IS111 – Lab 5
Lists

Instructions

You should attempt the questions using Visual Studio Code. Download and extract the files from the **Lab5_starting_code.zip**. You should obtain several *.py files to be used for some of the lab questions.

Please use the test cases provided in the starting code of Question 2 to 6 to test your respective solutions.

Q1: Initials [**]

Write a program that prompt for the names of people attending a meeting. After that, print out the initials of these people.

You can assume that each participant's name consists of a sequence of words separated by a single space. You can assume that each person's name contains at least one word. The initial of a person contains the first letters of all the words in that person's name.

A sample run of the program looks like the following:

```
How many people will attend the meeting? 5
Participant 1: John Smith
Participant 2: Jerry Lee Xiong Yi
Participant 3: Eric Wong Kee Wei
Participant 4: Felicia Koh
Participant 5: Julia Chan


The initials of the participants are as follows:
JS
JLXY
EWKW
FK
JC
```

Q2: List of Numbers [**]

Define the following functions that handle a list of numbers:

a) In the given file `q2a.py`, define a function called `get_leap_years()`. This function takes in a list of numbers that indicate years. It returns a list that contains only those years that are leap years. For definition of leap years, see the following link:

https://en.wikipedia.org/wiki/Leap_year#Algorithm

For example, `get_leap_years([2018, 2000, 1800, 1900, 2011, 2020])` returns the list `[2000, 2020]`.

b) Define a function called `all_older_than()`. The function takes two parameters: (1) A list of integers called `age_list`, where each element indicates the age of a person. (2) An integer called `n`, which is a threshold. The function returns `True` if ALL the age values in `age_list` are larger than `n`, and `False` otherwise.

For example, `all_older_than([24, 36, 45, 21], 20)` returns `True`, and `all_older_than([24, 36, 45, 21], 23)` returns `False`.

If `age_list` is empty, the function returns `True`.

c) Define a function called `get_sum_of_multiples()`. The function takes in two parameters: (1) A list of integers called `int_list`. (2) An integer `n`. The function returns an integer, which is the sum of all the integers in `int_list` that are multiples of `n`, i.e., that are divisible by `n`. You can assume that `n` is always a positive integer.

For example, `get_sum_of_multiples([2, 4, 5, 9, 13, 15], 3)` returns 24 (sum of 9 and 15), and `get_sum_of_multiples([2, 4, 5, 9, 13, 15], 5)` returns 20 (sum of 5 and 15).

d) Define a function called `get_prime_numbers()`. The function takes in two parameters: (1) A list of integers called `num_list`. (2) A string `sep` that serves as a separator. The function returns a **string** that contains the prime numbers inside `num_list`, separated by `sep`.
For example, `get_prime_numbers([2, 4, 7, 9, 11, 16, 19, 21], '-')` returns the string `"2-7-11-19"`.

See the following link for the definition of prime numbers:

https://en.wikipedia.org/wiki/Prime_number

Note: You should write a function to help you check whether a number is a prime number.

e) Define a function called `calculate_sums()`. The function takes in a list of numbers, call `num_list`. It returns a new list of numbers that has the same length of `num_list`. The `n`'th element of the returned list is the sum of the first `n` numbers in `num_list`.

For example, `calculate_sums([2, 3, 6, 1, 5])` returns the list `[2, 5, 11, 12, 17]`. (Here 5 is the sum of 2 and 3; 11 is the sum of 2, 3 and 6; 12 is the sum of 2, 3, 6 and 1; and 17 is the sum of 2, 3, 6, 1 and 5.)

If the list `num_list` is empty then return an empty list.

SMU SINGAPORE MANAGEMENT UNIVERSITY

IS111 – Lab 5
Lists

Q3: Shopping Cart [**]

You will be implementing a few functions dealing with an `item_list`. Each element of `item_list` is a tuple with three values: the name of an item, its unit price, and the quantity of the item in the shopping cart.

