Cardiff School of Computer Science and Informatics

Coursework Assessment Pro-forma

Module Code	:	CMT309
Module Title	:	Computational Data Science
Lecturer	:	Dr. Oktay Karakus, Dr. Luis Espinosa-Anke
Assessment Title	:	CMT309 Programming Exercises
Assessment Number	:	1
Date set	:	05-11-2021
Submission date and time	:	17-12-2021 at 9:30am
Return date	:	04-02-2022

This assignment is worth 30% of the total marks available for this module. If coursework is submitted late (and where there are no extenuating circumstances):

1.) If the assessment is submitted no later than 24 hours after the deadline, the mark for the assessment will be capped at the minimum pass mark;
2.) If the assessment is submitted more than 24 hours after the deadline, a mark of 0 will be given for the assessment.

Your submission must include the official Coursework Submission Cover sheet, which can be found here:

https://docs.cs.cf.ac.uk/downloads/coursework/Coversheet.pdf

Submission Instructions

Your coursework should be submitted via Learning Central by the above deadline. You have to upload the following files:

Description		Type	Name
Cover sheet	Compulsory	One PDF (.pdf) file	Student_number.pdf
Your solution to question 1	Compulsory	One Python (.py) file	Q1.py
Your solution to question 2	Compulsory	One Python (.py) file	Q2.py
Your solution to question 3	Compulsory	One Python (.py) file	Q3.py

For the filename of the Cover Sheet, replace 'Student_number' by your student number, e.g. "C1234567890.pdf". Make sure to include your student number as a comment in all of the Python files! Any deviation from the submission instructions (including the number and types of files submitted) may result in a reduction of marks for the assessment or question part.

You can submit multiple times on Learning Central. ONLY files contained in the last attempt will be marked, so make sure that you upload all files in the last attempt.

Staff reserve the right to invite students to a meeting to discuss the Coursework submissions.

Testing Your Codes

You are given with three Python codes, named testQ1.py, testQ2.py and testQ3.py. These codes will give you the chance to test your implementations of the questions. Each code runs your implementations for a number of testcases. You use the test codes to make sure that:

- Your function does not crash, that is, there is no Python errors when trying to run the function.
- Compare the results of the testcases to your results. Expected results are given at the end of each code.

For Q1 and Q3, returning the same outputs in the test codes does not assure that you will get full marks. We will use additional testcases (not disclosed) to test your functions. For Q2, all the testcases are shown in the question. How to use the test codes:

- Create your functions in any environment.
- In each code, replace `pass` command with your implementation of each required function.
- Execute the updated test file. It will run your implementations for a number of testcases specified in the questions. If any errors occur, you need to correct your code.

IMPORTANT: You must make sure that your file executes and does not crash before submitting to Learning Central. Any function that crashes or does not execute will receive 0 marks on the respective (sub)question. Note that the test codes are only provided for your convenience.

Assignment

Start by downloading the following files from Learning Central:

- Q1.py
- Q2.py
- Proper_nouns.txt
- Q3.py

Then answer the following questions. You can use any Python expression or package that was used in the lectures and practical sessions. Additional packages are not allowed unless instructed in the question.

Question 1 - Arithmetic (Total 40 marks)

In this question, your task is to implement two functions that perform arithmetic operations. These are (Q1.a) `do_arithmetic()` and (Q1.b) `sum_of_digits()`.

Q1.a) Simple arithmetic (20 marks)

Write a function `do_arithmetic(x,y,op)` which takes three input arguments: a number `x`, a number `y`, and a string `op` representing an operation. The function has to perform the operation on the two numbers and return the result.

Example: if you call the function with the parameter `do_arithmetic(10,4,'add')` the function should return the value `14.0`.

Example 2: `do_arithmetic(2,3,'*')` should return `6.0`.

Detailed instructions:

- You can assume that `x` and `y` are always provided and always valid numbers (e.g. integers or floats).
- `op` is a string representing the operation to perform. You have to implement four operations with `'add'`, `'+'`, `'subtract'`, `'-'`, `'multiply'`, `'*'`, and `'divide'`, `'/'`.
- If `op` is not specified by the user it should default to `'add'`.
- If `op` is specified by the user but it is not one of the four operations (with eight keywords), print `'Unknown operation'` and return `None`.
- Division by zero should be avoided. To this end, return `None` whenever division by zero would occur, and print `'Division by 0!'`
- The returned result should always be of type float.

Test code: If you run testQ1.py after updating `do_arithmetic()` as instructed, the following test cases will be tested:

```
do_arithmetic(24, -7, 'add')

do_arithmetic(6, 6, 'multiply')

do_arithmetic(4, 0, '/')

do_arithmetic(3, 9, '-')
```

Q2.b) Sum of digits (20 marks)

Write a function `sum_of_digits(s)` that takes as input a string `s` that contains some numbers. The function calculates the sum of all the digits in the string, ignoring any symbols that are not digits.

