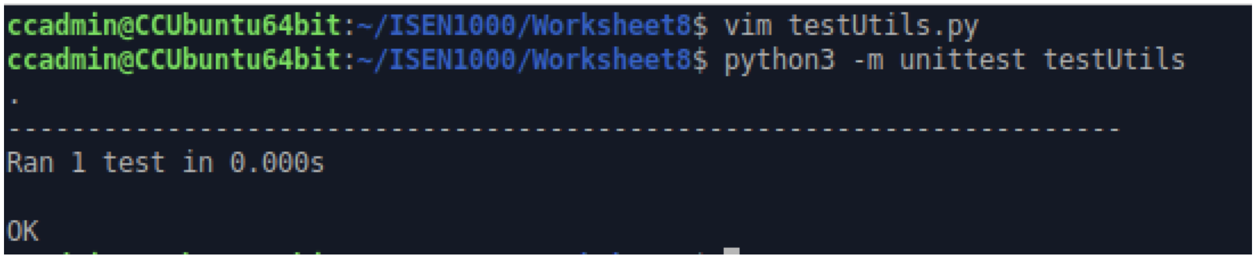
It's good experience to continue to use the `unittest` module, although you can still perform the exercise without it.



```
#
# testUtils.py
#

import unittest
import sys, io
import Utils

class TestSuite(unittest.TestCase):
    def test_printCoordinates(self):
        capOut = io.StringIO()
        sys.stdout = capOut
        Utils.printCoordinates(4.7,3.141516784,8.986)
        self.assertEqual("(4.70,3.14,8.99)\n", capOut.getvalue(), "Printed coordinates")
~
~
~
```

(c) Run your tests against the production code.

To ensure that your test code is working, it's helpful to temporarily *break* the production code. You could do this by editing the production code to alter the output/results very slightly.



```
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ vim testUtils.py
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ python3 -m unittest testUtils
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

2. readChar()

`readChar()` reads a single "valid" character from the user. The user enters a character, but must re-enter their input if it's invalid; i.e., if the character they enter does not occur within the `validChars` parameter.

Hint: there are two paths, and hence two test cases here.

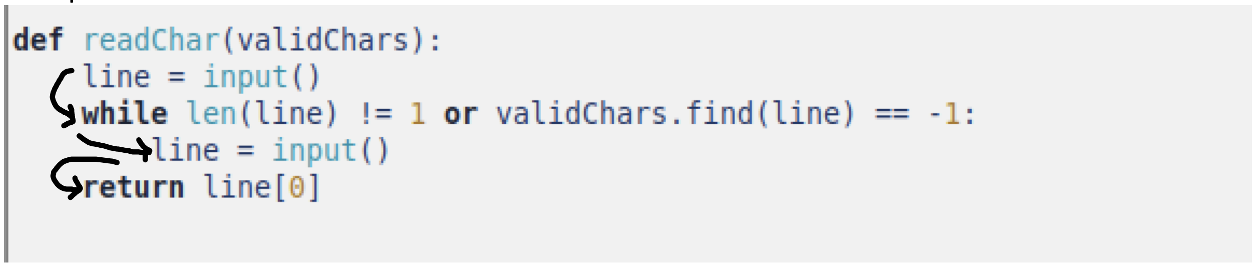
```
def  readChar(validChars):
    line = input()
    while len(line) != 1 or validChars.find(line) == -1:
        line = input()
    return line[0]
```

(a) Design your test cases:

- Identify the paths through the production code.
- Select test data for each test case. In other words, for each path, select inputs (parameters, console input, and/or input files) that will cause the production code to follow that path.
- For each test case, determine the expected results. This includes return values, exceptions thrown, console output, and output files.

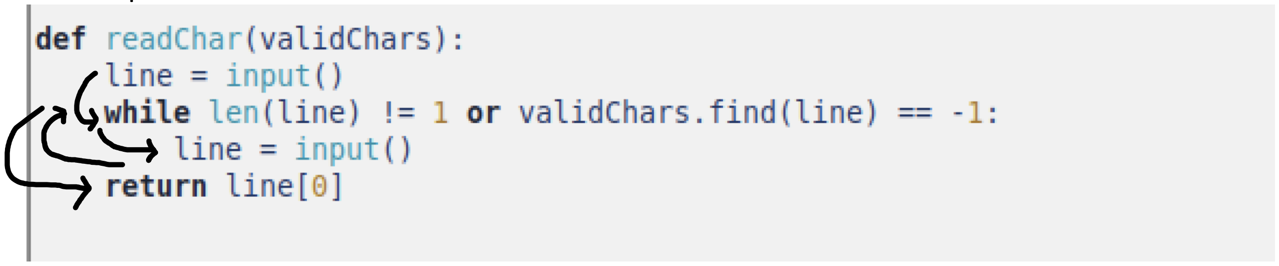
➔ Number of paths: 2

First path:



```
def readChar(validChars):
    line = input()
    while len(line) != 1 or validChars.find(line) == -1:
        line = input()
    return line[0]
```

Second path:

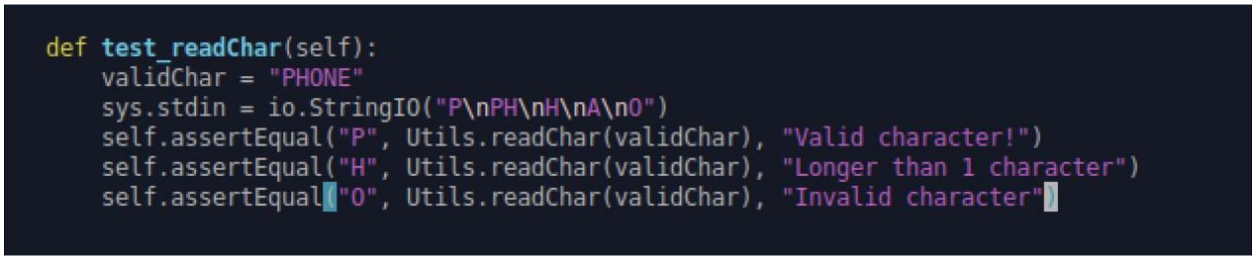


