11256 EE20084

25 May 2021 11:02

1a)

i)

In the most simplest form docstrings are multiline comments where doctest is a module included in the standard collection of python libraries that is used to perform tests on programs, indicate expected outputs given certain inputs and their code is included within docstrings.

Docstrings:
These are multiline comments denoted by two " ''' " (triple apostrophes) that are used to initiate and terminate the segment of the comments. They are useful when commenting is likely to take multiple lines and is a great alternative to using "#" prior to every line of comment.
They are most often used when documenting code and by convention they are often defined straight after a function, class, module or method definition. Within this documentation, programmers often include a description of the input and output arguments, the description of the use and behaviour of the program, and the doctests for the function/method/class/module.

Doctests:
Doctest are used and included within docstrings, they are imported to python as a library and the main functionality to call and perform the tests are denoted by " >>> ". The expected output to the function call must follow right after. Note that the exact output must precede (whether string formatting or a list converted into a string). These are most useful for testing things that return a value. The testing of the function falls under the category unit testing as it doesnt test the entire functionality and interactions between different functions.
As programmers they can be used to check that the function is up to date and there aren't any conflicting changes.
They can be used to create a standard/criteria for the segment of code. (e.g. If you wanted to create an addition program, you know how to add using a calculator so you can write the doctests by calculating expected values)
A test object or a test file can be imported to ensure that the code runs as given with interactive examples.

ii)

Advantages of the doctest Framework:

- It encourages modularity when coding - narrows you to ensuring each individual function is performs its primary tasks and functions. Each function can be tested as a stand alone block.
- Reduces confusion:
  Firstly, as it is situated directly within the function, it is clear to see what the tests are related to.
  Secondly, it provides an extra base of understanding between programmers as providing examples is complementary to explaining functionality (increases readability). This is important in working on projects with multiple people and creating open source libraries.
- Allows easy use of importing external files to test the code.
- Easy implementation of unit testing ensuring each individual function performs as expected and easy/quick debugging to pinpoint problems.

- Efficient testing: you do not have to create extra functions/segments of code, can directly use inputs and outputs as oppose to, say, having to create an object, run its methods to format it, then test the focus method.

Disadvantages of the doctest Framework:

- Becomes unnecessarily difficult to test functions that are complex in nature and return a complex form of output.
- Can be difficult to exactly represent the expected output - e.g. For a decimal irrational output, your calculator may display up to 10 digits but python returns 15.
- Harder to test objects as usually python returns the address (by reference) of the object, not its inner abilities.
- Cannot test the inner workings of a function/method/class/module, only can test an expected output given input - hence it becomes difficult to test a function that does not return anything.
- Cannot easily test how the it interacts with other functions as the data is only returned in the terminal and not saved for use in testing with other parts of the code.

iii)

Whilst developing a comprehensive test strategy it is important to consider the objectives and requirements of the task. The idea of testing is to ensure that the program follows standards and does as intended. This also implies that the program should be free from errors or cases that can crash the code. Hence we can deduce relying solely on doctests does not provide adequate reassurance that out program will run as expected. To provide more sense of security we can improve the comprehensiveness of our testing by considering and including the following strategies:

Cases:
This is the most important thing to consider when injecting test inputs into out program as it defines the diversity of our tests. We should use a rich mixture of test types below:
- Valid Cases: These are the usual expected inputs that the program is likely to receive. These shouldn't cause any errors and give us an output that is as expected.
- Invalid Cases: Outputs that are expected to be not acceptable by the program and its intended use.
- Boundary/Edge Cases: These can be both valid and invalid. They are defined as test inputs that are valid but on the boundary of being invalid, or invalid on the boundary of being valid. E.g. If the acceptable range is 0-100, then 0 is a valid input but on the edge of being invalid.
- Extraneous: These are cases that are completely unacceptable by the program and are usually to test to see if the program would crash. E.g. Passing an object as a integer expected parameter.

Integration Testing:
This tests how the functions/methods/classes/modules work with eachother. This can be used to test subsystems within a program to ensure parts of a program work together inclusively and to test how classes interact and work with outside functions/methods.