Example: `sum_of_digits("123")` should return `6` since `1+2+3 = 6`.

Example 2: `sum_of_digits("10a20")` should return `3` because `1+0+2+0 = 3`.

Detailed instructions:

- if `s` includes both digits and nondigits
  - Calculate the sum of digits and return the result whilst ignoring any non-digit symbols in the string.
  - print e.g. for `sum_of_digits("10a20")`
    `'The sum of digits operation performs 1+0+2+0'`
  - Save the extracted non-digits in a variable of interest as a `list` and print
    `"The extracted non-digits are: ['a']"`
- If `s` is not provided or an empty string return `0` and print
  `'Empty string entered!'`.
- If `s` is provided, but it contains no digits return `0` and print
  `'The sum of digits operation could not detect a digit!'`
  `'The returned input letters are: [ALL NONDIGITS HERE]'`
- The returned number should be an integer.

Example 1: When you run `sum_of_digits("a1w3")`, the function should print

```
The sum of digits operation performs 1+3
The extracted non-digits are:  ['a', 'w']
4
```

Example 2: When you run `sum_of_digits("united")`, the function should print

```
The sum of digits operation could not detect a digit!
The returned input letters are:  ['u', 'n', 'i', 't', 'e', 'd']
0
```

Test code: If you run testQ1.py after updating `sum_of_digits()` as instructed, the following test cases will be tested:

```
sum_of_digits("123")

sum_of_digits("we10a20b")

sum_of_digits("united")

sum_of_digits("")
```

As a starting point, you can use testQ1.py from Learning Central.

Question 2 - Write a pluralize function (Total 30 marks)

Write a function `pluralize(word)` that determines the plural of an English word.

Input: Your function takes one argument as input, `word`, a string representing the word to be pluralised. We assume this word will only be one single token, i.e., no spaces. Your function will also have to load the text file *proper_nouns.txt*, from the same directory in which your script Q2.py is located. The file is provided in Learning Central.

Output: The function should return a dictionary of the form:

- `{'plural': word_in_plural, 'status': x}`

where `word_in_plural` is the pluralized version of the input argument `word` and `x` is a string which can have one of the following values: `'empty_string'`, `'proper_noun'`, `'already_in_plural'`, `'success'`.

Below is the logic (**in order**) that the function should execute.

1) Determine if the word is an empty string, already in plural, or a proper noun.

- If the word is an empty string, then your function returns a dictionary with the following values:
  - `word_in_plural = ''` and `x = 'empty_string'`
  - **Explanation:** The input word is an empty string and it cannot be pluralized.
- If the word is already in plural, then your function returns a dictionary with the following values:
  - `word_in_plural = word` and `x = 'already_in_plural'`.
  - **Explanation:** The input word remains untouched (e.g., input: houses, output: houses).
- If the word is a proper noun, then your function returns a dictionary with the following values:
  - `word_in_plural = word` and `x = 'proper_noun'`.
  - **Explanation:** The input word is a proper noun, and therefore cannot be pluralized (e.g., input: Adam, output: Adam).
- **How to determine plural form:** We will assume a word is in plural if it ends with 's'. We will make the strong assumption that there are no singular words in English ending with 's'.
- **How to determine if a word is a proper noun:** We will assume a word is a proper noun if it exists in the file *proper_nouns.txt*. Note that your function should convert any capitalised input to lower case first because the proper nouns in the list are all lower case. The values in your output dictionary, however, will retain the original capitalisation.

2) If the word is not plural and is not a proper noun, then:

Apply the following English plural rules.

- If the word ends with a vowel:
  - add -s.
- Otherwise:
  - If it ends with 'y' and is preceded by a consonant, erase the last letter and add -ies.
  - If it ends with 'f', erase the last letter and add -ves.
  - If it ends with 'sh'/'ch'/'z', add -es.
  - If none of the above applies, just add -s.

- After these rules are applied, your output dictionary should be:
  - `{'plural': word_in_plural, 'status': 'success'}`

You can assume that *proper_nouns.txt* is always provided and will be located in the same directory as the script Q2.py.

Test code: If you run testQ2.py after updating `pluralize()` as instructed, the following testcases and no others will be tested (a few sample outputs with >>> added for extra clarity):

```
pluralize("failure")
>>> {'plural':'failures', 'status':'success'}
pluralize("food")
pluralize("Zulma")
pluralize("injury")
pluralize("elf")
pluralize("buzz")
pluralize("computers")
pluralize("PCs")
>>> {'plural':'PCs', 'status':'already_in_plural'
pluralize("")
>>> {'plural':'', 'status':'empty_string'}
pluralize("highway")
pluralize("presentation")
pluralize("pouch")
pluralize("COVID-19")
pluralize("adam")
```

As a starting point, you can use testQ2.py from Learning Central.