```
def readChar(validChars):
    line = input()
    while len(line) != 1 or validChars.find(line) == -1:
        line = input()
    return line[0]
```

Path	Test Data	Expected result
Skip loop	P	P
Enter while loop (len(line) != 1)	PH, H	H
Enter while loop (character is not include in the char validChars.find(line) == -1)	A, O	O

(b) Implement your test cases.

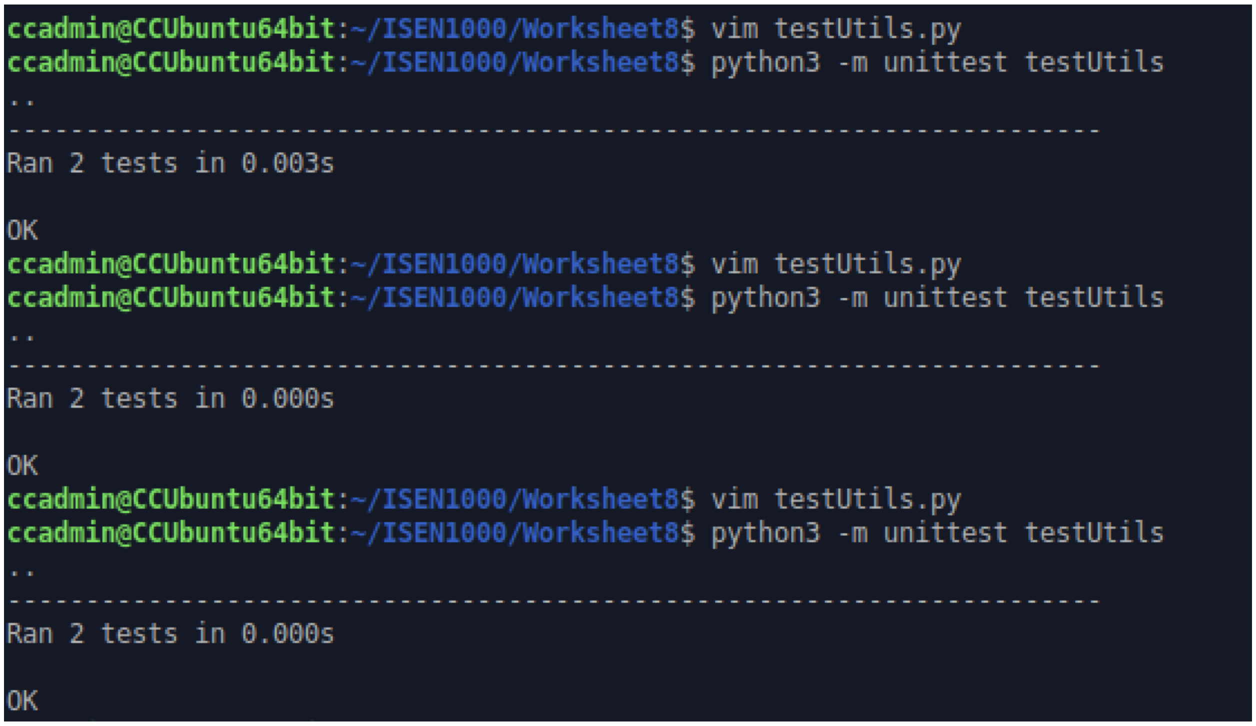
It's good experience to continue to use the `unittest` module, although you can still perform the exercise without it.



```
def test_readChar(self):
    validChar = "PHONE"
    sys.stdin = io.StringIO("P\nPH\nH\nA\nO")
    self.assertEqual("P", Utils.readChar(validChar), "Valid character!")
    self.assertEqual("H", Utils.readChar(validChar), "Longer than 1 character")
    self.assertEqual("O", Utils.readChar(validChar), "Invalid character")
```

(c) Run your tests against the production code.

To ensure that your test code is working, it's helpful to temporarily *break* the production code. You could do this by editing the production code to alter the output/results very slightly.



```
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ vim testUtils.py
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ python3 -m unittest testUtils
..
----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ vim testUtils.py
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ python3 -m unittest testUtils
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ vim testUtils.py
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ python3 -m unittest testUtils
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

3. guessingGame()

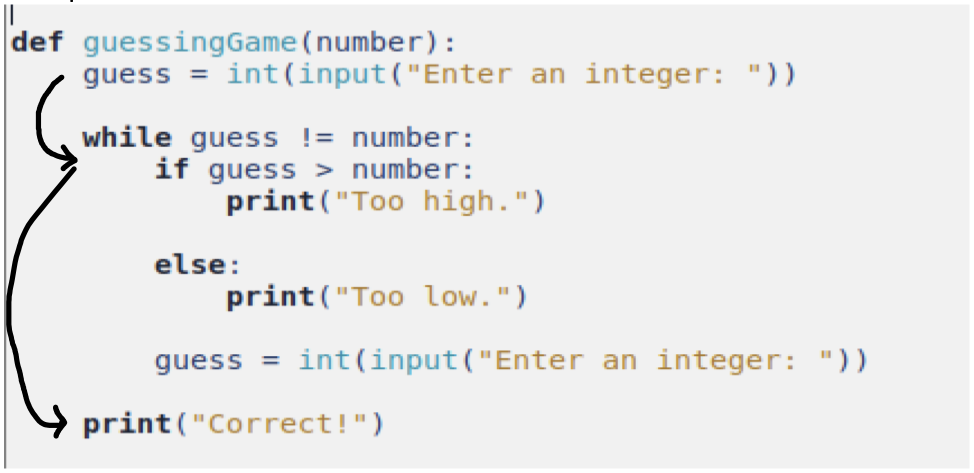
guessingGame() runs a console-based number guessing game. The user is repeatedly asked to guess what the number is, and is told whether their guess is too high, too low, or correct (at which point the game ends).

(a) Design your test cases:

- Identify the paths through the production code.
- Select test data for each test case. In other words, for each path, select inputs (parameters, console input, and/or input files) that will cause the production code to follow that path.
- For each test case, determine the expected results. This includes return values, exceptions thrown, console output, and output files.

