June 3, 2024

ICS3U Task 4
- Sequences: Lists and Strings -

The purpose of this task is to practice using ***lists*** and ***strings***. Important when assessing this task will be:

- Completeness and correctness (task complete, meeting specifications, and working)
- Design (minimal duplicate code, good logic and structure)
- Clarity of code (whitespace, good variable names, documentation/comments, etc.)

Submission*: Thursday June 13th by the start of class.*
Submit **`task4_functions.py`** and **`task4_tester.py`** through the Task 4 assignment post in the Google Classroom.

- ***The final days of the course will be used to provide an overview of the exam, but will not contain significant work in terms of reviewing***
- ***The final written exam will contain significant content that is based on the learning from this assignment***

Overview
In this task you will complete a ***subset*** of 8 functions in a file called **`task4_functions.py`** that use lists and strings. There is also code to complete in the interactive tester (much like on Task 3), though most of it is fully functioning and interactive.

"Wait… A Subset?"
There are more components defined within this task than on previous ones. As a result, you are not necessarily expected to ***complete*** them ***all*** in order to receive a level-4 evaluation. However, to receive a 'perfect' (if that is your goal) you should aim to complete all elements, without any flaws.

A breakdown is provided for you (*see last page of this handout*) in terms of how the work completed in the individual functions will contribute towards the knowledge, application, and thinking components of the overall evaluation. While this is not as detailed a mark breakdown as you receive with the final marked work, it adequately shows you where the different parts of the assignment reside in terms of the categories.

The more you complete, the better your potential mark will be. However, it is not a 'linear' scale, meaning each component doesn't apply equally. Mr. Foster will assess the full product all together when assigning final marks in the categories on this task, and it will vary for each student based on the work they produce. Do not ask questions like *"what mark will I get if I do X, Y and Z but not A and B?"*.

It should be noted that the requirements below are not listed in order of expected difficulty. You may complete them in any order. There is no dependence of one function upon another. The only dependence is with the two interactive tester components, which need to have `get_proper_divisors()` and `rolling_averages()` completed in order to work, so you may wish to prioritize those two.

The 8 Functions:

Function 1 – `get_closest()`

Write a function that takes a list of integers, and a single 'target' integer. The function will return the value in the list that is closest to the target (above or below it), or any one of the closest if there is a tie. You may assume that the list has at least 1 value in it. The list argument must not be mutated.

Example:
```
numbers = [5, 9, 0, -3, 15, 8, 4]
print(get_closest(numbers, 10))
```
prints **9**

Function 2 – `get_closest_pair()`

This function will work with two ***related lists*** of integers given as arguments. The function will return a 2-element list with the related pair of values that were closest to each other. A related pair is a pair of values at the same position, one in each list. The returned 2-element list will be `[list1_value, list2_value]`. In the case that the two lists are both empty or not the same length (and thus not truly related), the function should return an empty list `[]`. The list arguments must not be mutated.

Examples:
```
numbers1 = [5, 9, 0, -3, 15, 8, 4]
numbers2 = [12, 13, 8, 2, 0, 11, -1]
print(get_closest_pair(numbers1, numbers2))
```
prints **[8, 11]**
```
print(get_closest_pair(numbers1, [1, 2, 3]))
```
prints **[]**

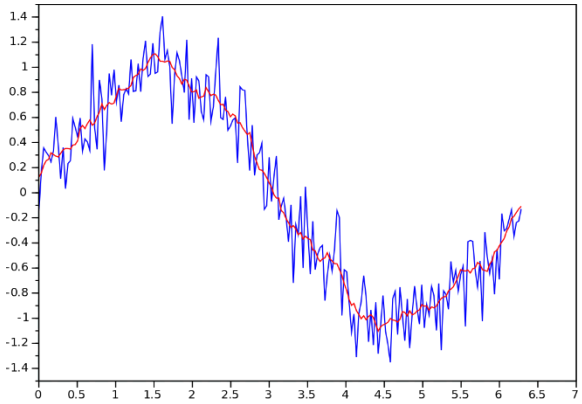
Function 3 – `get_proper_divisors()`

Write a function that takes a single positive integer and returns a list of all that number's proper divisors (from largest to smallest). A proper divisor of a number is any factor other than the number itself. So the proper divisors of 12 are 1, 2, 3, 4, and 6. The proper divisors of 16 are 1, 2, 4, and 8. There are ways to be more efficient your approach. One of the best ways is to realize that 1 is always a proper divisor (of any number ***except*** 1!), and the largest proper divisor would be *half* the number itself. That means if the number is 100, the largest number you would have to 'test' is 50.

