NYU Tandon School of Engineering

CS-UY 1114 Fall 2022

Homework 08

Due: 11:59pm, Thursday, November 17th, 2022

Submission instructions

1. You should submit your homework on **Gradescope**.
2. For this assignment you should turn in 2 separate `.py` files named according to the following pattern: `hw8_q1.py` and `hw8_q2.py`.
3. Each Python file you submit should contain a header comment block as follows:

```
"""
Author: [Your name here]
Assignment / Part: HW8 - Q1 (depending on the file name)
Date due: 2022-11-17, 11:59pm
I pledge that I have completed this assignment without
collaborating with anyone else, in conformance with the
NYU School of Engineering Policies and Procedures on
Academic Misconduct.
"""
```

No late submissions will be accepted.

REMINDER: *Do not use any Python structures that we have not learned in class.*

The use of `eval()` and `break` are no longer permitted in this class.

For this specific assignment, you may use everything we have learned up to, **and including**, dictionaries. Please reach out to us if you're at all unsure about any instruction or whether a Python structure is or is not allowed.

Do **not** use, for example, file i/o, exception handling, and/or object-oriented programming.

Problems

1. **(Probably) The Most Famous Computer Science Problem Ever (`hw8_q1.py`)**
2. **Dictionaries, I Choose You! (`hw8_q2.py`)**
3. **Gotta Archive 'Em All! (`hw8_q2.py`)**

Problem 1: *(Probably) The Most Famous Computer Science Problem Ever*

One of the canonical problems in computer science is the character/word counter. As the name implies, it is a program that scans a body of characters, and returns a data structure that tells you how many times each character/word appears in a given corpus. We will be implementing a simple version of this program using nucleotides.

Write a function, **`get_nucleotide_frequencies()`**, that accepts `nucleotides` as a parameter (i.e. a string of characters), and returns both of the following values:

- A dictionary containing the frequency of each of the four nucleotides: adenine, represented by the character `'A'`, guanine (`'G'`), thymine (`'T'`), and cytosine (`'C'`).
    - The keys are the nucleotide characters, and the values will keep track of the frequencies.

    - This dictionary must also keep a `"Junk"` key, with its corresponding value being a dictionary of junk nucleotides (see sample behaviour below).
- The amount of junk nucleotides encountered in this string.

That is, if the following script were to be executed

```
from pprint import pprint  # all this function does is print dictionaries nicely.
                           # You don't have to use it

def main():
    frequencies, junk_count =
get_nucleotide_frequencies(sequence="ACTGTCaCGRFRTNfsHYCgggTCGACCSGTGTCACGT")
    pprint(frequencies)
    print(f"Number of junk nucleotides: {junk_count}.")

main()
```

Output:

```
{'A': 4,
 'C': 9,
 'G': 9,
 'Junk': {'F': 2, 'H': 1, 'N': 1, 'R': 2, 'S': 2, 'Y': 1},
 'T': 7}
Number of junk nucleotides: 9.
```

Problem 2: *Dictionaries, I Choose You!*

Note: Problems 2 and 3 should be included in the same file, `hw8_q2.py`.

> *"As a wise man once said, "Trust in yourself, and you will find your way." These words are worth taking to heart! As long as we truly believe in ourselves, I'm sure we'll even complete that Pokédex of ours someday!"*

— Professor Laventon

Back at it again with the *Pokémon* problems. Problems 2 and 3 will be related to building a PokéDex which, in case you have never seen the show or played the games, it is basically a dictionary (conveniently) where you can look up the information of any Pokémon. Let's first focus on creating a single entry. Write a function **`create_entry()`**, that will accept the following parameters:

Parameter Name	Type	Notes
`number`	`int`	Commonly abbreviated as simply `#`, denotes the ID number of this Pokémon.
`name`	`str`	
`type_1`	`str`	
`type_2`	`str`	While all Pokémon have a `type_1`, not all Pokémon have a `type_2`. Be careful!
`health_points`	`int`	Commonly abbreviated as `HP`.
`attack`	`int`	
`defense`	`int`	
`speed`	`int`	
`is_legendary`	`bool`	There are special, usually very powerful, Pokémon called "Legendary Pokemon."

Figure 1: Parameters of function `create_entry()`.

