Problem 1

Consider the following Python code fragment:

```
print("Division Calculator: op1 // op2")
equation = input("Please, enter integers op1 and op2 : ")
if equation.isalpha( ) :
    print("is alpha!")
else :
    theOps = equation.split() # default is whitespace
    if len(theOps) == 0 :
        print("Nothing!")
    elif len(theOps) == 1 :
        print("Only 1 op!")
    elif len(theOps) > 2 :
        print("Too many ops!")
    elif int(theOps[1]) == 0 :
        print("Division by 0!")
    else :
        print("{} // {} = {}".format(theOps[0], theOps[1],
                                     int(theOps[0]) // int(theOps[1]) ))
```

In the table below, create the minimum number of **test cases** we would need to completely test our Python code fragment above.

To completely test means that all statements of our Python code fragment have been executed, i.e., tested. This does not mean that each test case executes all the statements. That would be impossible. It means that once all the test cases have been used to test our Python code fragment, they have execute all its statements.

Test Case #	**Test Data**	**Expected Results**

Problem 2.

Write a **Python program** that adds a tax to a given price. Your program must ask the user for a price (float), then a tax (float). The format of the tax to be entered is, for example, 0.12 instead of 12%. The user will always enter a valid positive float value as a price and as a tax.

For example:

- if the user enters the price 2.65 and the tax 0.07 (i.e., 7%), then your program produces and prints 2.84.
- if the user enters the price 12.85 and the tax 0.12 (i.e., 12%), then your program produces and prints 14.39.

Hint:

- You may appreciate the built-in function round(<someResult>,2).

If you find it useful to start by designing an algorithm, feel free to do so.

Problem 3

From our e-textbook: http://www.greenteapress.com/thinkpython2/thinkpython2.pdf

The following example shows how to use concatenation (string addition) and a `for` loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print(letter + suffix)
```

The output is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because "Ouack" and "Quack" are misspelled. As an exercise, modify the program to fix this error.

Problem 4

Problem Statement: Write a **complete Python program** that figures out how many upper case letters appear in a sentence. For the purpose of this program, we define a sentence to be a sequence of two or more words separated by one blank space (or white space) character and a word to be a sequence of one or more letters.

Here are four **sample runs**:

- If the sentence is "`ThE SKy Is BluE.`", our program prints "`ThE SKy Is BluE. contains 7 upper case letter(s).`" on the computer monitor screen.
- If the sentence is "`YeLlOw bAnAnA`", our program prints "`YeLlOw bAnAnA contains 6 upper case letter(s).`" on the computer monitor screen.
- If the sentence is "" (i.e., an empty string), our program prints "`This is not a sentence.`" on the computer monitor screen.
- If the sentence is "`baNana`", our program prints "`This is not a sentence.`" on the computer monitor screen.

Note:

- You can assume that the user is "well-behaved", i.e., s/he will enter only the three categories of test data mentioned in the four sample runs above, namely a valid sentence (sample runs 1. and 2.), an empty sentence (a sentence containing no words – sample run 3.) and a sentence containing one word (sample run 4.).

If you find it useful to start by designing an algorithm, feel free to do so.