For example, `item_list` may look like the following:

```
[("milk", 5.45, 2), ("eggs", 2.45, 1), ("shampoo", 8.90, 2)]
```

a) Define a function called `calculate_total_price()` that takes a parameter called `item_list`, as described above. The function returns the total price (unit price multiplied by quantity) of all the items in the shopping cart.

b) Define a function called `get_items()` that takes a parameter called `item_list`, as described above. The function returns a list of strings, which are the names of the items in `item_list`. For example, `get_items([("milk", 5.45, 2), ("eggs", 2.45, 1), ("shampoo", 8.90, 2)])` returns `["milk", "eggs", "shampoo"]`.

c) Define a function called `get_items_more_expensive_than()`. The function takes in two parameters: (1) `item_list`. (2) A float value called `min_price`. The function returns a list of tuples that represents those items in `item_list` whose unit price is above `min_price`. Each tuple in the returned list contains the name of an item and its unit price. For example, `get_items_more_expensive_than([("milk", 5.45, 2), ("eggs", 2.45, 1), ("shampoo", 8.90, 2)], 3.0)` returns `[("milk", 5.45, 2), ("shampoo", 8.90, 2)]`

Q4: Spelling Check [**]

You are given a file called `q4.py`. Inside the file, you're given a list called COMMON_WORDS that contains 5000 commonly used English words. Define a function called `check_spelling()` that checks for misspellings. The function takes in a string that represents a piece of text. It returns a list of words from the text that are possibly misspelled, i.e., a list of words that are not found in the 5000 commonly used English words.

For example, `check_spelling("I studdy at Singapore Managment Univercity")` should return `["studdy", "Managment", "Univercity"]`.

Note: Words in the COMMON_WORDS list are in lower cases.

Q5: Tax Calculation [***]

Let us now revisit the tax calculation task. Recall that we have the following tax rates:

Chargeable Income	Income Tax Rate (%)	Gross Tax Payable ($)

First $20,000 Next $10,000	0 2	0 200
First $30,000 Next $10,000	- 3.50	200 350
First $40,000 Next $40,000	- 7	550 2,800
First $80,000 Next $40,000	- 11.5	3,350 4,600
First $120,000 Next $40,000	- 15	7,950 6,000
First $160,000 Next $40,000	- 18	13,950 7,200
First $200,000 Next $40,000	- 19	21,150 7,600
First $240,000 Next $40,000	- 19.5	28,750 7,800
First $280,000 Next $40,000	- 20	36,550 8,000
First $320,000 In excess of $320,000	- 22	44,550

The information above can be stored inside a list of tuples as shown below:

```
TAX_INFO = [
    (20000, 0, 0.02),
    (30000, 200, 0.035),
    (40000, 550, 0.07),
    (80000, 3350, 0.115),
    (120000, 7950, 0.15),
    (160000, 13950, 0.18),
    (200000, 21150, 0.19),
    (240000, 28750, 0.195),
    (280000, 36550, 0.2),
    (320000, 44550, 0.22)
]
```

You can see that each element of this list is a tuple with three values: (1) an amount of chargeable income, (2) the payable tax for that amount, and (3) the tax rate for extra income above that amount. For example, the tuple `(30000, 200, 0.035)` indicates that for the first $30,000 of chargeable income, $200 is charged as tax, and for any additional income above $30,000 (and below the next threshold of $40,000), a tax rate of 3.5% is applied.

The list above is given in `q5.py`. Implement a function called `calculate_tax()` inside `q5.py` that takes in a number representing the taxable income of a person. The function returns the amount of tax that person has to pay. Some test cases have been given in `q5.py`.

Q6: More on Lists [***]

In all the questions below, you can assume that the lists passed to the functions (i.e., the parameters) do not contain any duplicate elements. You can also assume that the lists passed to the functions are not empty.

a) Define a function called `get_all_combinations()`. The function takes in two lists. The first list is called `str_list` and contains a sequence of strings. The second list is called `num_list` and contains a sequence of numbers. The two lists may have different lengths. The function returns a list of tuples, where each tuple is a combination of an element from `str_list` and an element from `num_list`. The returned list should contain all possible combinations.

   For example, `get_all_combinations(["a", "b"], [1, 2, 3])` should return `[("a", 1), ("a", 2), ("a", 3), ("b", 1), ("b", 2), ("b", 3)]`.

b) Define a function called `get_larger_numbers()`. The function takes in two lists of numbers, `num_list1` and `num_list2`. The function returns all the numbers in `num_list1` that are larger than all the numbers in `num_list2`.

   For example, if `num_list1` is `[4, 6, 10]` and `num_list2` is `[1, 3, 5]`, then the function should return `[6, 10]`. This is because 4 is not larger than all the numbers in `num_list2`, but 6 and 10 are both larger than all the numbers in `num_list2`.

c) Define a function called `get_non_common_strings()`. The function takes in two lists of strings, `str_list1` and `str_list2`. The function returns a list of strings that can be found in either `str_list1` or `str_list2`, BUT NOT in both.

   For example, if `str_list1` is `["a", "b", "c", "d"]`, `str_list2` is `["b", "d", "e", "f"]`, then this function returns `["a", "c", "e", "f"]`.