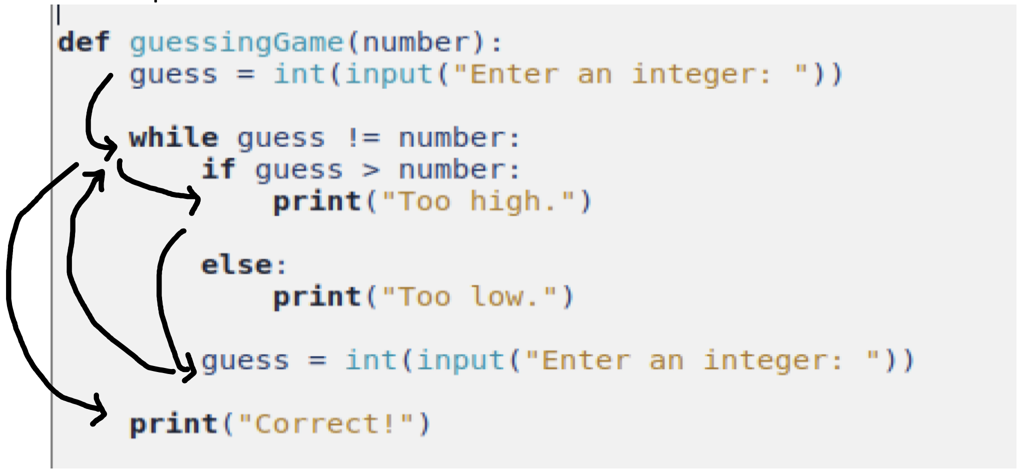
➔ Number of paths: 3

First path:



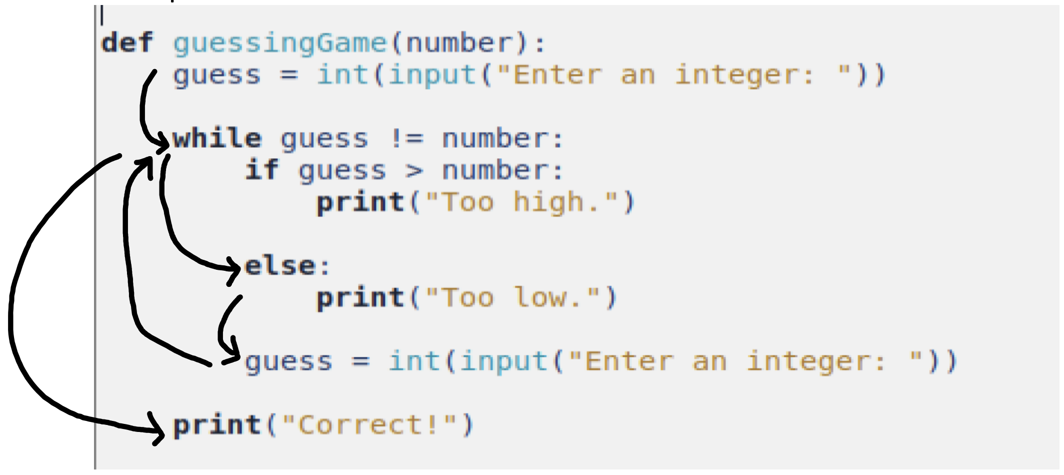
```python
def guessingGame(number):
    guess = int(input("Enter an integer: "))

    while guess != number:
        if guess > number:
            print("Too high.")

        else:
            print("Too low.")

        guess = int(input("Enter an integer: "))

    print("Correct!")
```

Second path:



```python
def guessingGame(number):
    guess = int(input("Enter an integer: "))

    while guess != number:
        if guess > number:
            print("Too high.")

        else:
            print("Too low.")

        guess = int(input("Enter an integer: "))

    print("Correct!")
```

Third path:

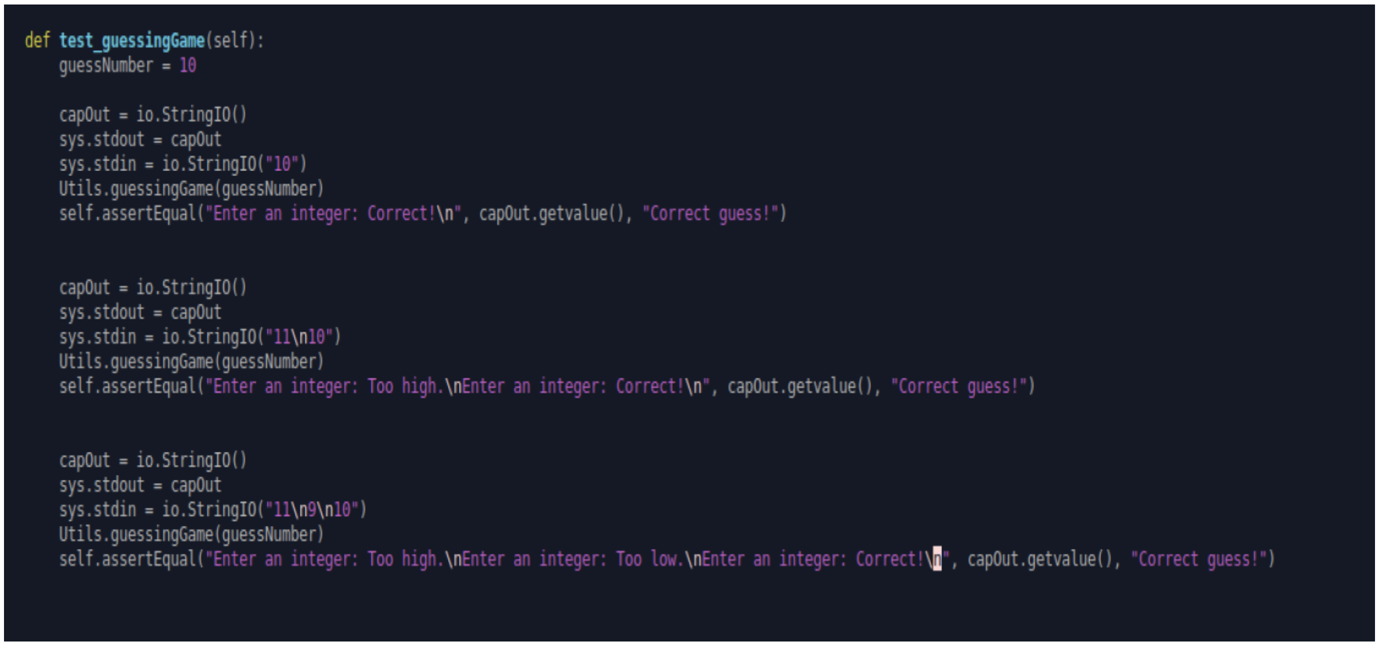


```
def guessingGame(number):
    guess = int(input("Enter an integer: "))

    while guess != number:
        if guess > number:
            print("Too high.")

        else:
            print("Too low.")

        guess = int(input("Enter an integer: "))

    print("Correct!")
```

Path	Test Data	Expected result
Skip loop	10	Correct!
Enter if statement	11, 10	Correct!
Enter else statement	11,9, 10	Correct!