Examples:
```
print(get_proper_divisors(12))
```
prints **[6, 4, 3, 2, 1]**
```
print(get_proper_divisors(29))
```
prints **[1]** ***← 29 is a prime number!***
```
print(get_proper_divisors(1))
```
prints **[]** ***← 1 has no proper divisors***
```
print(get_proper_divisors(268435457))
```
Hmmm……

Function 4 – `rolling_averages()`

Often data in a list is in a meaningful order, such as daily gas prices. Because gas prices can vary a lot on single days you will often see these amounts referred to using a '*rolling average*'. For example, a '5 day rolling average' would calculate the average gas price for a chunk of 5 consecutive days. Looking at this number gives a better representation of the trend in gas prices over time because it isn't as influenced by single day price jumps, but 'smooths out' the data a bit. The larger the 'chunk' you use for the average, the more it smooths out the data.



Write a function that takes a list of numbers and a *positive* integer chunk size as arguments. This function will return a new list which contains the averages of all consecutive 'chunks' of the given size. This resulting list will be smaller than the given list. You may assume the given chunk size is positive and not larger than the size of the list of values.

For example, if the list is **[3, 9, 2, 10, 14, 20, 13]** containing 7 values (with positions 0 – 6) and the chunk size is 2, the returned list will be:

$$\left[\frac{list[0]+list[1]}{2}, \frac{list[1]+list[2]}{2}, \frac{list[2]+list[3]}{2}, \frac{list[3]+list[4]}{2}, \frac{list[4]+list[5]}{2}, \frac{list[5]+list[6]}{2}\right]$$

$$= \left[\frac{3+9}{2}, \frac{9+2}{2}, \frac{2+10}{2}, \frac{10+14}{2}, \frac{14+20}{2}, \frac{20+13}{2}\right]$$

= **[6.0, 5.5, 6.0, 12.0, 17.0, 16.5]**

Notice how there are 6 items in the resulting list (1 less than the original).

Using the same list but with a chunk size of 4, the process would be:

$$\left[\frac{list[0]+list[1]+list[2]+list[3]}{4}, \frac{list[1]+list[2]+list[3]+list[4]}{4}, \frac{list[2]+list[3]+list[4]+list[5]}{4}, \frac{list[3]+list[4]+list[5]+list[6]}{4}\right]$$

$$= \left[\frac{3+9+2+10}{4}, \frac{9+2+10+14}{4}, \frac{2+10+14+20}{4}, \frac{10+14+20+13}{4}\right]$$

= **[6.0, 8.75, 11.5, 14.25]**

Notice how there are 4 items in the resulting list (3 less than the original).

The size of the returned list will always be (`chunk_size - 1`) *less* than the original list.

Examples:

```
numbers = [3, 9, 2, 10, 14, 20, 13]
print(rolling_averages(numbers, 2))  prints [6.0, 5.5, 6.0, 12.0, 17.0, 16.5]
print(rolling_averages(numbers, 4))  prints [6.0, 8.75, 11.5, 14.25]
print(rolling_averages(numbers, 10)) prints []
```

Function 5 – `align_strings()`

This function will take a list of strings, and a single string to search for. It will then identify all the strings that contain the substring (***ignoring all casing***). The matching strings will be built up as a single large string, with one original version of each string per line. Any strings in the list that don't contain the search substring will be ignored. Within each string, the substring being searched for will be all **UPPERCASE**, and the rest of the string will be all **lowercase**. Finally, each line will be aligned horizontally so that the search string will be in the exact same position for each line. In order to determine how much to space over each line you will first need to figure out which line has the most characters *before* the search string. This line will not need to be spaced over at all. If the search string appears more than once in a string, only the first occurrence should be made uppercase, and that should be what the alignment is based on.

Examples:

```
lines = ["Computer", "Science", "is", "a challenge", "for MANY", "people"]
print(align_strings(lines, "e"))
```

prints

```
computEr
   sciEnce
a challEnge
     pEople
```

for clarity→

	c	o	m	p	u	t	**E**	r		
				s	c	i	**E**	n	c	e
a		c	h	a	l	l	**E**	n	g	e
						p	**E**	o	p	l	e

```
words = ["mathematics", "radius", "theorem", "breathe", "apothem", "area"]
print(align_strings(words, "The"))
```

prints

```
maTHEmatics
   THEorem
breaTHE
 apoTHEm
```

for clarity→

		m	a	**T**	**H**	**E**	m	a	t	i	c	s
				T	**H**	**E**	o	r	e	m		
b	r	e	a	**T**	**H**	**E**						
	a	p	o	**T**	**H**	**E**	m					

`make_decks()`

This function is complete and will not be edited. It is useful to have this function so you can use card decks for the next functions.