Testing bias:
If one person creates the code and tests it, they are likely to have the same errors in testing and coding. Hence it is important to ensure that there are tests from both the programmer but someone else who knows what the function should do but did not code it. - e.g. Someone else in the team.

White box testing:

These are tests performed to check the inner workings of a program. They include tests that highlight how variables change and how the certain inputs filter through/traverse the function.

Black box testing:
Black-box testing is a method of testing that tests the functionality of a function/method/class/module without knowing/looking into the internal workings of it.

Alpha Testing:
This is where the programmer tests the entire module. The benefit is that the programmers know what requirements they have coded towards and can match the outputs to see if they've achieved their success criteria. One method is by importing the library into a proven working drone to see if any errors occur. The returned values can be monitored/documented on a computer whilst testing in the real world.

Beta Testing:
This is testing performed by a small group of people unrelated to the development process. The idea behind it is to simulate how a user would use the software and allows detection of hidden errors.

1b)
i)

The approach I am taking for calculating the voltage for a given charge percentage is using a function.

The function will have the input argument of a battery percentage.

Using this, I will define the x coordinates (battery charge percentage) and the y coordinates (battery voltage) and save them within a variable.

Using numpy, I am going to use their brilliant function that given both x coordinates and y coordinates and an input x coordinate, it will return a y coordinate using linear interpolation. I will use this to find the battery voltage given a charge percentage.

To ensure my code runs perfectly, I am going to use a range of doctests to predetermine how the function should behave with some boundary, invalid and valid test cases.

I also am going to use error checking through a try and except clause and return a simple error message in the command line if there are any problems and return 0. NOTE: numpy.interp() accepts strings as an argument as long as they represent a number.

```python
import numpy as np

    def calcVoltage(percentageCharge):
        """ Returns the voltage given a charge percentage using interpolation
        :param percentageCharge: Value to be interpolated
        :type percentageCharge: real, int, float and other
        :return: A value of the voltage that was interpolated
        :rtype: float
        >>> calcVoltage(1)
        20.0
        >>> calcVoltage(2)
        20.0
        >>> calcVoltage(0)
        0.0
        >>> calcVoltage(0.1)
        10.0
        >>> calcVoltage(0.3)
        17.0
        >>> calcVoltage(0.03)
        0.6
        >>> calcVoltage(0.05*0.5)
        0.5
        >>> calcVoltage(-1)
        0.0
        """
        # X points for interpolation
        xPoints=[0,0.05,0.1,0.2,0.4,0.6,0.8,1]
        # Y points for interpolation
        yPoints=[0,1,10,15,19,20,20,20]
        # Interpolate our input paramter using
        # the defined X and Y points.
        # Uses try except to catch any errors
        try:
            return np.interp(percentageCharge,xPoints,yPoints)
        except:
            print("Error invalid parameter")
            return 0

import doctest
doctest.testmod()
```

ii)

My approach to coding this part is to define a "Battery" class and define all the functions defined as methods within it

Using the same code from the previous part I am going to implement the calculate voltage part. I am going to use the formulas defined in the datasheet using simple python operators.

For the instantiation class, I am going to ensure everything is contained within the attributes so they can be shared across the entire class.