(b) Implement your test cases.

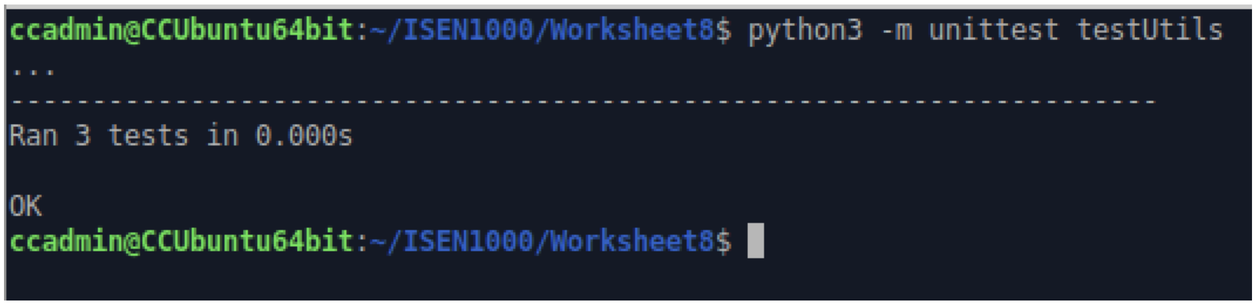
It's good experience to continue to use the `unittest` module, although you can still perform the exercise without it.



```
def test_guessingGame(self):
    guessNumber = 10

    capOut = io.StringIO()
    sys.stdout = capOut
    sys.stdin = io.StringIO("10")
    Utils.guessingGame(guessNumber)
    self.assertEqual("Enter an integer: Correct!\n", capOut.getvalue(), "Correct guess!")


    capOut = io.StringIO()
    sys.stdout = capOut
    sys.stdin = io.StringIO("11\n10")
    Utils.guessingGame(guessNumber)
    self.assertEqual("Enter an integer: Too high.\nEnter an integer: Correct!\n", capOut.getvalue(), "Correct guess!")


    capOut = io.StringIO()
    sys.stdout = capOut
    sys.stdin = io.StringIO("11\n9\n10")
    Utils.guessingGame(guessNumber)
    self.assertEqual("Enter an integer: Too high.\nEnter an integer: Too low.\nEnter an integer: Correct!\n", capOut.getvalue(), "Correct guess!")
```

(c) Run your tests against the production code.

To ensure that your test code is working, it's helpful to temporarily *break* the production code. You could do this by editing the production code to alter the output/results very slightly.



```
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ python3 -m unittest testUtils
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ 
```

4. sumFile()

sumFile(): opens a file (assumed to contain a list of numbers), and adds up the numbers, and returns the total. If the file could not be opened, it returns -1 instead.

Hint: the empty string "" is an invalid filename that, by definition, cannot be opened.

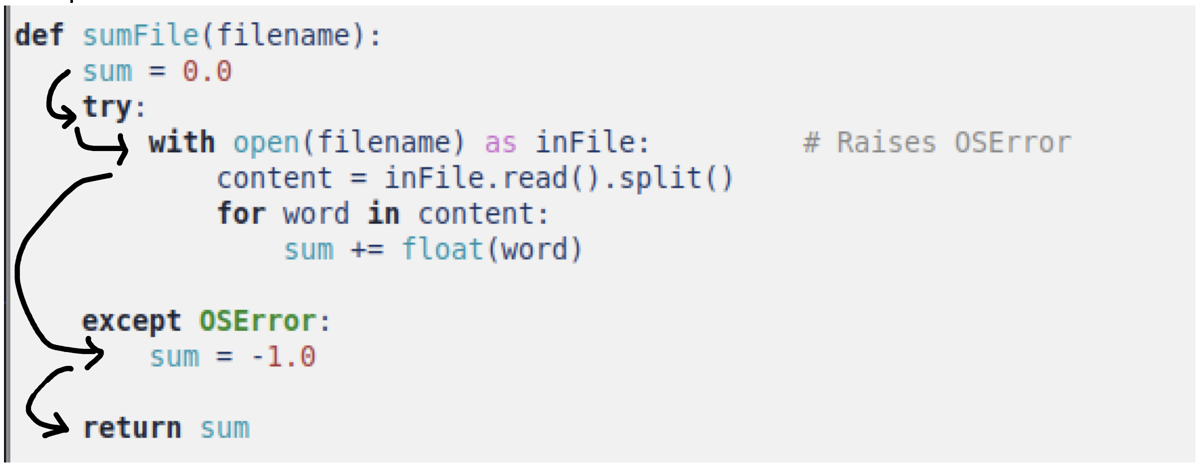
```python
def sumFile(filename):
    sum = 0.0
    try:
        with open(filename) as inFile:          # Raises OSError
            content = inFile.read().split()
            for word in content:
                sum += float(word)

    except OSError:
        sum = -1.0

    return sum
```

(a) Design your test cases:

- Identify the paths through the production code.
- Select test data for each test case. In other words, for each path, select inputs (parameters, console input, and/or input files) that will cause the production code to follow that path.
- For each test case, determine the expected results. This includes return values, exceptions thrown, console output, and output files.