Example:

```
decks = make_decks(2)
print(len(decks))    prints 104
print(decks)     prints ['A♠', 'A♠', 'A♣', 'A♣', 'A♥',  …  , 'K♥', 'K♦', 'K♦']
```

Function 6 – `deal_n_cards()`

This function will take a list of 'cards' as an argument (like the ones made by `make_decks()` but can be of ***any*** length) and a single integer representing the number of cards (*n*) to deal from the deck. The function will place the first *n* cards (from the front of the deck) into a new list (a 'hand') and return that new list. The cards that were 'dealt' should no longer be in the list of cards that was passed to the function, so the function will *mutate* that list. In the case that *n* is larger than the number of cards remaining in the deck, whatever remains will be dealt without crashing (and the returned hand will have less than *n* cards).

Example:

```
decks = make_decks(1)
hand = deal_n_cards(decks, 8)
print(len(hand), len(decks))     prints 8 44
print(hand)      prints ['A♠', 'A♣', 'A♥', 'A♦', '2♠', '2♣', '2♥', '2♦']
print(decks)     prints ['3♠', '3♣', '3♥', …  , 'K♣', 'K♥', 'K♦']
```

Function 7 – `cut_the_cards()`

This function will take a list of 'cards' as an argument (like the ones made by `make_decks()` but can be of ***any*** length) and will 'cut the cards' by randomly picking any place to divide the deck such that that there is at least 1 card in each of the 2 parts, and then will place the 'bottom' part on top. The order of each part will not change. This function will not return anything but will mutate the list itself.

Example:

```
cards = ['T♣', 'A♠', '7♥', '4♣', '3♦', 'K♦', 'K♥', '5♣']
cut_the_cards(cards)
print(cards)        prints ['7♥', '4♣', '3♦', 'K♦', 'K♥', '5♣', 'T♣', 'A♠']

cut_the_cards(cards)
print(cards)        prints ['K♥', '5♣', 'T♣', 'A♠', '7♥', '4♣', '3♦', 'K♦']
```

**Note that these results are based on randomness. The coloured text show how each 'part' was swapped.*

Function 8 – `shuffler()`

This function will take a list of 'cards' as an argument (like the one made by `make_decks()`, but can be of ***any*** length) and will apply a shuffling routine (described below) to a copy of that list and return the result. Unlike `deal_n_cards()` and `cut_the_cards()`, this function will ***not*** mutate the list.

The card shuffle routine is as follows:

- Cut the deck into two halves by slicing it in the middle (i.e. a left and a right half). In the case where there is an odd number of cards, the right half will have the extra card.
- Merge the two back together by *repeatedly* taking the ***first*** card from the **left** half **OR** the ***last*** card from the **right** half and placing this new card at the bottom of a new deck pile. The chosen half will be based on a **random** 2-sided coin flip.
- Once one half runs out of cards, place the remaining cards from the other half on the bottom of the new deck in the same order they are in.

Example:

```
cards = ['T♣', 'A♠', '7♥', '4♣', '3♦', 'K♦', 'K♥', '5♣']
cards = shuffler(cards)
print(cards)        prints ['5♣', 'T♣', 'A♠', 'K♥', 'K♦', '7♥', '3♦', '4♣']
```

How/Why?

- The list was split into 2 halves: `['T♣', 'A♠', '7♥', '4♣']` & `['3♦', 'K♦', 'K♥', '5♣']`
- Left half was `['T♣', 'A♠', '7♥', '4♣']` and the right half was `['3♦', 'K♦', 'K♥', '5♣']` which have been coloured here for clarity
- *Tails*: grab bottom from right half: `['5♣']`
  Heads: grab top from left half: `['5♣', 'T♣']`
  Heads: grab top from left half: `['5♣', 'T♣', 'A♠']`
  Heads: grab bottom from right half: `['5♣', 'T♣', 'A♠', 'K♥']`
  Tails: grab bottom from right half: `['5♣', 'T♣', 'A♠', 'K♥', 'K♦']`
  Heads: grab top from left half: `['5♣', 'T♣', 'A♠', 'K♥', 'K♦', '7♥']`
  Tails: grab bottom from right half: `['5♣', 'T♣', 'A♠', 'K♥', 'K♦', '7♥', '3♦']`
  *Right half is empty, add rest of left: `['5♣', 'T♣', 'A♠', 'K♥', 'K♦', '7♥', '3♦', '4♣']`

**Note that these results are based on randomness of the coin toss, so results may vary.*
An alternative possible result: `['5♣', 'K♥', 'T♣', 'K♦', '3♦', 'A♠', '7♥', '4♣']`

Interactive Tester: `task4_tester.py`

In addition to the sample test code from this handout provided at the end of the given starter file **`task4_functions.py`** you are provided with a very thorough interactive tester file (**`task4_tester.py`**). In that file you have full menu interaction provided (even a couple nested menus). Although most of this interactive tester is complete, there are **TWO** key components to complete yourself (marked with **'TO DO'** comments).