Try except clauses were used throughout for simple error checking to ensure the program does not crash and returns 0 with a printed error message.

```
import numpy as np

class Battery:
    def __init__(self, internalResistance, mass, SHC, currentBatTemp):
        self.internalResistance = internalResistance
        self.mass = mass
        self.SHC = SHC
        self.currentBatTemp = currentBatTemp


    def calcVoltage(self,percentageCharge):
        """ Returns the voltage given a charge percentage
            using interpolation
        :param percentageCharge: Value to be interpolated
        :type percentageCharge: real, int, float and other
        :return: A value of the voltage that was interpolated
        :rtype: float
        """
        # X points for interpolation
        xPoints=[0,0.05,0.1,0.2,0.4,0.6,0.8,1]
        # Y points for interpolation
        yPoints=[0,1,10,15,19,20,20,20]
        # Interpolate our input paramter using
        # the defined X and Y points.
        # Uses try except to catch any errors
        try:
            return np.interp(percentageCharge,xPoints,yPoints)
        except:
            print("Error invalid parameter")
            return 0

    def calcTempChange(self, supplyCurrent, airTemperature = 20):
        """ Returns the temperature change given the ambient
            temperature and supply current
        :param supplyCurrent: current of the supply used to
                    calculate temperature change
        :type supplyCurrent: real, int, float and other
        :param airTemperature: the environment temperature,
                (defualts to 20)
        :type airTemperature: real, int, float and other
        :return: the dT/dt rate of change of temperature
                against time
        :rtype: float
        """
        # Uses try except clause to check any errors.
        try:
            # The three lines below incorperate the function to calculate
            # the temperature change rate. They are written on multiple lines
            # to reduce clogging. I've achieved this using the += and /= operators
            dTdt = (supplyCurrent**2) * self.internalResistance
            dTdt += 40*(airTemperature - self.currentBatTemp)
            dTdt /= self.mass*self.SHC
            # Returns the temperature change rate.
            return dTdt
        except:
            print("Error in input")
            return 0
    def updateTemp(self, supplyCurrent, airTemperature = 20, timeStepSize = 0.01):
        """ Returns the temperature change given the ambient
                temperature and supply current
        :param supplyCurrent: current of the supply used to calculate
                        temperature change
        :type supplyCurrent: real, int, float and other
        :param airTemperature: the environment temperature, defualts to 20
        :type airTemperature: real, int, float and other
        :return: the dT/dt rate of change of temperature against time
        :rtype: float
        """
        # Uses try except clause to check any errors.
        try:
            #Extracts the dT/dt calling the calcTempChange method, and stores in dTdt
            dTdt = self.calcTempChange(supplyCurrent, airTemperature)
            # Uses the formula to calculate the next battery temperature
            # using itself and the times step multiplied by the rate of change
            # Tn+1 = Tn + ΔT · dT/dt
            self.currentBatTemp += timeStepSize*dTdt
        except:
            print("Error in input")
```

2)

My approach to this exercise consists of taking the file from a user input, and if the user enters nothing/something invalid it should default to file.txt

From here I will read every line of the file. Once we have every line, I will go through every single character of the line and start using selective statements to deduce if the character is a space or a vowel. I will use a variable to tally how many characters, vowels and spaces there are in total across all lines. Using that I will find the percentages and print out an output. I will ensure that the output of the percentage is rounded to one decimal place using the round() function. There is also validation to ensure there is no zero division error when calculating the percentage.

The question asked for percentage of vowels and spaces, so I interpreted this as:
Percentage = (vowels+spaces)/totalCharacters * 100

```
# Takes file name as an input to the user
inputFileName = input("Enter a file name : ")
# Validation to check if the file exists, if it doesn't
# It defaults to file.txt
try:
    textFile = open(inputFileName, "r")
except:
    print("Invalid file name, defaulting to file.txt")
```

```python
    textFile = open("file.txt","r")
# Reads all the lines in the text file and splits it into
# an array by line
fileLines=textFile.read().splitlines()
textFile.close()
# initialises all the variables.
vowelCount=0
charCount=0
spaceCount=0
# Goes through every line and every character
# to check if any of the characters are vowels or
# spaces and to count the total
for line in fileLines:
    for char in line:
        charCount+=1
        if char == " ":
            spaceCount+=1
        if char.lower() in ["a","e","i","o","u"]:
            vowelCount+=1
# Formats the output to print all the tallied values
print("Number of Spaces (' ') : ",spaceCount)
print("Number of Vowels (a, e, i, o or u') : ", vowelCount)
# Ensures there is no zero division error when calculating
# the percentage.
if charCount!=0:
    percentage=round(((vowelCount+spaceCount)/charCount)*100,1)
    print("Percentage of vowels and spaces : ",percentage)
```