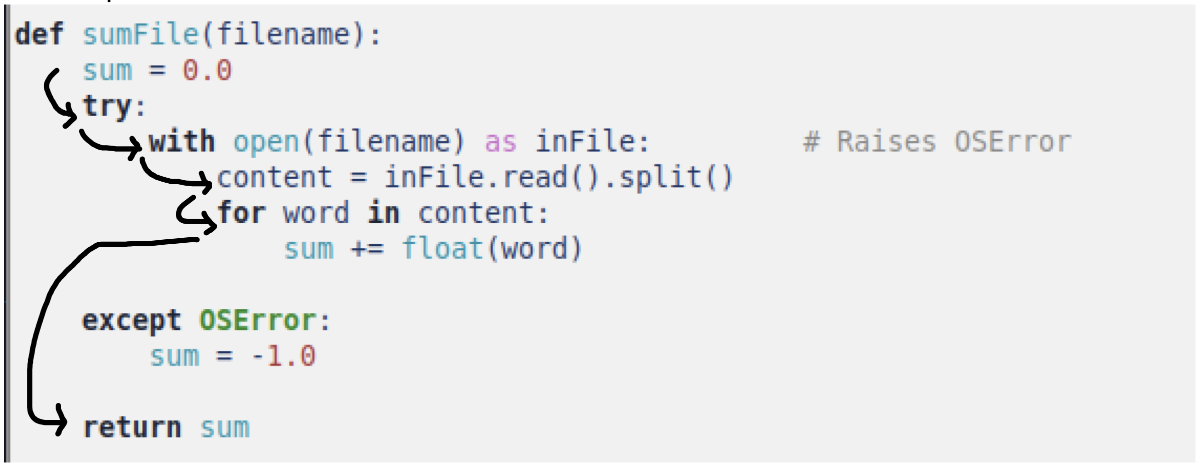
➔ Number of paths: 3

First path:



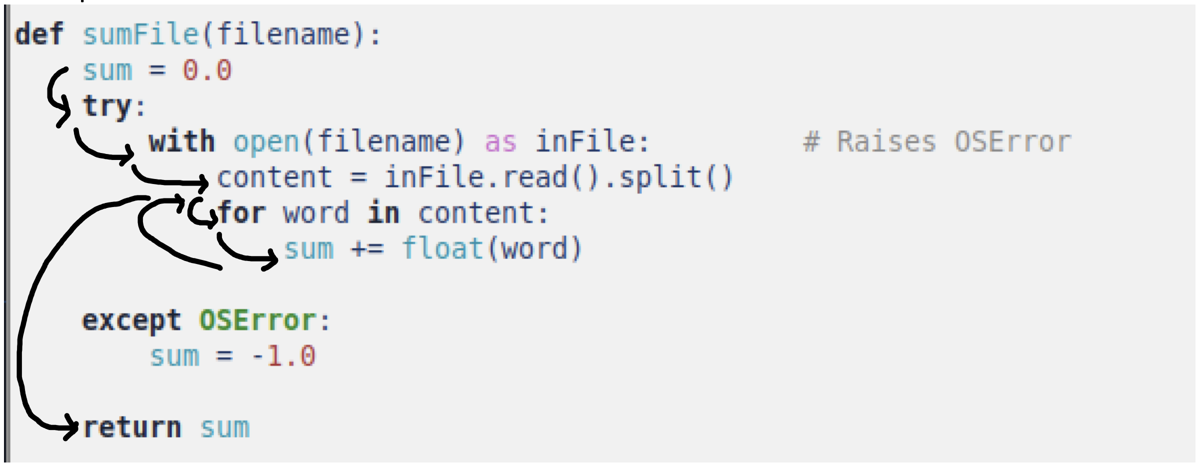
```python
def sumFile(filename):
    sum = 0.0
    try:
        with open(filename) as inFile:          # Raises OSError
            content = inFile.read().split()
            for word in content:
                sum += float(word)

    except OSError:
        sum = -1.0

    return sum
```

Second path:



```python
def sumFile(filename):
    sum = 0.0
    try:
        with open(filename) as inFile:          # Raises OSError
            content = inFile.read().split()
            for word in content:
                sum += float(word)

    except OSError:
        sum = -1.0

    return sum
```

Third path:

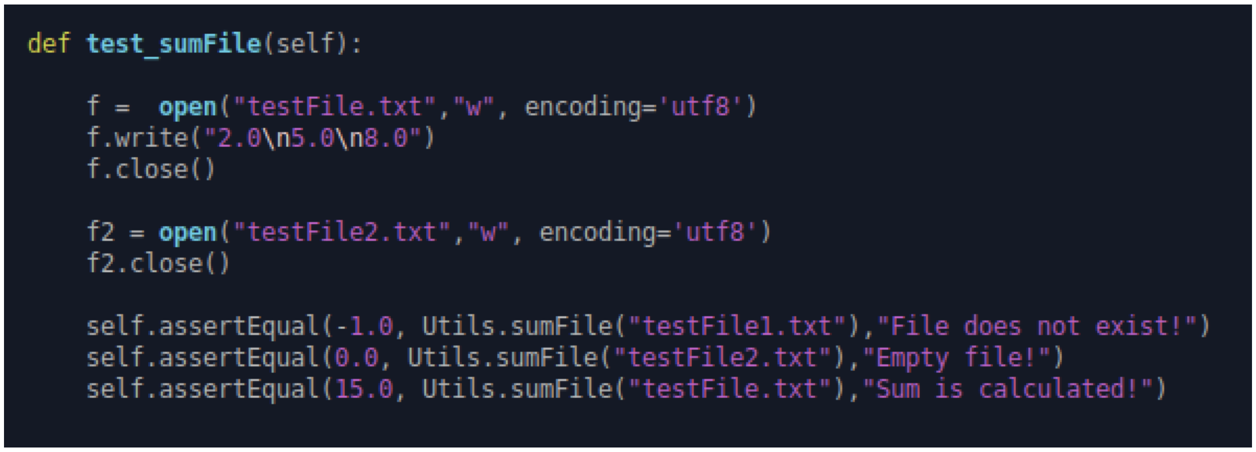


```python
def sumFile(filename):
    sum = 0.0
    try:
        with open(filename) as inFile:          # Raises OSError
            content = inFile.read().split()
            for word in content:
                sum += float(word)

    except OSError:
        sum = -1.0

    return sum
```

Path	Test Data	Expected result
Try and except	testFile1.txt	-1.0 (File does not exist)
Skip for loop	testFile2.txt	0.0 (Empty file)
Enter for loop	testFile.txt (2,5,8)	15.0 (Sum is calculated!)