For multiple options in the tester code is provided here that allows the user to '***paste***' in a list of numbers and it will convert it to a proper Python list (this uses the `eval()` function which you have never seen, and don't need to understand).

The kind of list you will paste might look like:

```
[59, 58, 4, 95, 16, 45, -14, 37, 47, 26, 37, 66, 96, 73, 66]
```

or:

```
[45, 90, 63, -5, 47, 47, 24, 88, 82, 71, 35, 68, 39, 72, 17]
```

- ***Menu Choice 3*: number analysis**
  You probably know that a number is considered ***prime*** if it only has 1 as a proper divisor. In number theory, a number can also be classified based on the **sum** of its proper divisors. A number is:
  - ***abundant*** if the sum of its proper divisors is *greater* than the number itself
  - ***perfect*** if the sum of its proper divisors is *equal to* the number itself
  - ***deficient*** if the sum of its proper divisors is *less than* the number itself

  Code has been provided that asks for a low and high end for a range of numbers to analyze. You will add code that builds **4 lists** of numbers based on their classification (prime, abundant, perfect, deficient). Note that all prime numbers are deficient by definition and they would appear in ***both*** lists. You will output the results in the following exact format (sample shows low = 10, high = 30):

```
In the numbers from 10 to 30:
-> 5 are abundant: [12, 18, 20, 24, 30]
-> 1 are perfect: [28]
-> 15 are deficient: [10, 11, 13, 14, 15, 16, 17, 19, 21, 22, 23, 25, 26, 27, 29]
-> 6 are prime: [11, 13, 17, 19, 23, 29]
```

- ***Menu Choice 5*: all adjacent average**
  Code is provided here that allows the user to 'paste' in a list of numbers.
  You will add code that will repeatedly update this list of numbers to be the rolling average using a chunk size of 2. This will reduce the size of the list by 1 each time. You will repeat this until the list has just a single number in it, which will be considered the 'all adjacent average' of the original list.

  Finally, you will output 2 lines showing the 'all adjacent average' and the true actual average of the given list of numbers in the format below (no rounding at all).

  For example, if the input list is `[1, 45, -3, 78, -10, 0, 12]`:

```
All adjacent average: 25.75
Actual average: 17.571428571428573
```

Some Important Notes:

- All functions should all have *type hints* and properly formatted *docstrings*. You should also have ***internal documentation*** – comments which describe more complicated elements within the code or add readability/structure.
- You ***cannot*** use *any* programming concepts that we have not discussed yet in ICS3U or which are defined in this handout (list comprehensions, enumerate(), zip(), etc). If in doubt, ask Mr. Foster!
- Be sure to **test thoroughly**. Don't just accept the 'random looking' output without inspecting the cases closely to see if they are correct! All the test cases in this handout have been included as test cases in the `task4_functions.py` file.
- Remember the important style elements we have addressed: variable names, vertical whitespace (like paragraphs), etc. Look back at previous tasks' evaluation sheets, and the valuable feedback.
- Mr. Foster *wants* to help you! Ask good questions, show meaningful attempts, and get help when needed & warranted.

As a guideline, Mr. Foster's *entire* commented & whitespaced solution for the completed 8 functions file was between 150 and 200 lines (not including the given testing code) and the given starter file is already 50 lines.
The *completed* tester file was itself about 200 lines (the given file is already over 170 lines).

Evaluation Breakdown Guideline:

Communication:

- See your past evaluation sheets for the standard communication components.

Knowledge:

Primary focus for evaluating this category will be:

- `get_closest()`
  - identifies the closest, above or below target
- `get_proper_divisors()`
  - identifying factors
- `deal_n_cards()`
  - valid hand of cards returned
- `cut_the_cards()`
  - cards are cut in a valid way
- all output formatting to specification

Application:

Primary focus for evaluating this category will be:

- `get_closest_pair()`
  - handling non-related & empty lists
  - identifying closest
- `get_proper_divisors()`
  - list has complete and correctly formatted list
- `deal_n_cards()`
  - ensures list itself is mutated
- `cut_the_cards()`
  - ensures list itself is mutated
- tester choice 3
  - summarizes all numbers in correct lists
- tester choice 5
  - calculates both correct averages

Thinking:

Primary focus for evaluating this category will be:

- `rolling_averages()`
  - correctly calculates averages of 'chunks' and builds correct list
- `align_strings()`
  - multiple steps/components to break down this problem
- `shuffler()`
  - multiple steps/components to break down this problem
- efficiency of approaches to problems