Python Assignment-1
CS1023

1. Start with a number $n > 1$. Find the number of steps it takes to reach one using the following process: If n is even, divide it by 2. If n is odd, multiply it by 3 and add 1.

   Write a recursive function (a function that calls itself) to solve this problem.

2. Reverse Lines:

   Write a function ***reverse_line*** which takes one argument: the name of the input file (to be read from). The output file will be created by the program. The newline remains at the end of the string, while the rest of the characters are all reversed.

   Sample input: `abc def`

   Sample output: `fed cba`

   Create a text file whose contents are given in the sample input (give the input filename as *`reverse_lines.txt, this file is already created for you, see the repo.`*). The program should create an empty text file whose contents will be written by the program as per the *reverse_line* logic. Give the output filename as *`reverse_lines_op.txt`*.

3. Word count:

   Unix systems contain many utility functions. One of the most useful is `wc`, the word count program. If you run `wc` against a text file, it'll count the characters, words, and lines that the file contains. The challenge for this exercise is to write a ***wordcount*** function that mimics the `wc` Unix command. The function will take a filename as input and will print four lines of output:

   1. Number of characters (including whitespace)

   2. Number of words (separated by whitespace)

   3. Number of lines

   4. Number of unique words (case sensitive, so `"NO"` is different from `"no"`)

Input file content:

Save this text in a file named *wcfile.txt*. (There's an empty line between each line, this file is already created for you).

This is a test file.

It contains 28 words and 20 different words.

It also contains 165 characters.

It also contains 11 lines.

It is also self-referential.

Wow!

4. Write a function, ***dictdiff***, that takes two dicts as arguments. The function returns a new dict that expresses the difference between the two dicts. If there are no differences between the dicts, *dictdiff* returns an empty dict. For each key-value pair that differs, the return value of *dictdiff* will have a key-value pair in which the value is a list containing the values from the two different dicts. If one of the dicts doesn't contain that key, it should contain `None`.

   Sample Inputs:

   Case I:

   d1 = {'a':1, 'b':2, 'c':3}

   d2 = {'a':1, 'b':2, 'c':4}

   Case II:

   d3 = {'a':1, 'b':2, 'd':3}

   d4 = {'a':1, 'b':2, 'c':4}

Case III:

d5 = {'a':1, 'b':2, 'd':4}

Sample Outputs:

Case I:

dictdiff(d1,d1)

dictdiff(d1,d2)

{}

{'c': [3, 4]}

Case II:

dictdiff(d3,d4)

{'c': [None, 4], 'd': [3, None]}

Case III:

dictdiff(d1,d5)

{'c': [3, None], 'd': [None, 4]}

5. Alphabetize Names:

   Let's assume you have phone book data in a list of dicts, as follows:

```
PEOPLE = [{'first':'Narendra', 'last':'Modi',
'email':'naren@modi.co.in'}, {'first':'Donald',
'last':'Trump','email':'president@whitehouse.gov'},
{'first':'Vladimir', 'last':'Putin',
'email':'president@kremvax.ru'}]
```

   Write a function, ***alphabetize_names***, that assumes the existence of a **PEOPLE** constant defined as shown in the code above. The function should return the *list of dicts*, but sorted by last name and then by first name.

6. Word with Most Repeating Letters:

   Write a function, ***most_repeating_word***, that takes a sequence of strings as input. The

function should return *the string that contains the greatest number of repeated letters*. In other words, for each word, find the letter that appears the most times. Find the word whose most-repeated letter appears more than any other. That is, if words are set to `words = ['this', 'is', 'an', 'elementary', 'test', 'example']` then your function should return `elementary`.

7. Write a Python function, ***format_sort_records***, that takes the **PEOPLE** list and returns a formatted string that looks like the following:

```
PEOPLE = [('Donald', 'Trump', 7.85), ('Vladimir', 'Putin',
3.626), ('Narendra', 'Modi', 10.603)]
```

Output:

```
Trump Donald 7.85

Putin Vladimir 3.63

Modi Narendra 10.60
```

8. Write a function, ***passwd_to_dict***, that reads from a Unix-style "`password file`", commonly stored as */etc/passwd*, and returns a dict based on it in which the dict's keys are usernames and the values are the users' IDs.