(b) Implement your test cases.

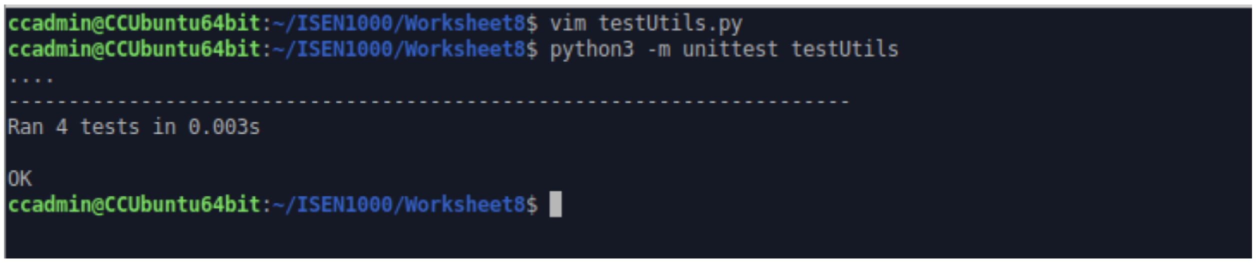
It's good experience to continue to use the `unittest` module, although you can still perform the exercise without it.



```python
def test_sumFile(self):

    f =  open("testFile.txt","w", encoding='utf8')
    f.write("2.0\n5.0\n8.0")
    f.close()

    f2 = open("testFile2.txt","w", encoding='utf8')
    f2.close()

    self.assertEqual(-1.0, Utils.sumFile("testFile1.txt"),"File does not exist!")
    self.assertEqual(0.0, Utils.sumFile("testFile2.txt"),"Empty file!")
    self.assertEqual(15.0, Utils.sumFile("testFile.txt"),"Sum is calculated!")
```

(c) Run your tests against the production code.

To ensure that your test code is working, it's helpful to temporarily *break* the production code. You could do this by editing the production code to alter the output/results very slightly.



```
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ vim testUtils.py
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ python3 -m unittest testUtils
....
----------------------------------------------------------------------
Ran 4 tests in 0.003s

OK
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ 
```

5. saveData()

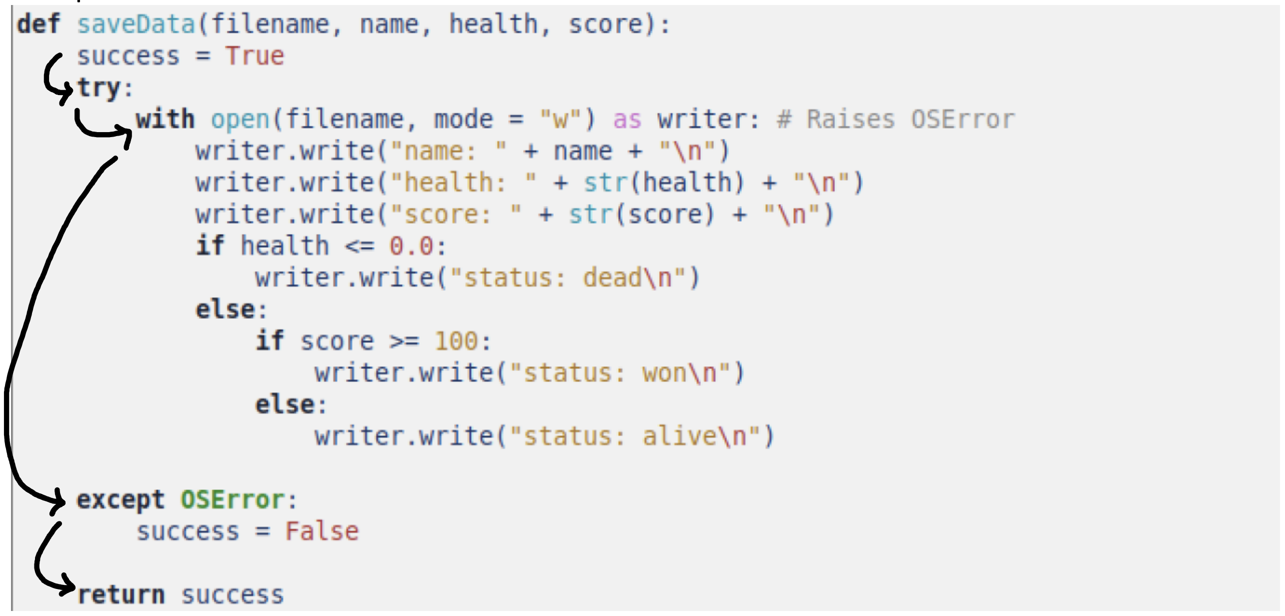
`saveData()` opens a given file, and writes several lines of text to it. Returns `true` if the file could be written, and `false` if not (e.g., because an invalid name was given).

(a) Design your test cases:

- Identify the paths through the production code.
- Select test data for each test case. In other words, for each path, select inputs (parameters, console input, and/or input files) that will cause the production code to follow that path.
- For each test case, determine the expected results. This includes return values, exceptions thrown, console output, and output files.