When these arguments are passed as parameters, a `create_entry()` function call will create and return a dictionary.

```
def main():
  a_random_pokemon = create_entry(81, "Magnemite", "Electric", "Steel", 25, 35, 70, 95,
False)

  for key in a_random_pokemon.keys():
      print(f"{key}: {a_random_pokemon[key]}")


main()
```

Possible output for the program (since dictionary key-value pairs are unordered, the key-value pairs could be printed out in any order and still be correct):

```
Number: 81
Name: Magnemite
Types: ('Electric', 'Steel')
Battle Stats: {'HP': 25, 'Attack': 35, 'Defense': 70, 'Speed': 45}
Legendary: False
```

A few of things to note when working on this problem:

- Notice that not all Pokémon will have a second type. If this is the case, you may assume `type_2` will be passed into `create_entry()` as an **empty string** (`""`).
- Instead of having an individual key for both types, create a tuple containing both types, in order. If `type_2` is an empty string, the second value in the tuple should be `None` (the keyword, not a string).
- Also, notice that instead of creating an individual entry with each Pokémon's battle stat as a key, the key `Battle Stats` maps to another dictionary containing them. The example output below demonstrates this dictionary structure:

```
print(a_random_pokemon["Battle Stats"])
print(a_random_pokemon["Battle Stats"]["HP"])
print(a_random_pokemon["Battle Stats"]["Attack"])
```

Output:

```
{'HP': 25, 'Attack': 35, 'Defense': 70, 'Speed': 45}
25
35
```

Writing a separate function to create this battle stats dictionary might be useful, but that's up to you.

Problem 3: *Gotta Archive 'Em All!*

In data science, data is often recorded in **CSV (comma-separated values) files**. That is, a file...

> *...that uses a **comma to separate values**. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format. A CSV file typically stores tabular data (numbers and text) in plain text, in which case each line will have the same number of fields.*

For example, if there were a CSV file containing Pokémon data, its first 5 lines might look like this:

```
#,Name,Type 1,Type 2,Total,HP,Attack,Defense,Sp. Atk,Sp. Def,Speed,Generation,Legendary
1,Bulbasaur,Grass,Poison,318,45,49,49,65,65,45,1,False
2,Ivysaur,Grass,Poison,405,60,62,63,80,80,60,1,False
3,Venusaur,Grass,Poison,525,80,82,83,100,100,80,1,False
3,VenusaurMega Venusaur,Grass,Poison,625,80,100,123,122,120,80,1,False
4,Charmander,Fire,,309,39,52,43,60,50,65,1,False
```

#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	False
2	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	1	False
3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	1	False
3	VenusaurMega Venusaur	Grass	Poison	625	80	100	123	122	120	80	1	False
4	Charmander	Fire		309	39	52	43	60	50	65	1	False

Figures 2 & 3: A Pokémon CSV data file and its tabular representation. First 6 lines. Note `Venusaur` and `VenusaurMega Venusaur` have the same number ID. You can consider them to be distinct and individual Pokémon.

We'll deal with reading data from actual CSV files next week, but for now, pretend that this data exists instead in a really (really) long, multi-line string:

```
POKEMON_DATA = """#,Name,Type 1,Type 2,Total,HP,Attack,Defense,Sp. Atk,Sp.
Def,Speed,Generation,Legendary
1,Bulbasaur,Grass,Poison,318,45,49,49,65,65,45,1,False
2,Ivysaur,Grass,Poison,405,60,62,63,80,80,60,1,False
3,Venusaur,Grass,Poison,525,80,82,83,100,100,80,1,False
3,VenusaurMega Venusaur,Grass,Poison,625,80,100,123,122,120,80,1,False
4,Charmander,Fire,,309,39,52,43,60,50,65,1,False"""
```

Code Block 1: A multi-line Python string containing the data from figures 2 and 3. This is only a sample string; your function must work with a string containing any number of Pokémon data.

Write a function, `create_pokedex()`, that will accept one parameter, `pokemon_data` which, for now, will be a multi-line string of the same format (but not necessarily of the same contents) as `POKEMON_DATA` above.

Your program must then:

1. Create a Pokémon entry from every Pokémon entry inside the multi-line string (using the function `create_entry()` we created in problem 2). Once you create an entry, add it to the PokéDex dictionary (that is, just a regular Python dictionary that is initially empty), using the Pokémon's **name as the key**, and the **whole entry as the value**.
2. Once you have gone through every Pokémon in the string, return the PokéDex dictionary.

When `create_pokedex()` is implemented, your program will exhibit the following behavior:

```
def main():
    pokedex = create_pokedex(POKEMON_DATA)
    pokemon_key = "Ivysaur"

    if pokemon_key not in pokedex:
        print(f"ERROR: Pokemon {pokemon_key} does not exist!")
    else:
```

```
        print(f"PRINTING {pokemon_key}'S INFORMATION...")

        for key in my_favorite_pokemon.keys():
            print(f"{key}: {my_favorite_pokemon[key]}")


main()
```

Output:

```
PRINTING Ivysaur'S INFORMATION...
Number: 2
Name: Ivysaur
Types: ('Grass', 'Poison')
Battle Stats: {'HP': 60, 'Attack': 62, 'Defense': 63, 'Speed': 60}
Legendary: False
```

If we changed the value of the variable `pokemon_key` to, say, `"Mikami"` (i.e. not a Pokémon), we'd instead see the following message printed onto our console:

```
ERROR: Pokemon Mikami does not exist!
```