   Create a text file and save it with the name `passwd.txt (the file is already created for you)`. The contents of the file are as follows:

```
nobody:*:-2:-2::0:0:Unprivileged User:/var/empty:/usr/bin/false

root:*:0:0::0:0:System Administrator:/var/root:/bin/sh

daemon:*:1:1::0:0:System Services:/var/root:/usr/bin/false
```

   Each line is one user record, divided into colon-separated fields. The first field (index `0`) is the username, and the third field (index `2`) is the user's unique ID number. (In the system from which I took the */etc/passwd* file, `nobody` has ID `-2`, `root` has ID `0`, and `daemon` has ID `1`.) For simplicity, you can ignore all but these two fields.

9. When you were little, you might have created or used a "secret" code in which "`a`" was 1, "`b`" was 2, "`c`" was 3, and so forth, until "`z`" (which was 26). This type of code happens to be quite ancient and was used by a number of different groups more than 2,000 years ago. "***Gematria***," as it is known in Hebrew, is the way in which biblical

verses have long been numbered. You create a dict whose keys are the (lowercase) letters of the English alphabet, and whose values are the numbers ranging from 1 to 26. You are expected to create this list using *dictionary comprehension*.

Now consider this constant dictionary:

```
DICT_WORDS = ["cat", "bat", "act", "tap", "pat", "rat", "tar",
"art", "zap", "ax", "fox", "box"]
```

Your task is to write two functions:

a. ***gematria_for***, which takes a single word (string) as an argument and returns the gematria score for that word.
b. ***gematria_equal_words***, which takes a single word and returns a list of those words in the `DICT_WORDS` whose gematria scores match the current word's score.

For example, if the function is called with the word `cat`, with a gematria value of 24 (3 + 1 + 20), then the function will return a list of strings (unique), all of whose gematria values are also 24. Now also return the words in `DICT_WORDS` having the same gematria value as 24.

Any nonlowercase characters in the user's input should count 0 toward your final score for the word.

10. Write a *password-generation* function. We might need to generate a large number of passwords, all of which use the same set of characters. (Some applications require a mix of capital letters, lowercase letters, numbers, and symbols; whereas others require that you only use letters; and still others allow both letters and digits.) You'll thus call the function ***create_password _generator*** with a string. That string will return a function, which itself takes an integer argument. Calling this function will return a password of the specified length, using the string from which it was created; for example:

```
alpha_password = create_password_generator('abcdef')

symbol_password = create_password_generator('!@#$%')

print(alpha_password(5)) # efeaa

print(alpha_password(10)) # cacdacbada

print(symbol_password(5)) # %#@%@

print(symbol_password(10)) # @!%%$%$%%#
```

11. Define a class, ***Circle***, that takes two arguments when defined: a *sequence* and a *number*. The idea is that the object will then return elements the defined number of times. If the number is greater than the number of elements, then the sequence repeats as necessary.

Hint: You may want to define the class such that it uses an iterator (`CircleIterator`).

Sample Input: `c = Circle('abc', 5)`
Sample Output: `a b c a b`

12. You are tasked with managing a **zoo** consisting of multiple **cages** where animals are stored. Each cage can contain multiple animals, and each animal has specific attributes, such as its **color** and the **number of legs** it has.

a. **Zoo Management**:
   - The zoo contains several cages, and each cage can hold multiple animals.
   - The zoo should allow adding new cages to the zoo, and each cage has a unique identifier (such as a number).
   - The zoo should be able to print out information about the cages and the animals contained within them.

b. **Animal Information**:
   - Each animal has at least two key attributes:
     - **Color**: The animal's color.
     - **Number of Legs**: The number of legs the animal has (e.g., 2 for a bird, 4 for a dog, etc.).

Wraith a program on **zoo management**. Your program should have the following functionalities:

- **Add Animals to Cages**: Assign multiple animals to a cage.
- **Filter Animals by Color**: Return a list of animals that share a specific color.
- **Filter Animals by Number of Legs**: Return a list of animals that share a specific number of legs.
- **Calculate Total Number of Legs**: Compute the total number of legs of all animals in all cages combined.

13. Enter a number and have the program generate PI (π) up to that many decimal places. Keep a limit to how far the program will go. Take input from the user.

14. Given a string, you need to find the longest substring that contains **at most two distinct characters**. The substring can contain exactly two distinct characters or just one distinct character. Print the corresponding substring too.

Sample Input: s = `"eceba"`

Sample Output: `3 ece`

15. Given a list of integers, you need to find the longest sublist of consecutive numbers in the list. The sublist does not need to be in sorted order, but the numbers must be consecutive.

Sample Input: `arr = [100, 4, 200, 1, 3, 2]`

Sample Output: `[1, 2, 3, 4]`