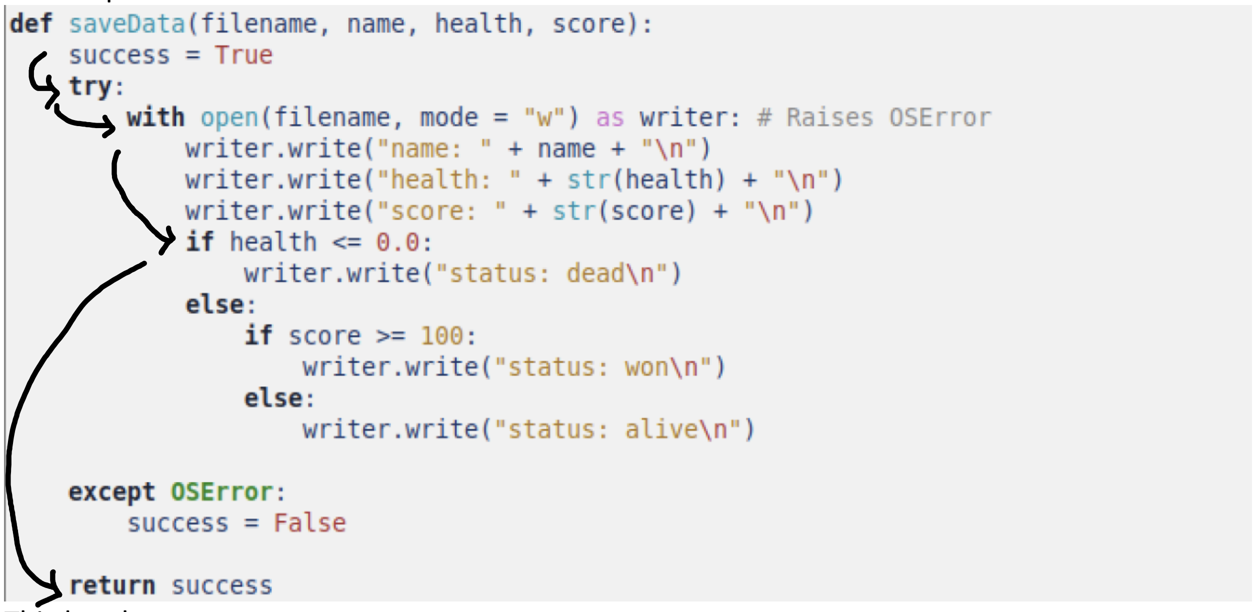
➔ Number of paths: 4

First path:



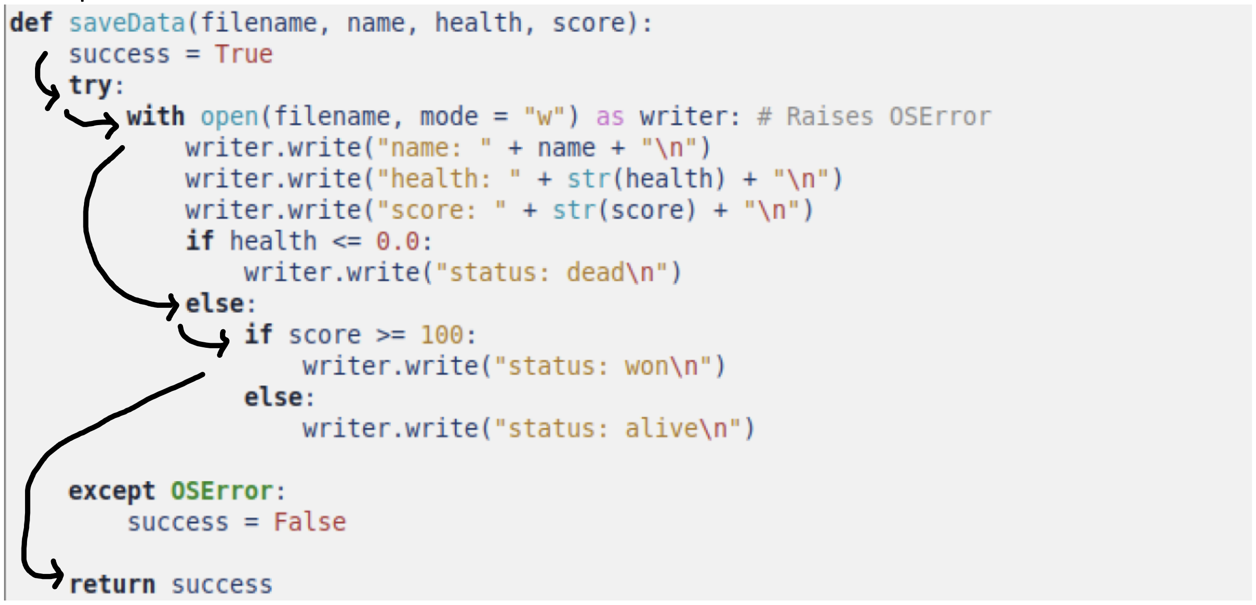
```python
def saveData(filename, name, health, score):
    success = True
    try:
        with open(filename, mode = "w") as writer: # Raises OSError
            writer.write("name: " + name + "\n")
            writer.write("health: " + str(health) + "\n")
            writer.write("score: " + str(score) + "\n")
            if health <= 0.0:
                writer.write("status: dead\n")
            else:
                if score >= 100:
                    writer.write("status: won\n")
                else:
                    writer.write("status: alive\n")

    except OSError:
        success = False

    return success
```

Second path:



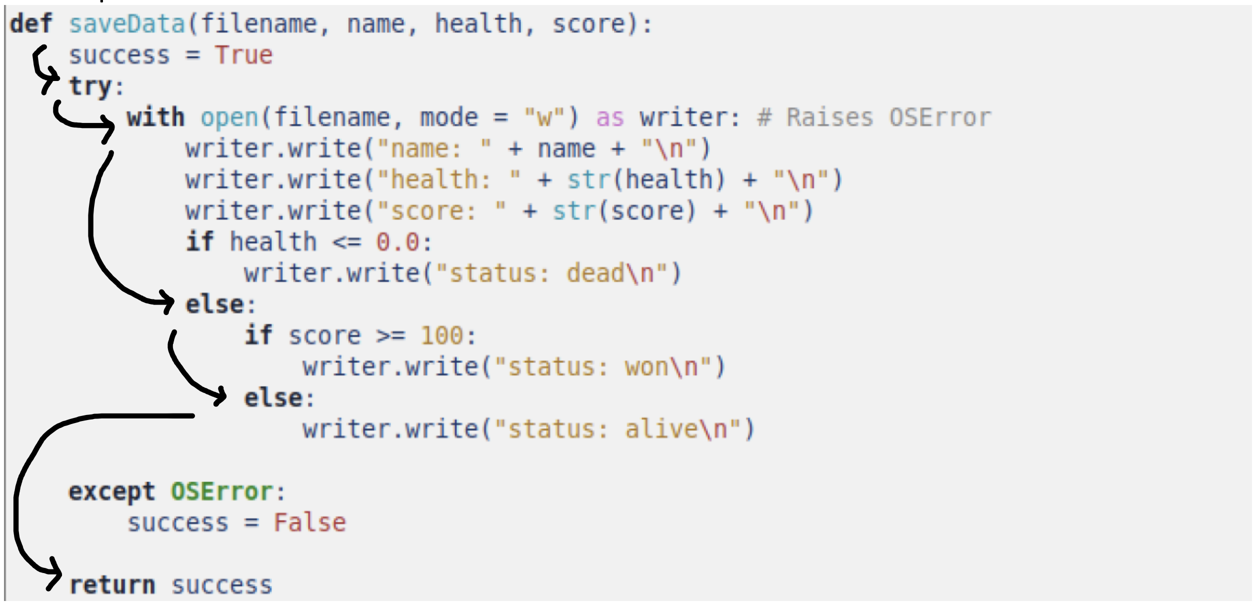
```python
def saveData(filename, name, health, score):
    success = True
    try:
        with open(filename, mode = "w") as writer: # Raises OSError
            writer.write("name: " + name + "\n")
            writer.write("health: " + str(health) + "\n")
            writer.write("score: " + str(score) + "\n")
            if health <= 0.0:
                writer.write("status: dead\n")
            else:
                if score >= 100:
                    writer.write("status: won\n")
                else:
                    writer.write("status: alive\n")

    except OSError:
        success = False

    return success
```