Question 3 - Function renamer (Total 30 Marks)

You are working on a large-scale software project involving thousands of different Python files. The files have been written by different programmers and the naming of functions is somewhat inconsistent. You receive a new directive stating that the function names all have to be in camel case. In camel case, function names consisting of multiple words have a capital letter in each word and most underscores are removed. For instance, `def MyArithmeticCalculator` is in camel case but `def my_arithmetic_calculator` is not in camel case.

You want to write a Python program that automatically processes Python code and renames the function names instead of doing it by hand. The instruction states that:

1.) All functions names need to be changed to camel case. E.g. a function `calculate_speed_of_vehicle` needs to be renamed to `CalculateSpeedOfVehicle`.
2.) If the function has one or more leading '_' (underscores) they need to be preserved. All other underscores need to be removed. E.g. a function `__calc_size` is renamed to `__CalcSize`.
3.) If the function is already in camel case, you do not need to change it but it still needs

to appear in the dictionary `d` specified below.

4.) You can assume that there will be no name clashes. That is, a given function name which is not in camel case will not already appear in camel case elsewhere. E.g. if there is a function `print_all_strings` then there is no function `PrintAllStrings` elsewhere in the code.

5.) **Tip:** You can use regular expressions to find the function names.

To implement this, write a function `function_renamer(code)` that takes as input a string code that represents the Python code. It is typically a multi-line Python string. Your function needs to return the tuple `(d, newcode)`:

- `d` is a nested dictionary where each key corresponds to the original function name. The value is a nested dictionary that has the following items:
  - `hash`: hash code of the original function name (tip: use Python's hash function)
  - `camelcase`: camel case version of original function name
  - `allcaps`: all caps version of original function name
- `newcode` is a string containing the code wherein all function names have been renamed by their camel case versions. Note that you need to change the function name and also all other locations where the function name is used (e.g. function calls). You should not change anything else in the code (e.g. the contents of any strings).

To clarify this, a few examples are given below:

Example 1: Assume your input code is the multi-line string

```
def add_two_numbers(a, b):
    return a + b
print(add_two_numbers(10, 20))
```

After processing it with your function, the code should be changed to

```
def AddTwoNumbers(a, b):
    return a + b
print(AddTwoNumbers(10, 20))
```

The nested dictionary is given by

```
d = {'add_two_numbers':
    {'hash':-9214996652071026704,
    'camelcase':'AddTwoNumbers',
    'allcaps':'ADD_TWO_NUMBERS'} }
```

Example 2: Assume your input code is the multi-line string

```
def _major_split(*args):
    return (args[:2], args[2:])
def CheckTruth(t = True):
    print('t is', t)
    return _major_split([t]*10)
x, y = _major_split((10, 20, 30, 40, 50))
```

```
CheckTruth(len(x) == 10)
```

After processing it with your function, the code should be changed to

```
def _MajorSplit(*args):
     return (args[:2], args[2:])
def CheckTruth(t = True):
     print('t is', t)
     return _MajorSplit([t]*10)
x, y = _MajorSplit((10, 20, 30, 40, 50))
CheckTruth(len(x) == 10)
```

The nested dictionary is given by

```
d = {'CheckTruth':
     {'hash':-6410081306665365595,
     'camelcase':'CheckTruth',
     'allcaps':'CHECKTRUTH',
     '_major_split':
     {'hash':484498917506710667,
     'camelcase':'_MajorSplit',
     'allcaps':'_MAJOR_SPLIT'}}
```

Test code: If you run testQ3.py after updating `function_renamer()` as instructed, the following test cases will be tested: Example 1 (see above) and Example 2 (see above).

As a starting point, you can use testQ3.py from Learning Central.

Learning Outcomes Assessed

- Use the Python programming language to complete a range of programming task
- Demonstrate familiarity with programming concepts and data structures
- Use code to extract, store and analyse textual and numeric data

Criteria for assessment

Credit will be awarded against the following criteria. The score in each implemented function is judged by its functionality. For all questions, the functions you have implemented will be tested against different test cases to judge their functionality. Additionally, quality will be assessed. Please use the test codes to double check your question after you change it and before you submit it in Learning Central: Any function that crashes or does not execute will receive 0 marks on the respective (sub)question. The below table explains the criteria.

	Mark	**Functionality (80%)**	**Quality (20%)**
Q1, Q2, Q3	Distinction (70-100%)	Fully working application that demonstrates an excellent understanding of the assignment problem using relevant python approach	Excellent documentation with usage of __docstring__ and comments
	Merit (60-69%)	All required functionality is met, and the application are working probably with some minors' errors	Good documentation with minor missing of comments
	Pass (50-59%)	Some of the functionality developed with and incorrect output major errors	Fair documentation
	Fail (0-50%)	Faulty application with wrong implementation and wrong output	No comments or documentation at all

Feedback and suggestion for future learning

Feedback on your coursework will address the above criteria. Feedback and marks will be returned within 4 weeks of your submission date via Learning Central. In case you require further details, you are welcome to schedule a one-to-one meeting. Feedback from this assignment will be useful for next year's version of this module as well as the Data Science Foundations and Statistical Programming modules.