Third path:



```python
def saveData(filename, name, health, score):
    success = True
    try:
        with open(filename, mode = "w") as writer: # Raises OSError
            writer.write("name: " + name + "\n")
            writer.write("health: " + str(health) + "\n")
            writer.write("score: " + str(score) + "\n")
            if health <= 0.0:
                writer.write("status: dead\n")
            else:
                if score >= 100:
                    writer.write("status: won\n")
                else:
                    writer.write("status: alive\n")

    except OSError:
        success = False

    return success
```

Fourth path

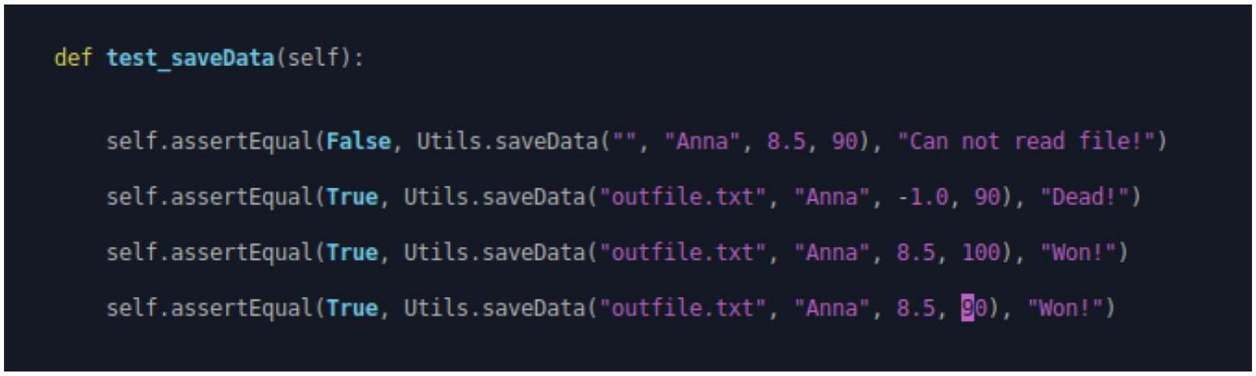


```python
def saveData(filename, name, health, score):
    success = True
    try:
        with open(filename, mode = "w") as writer: # Raises OSError
            writer.write("name: " + name + "\n")
            writer.write("health: " + str(health) + "\n")
            writer.write("score: " + str(score) + "\n")
            if health <= 0.0:
                writer.write("status: dead\n")
            else:
                if score >= 100:
                    writer.write("status: won\n")
                else:
                    writer.write("status: alive\n")

    except OSError:
        success = False

    return success
```

Path	Test Data	Expected result
Try and except	outFile.txt, Anna, 8.5, 90	success = False
Enter if statement	outFile.txt, Anna, -1.0,90	success = True
(else) Enter if statement	outFile.txt, Anna, 8.5, 100	success = True
(else) Enter else statement	outFile.txt, Anna, 8.5, 90	success = True

(b) Implement your test cases.

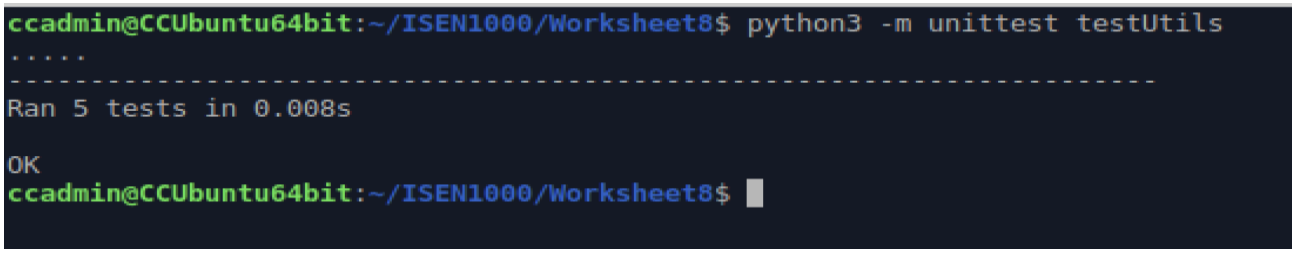
It's good experience to continue to use the `unittest` module, although you can still perform the exercise without it.



```python
def test_saveData(self):

    self.assertEqual(False, Utils.saveData("", "Anna", 8.5, 90), "Can not read file!")

    self.assertEqual(True, Utils.saveData("outfile.txt", "Anna", -1.0, 90), "Dead!")

    self.assertEqual(True, Utils.saveData("outfile.txt", "Anna", 8.5, 100), "Won!")

    self.assertEqual(True, Utils.saveData("outfile.txt", "Anna", 8.5, 90), "Won!")
```

(c) Run your tests against the production code.

To ensure that your test code is working, it's helpful to temporarily *break* the production code. You could do this by editing the production code to alter the output/results very slightly.



```
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ python3 -m unittest testUtils
.....
----------------------------------------------------------------------
Ran 5 tests in 0.008s

OK
ccadmin@CCUbuntu64bit:~/ISEN1000/Worksheet8$ 
```