Homework 5

Intro to Programming (Term I/2021–22)

built on 2021/10/28 at 22:05:39 ***due:*** *wed nov 10 @ 11:59pm*

This assignment will give you practice writing slightly larger programs, focusing on nested loops and nested structures. In this assignment, you will solve a number of programming puzzles and hand them in.

Be sure to read this problem set thoroughly, especially the sections related to collaboration and the hand-in procedure.

Overview:

Problem	File Name
1.	`bell.py`
2.	`hidden.py`
3.	`cipher.py`
4.	`jogging.py`

Problem	File Name
5.	`sleuth.py`
6.	`sgroup.py`

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Be sure to indicate who you have worked with (refer to the hand-in instructions).

Logistics

We're using a script to grade your submission before any human being looks at it. Sadly, the script is not as forgiving as we are. *So, make sure you follow the instructions strictly.* It's a bad omen when the course staff has to manually recover your file because the script doesn't like it. Hence:

- Save your work in a file as described in the task description. This will be different for each task. **Do not save your file(s) with names other than specified.**
- Before handing anything in, you should thoroughly test everything you write.
- You will upload each file to our submission site `https://assn.cs.muzoo.io/` before the due date. Please use your SKY credentials to log into the submission site. Note that you can submit multiple times but only the latest version will be graded.
- For some task, you will be able to verify your submission online. Please do so as it checks if your solution is gradable or not. Passing verification does not mean that your solution is correct, but, at least, it passes our preliminary check.
- At the beginning of each of your solution files, write down the number of hours (roughly) you spent on that particular task, and the names of the people you collaborated with as comments. As an example, each of your files should look like this:

```
# Assignment XX, Task YY
# Name: Eye Loveprograming
# Collaborators: John Nonexistent
# Time Spent: 4:00 hrs

... your real program continues here ...
```

- The course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online on Canvas discussion.

Task 1: Love Triangle, Ring A Bell (10 points)

For this task, save your work in `bell.py`

You will be given a positive integer $n > 0$ and you will construct a pattern that is made up of n rows:

- Row 0 contains 1 number—the number 1
- Each subsequent row is one longer than the one before and follows the pattern that you'll discover.

The first *five* rows are

```
[[1],
 [1,   2],
 [2,   3,  5],
 [5,   7, 10,  15],
 [15, 20, 27,  37,  52]
]
```

Your task: Implement a function `loveTri(n: int) -> list[list[int]]` that returns the first n rows of this (as a list of lists). (*Hint: how does a number relate to the number to the left of it and the number above that number?*)

Task 2: Hidden String (12 points)

For this task, save your work in `hidden.py`

Meow is playing a game with her friend. She gives her friend a string `s` and asks her whether another string `t` "hides" inside `s`. Let's be a bit more precise about hiding. Say you have two strings `s` and `t`. The string `t` hides inside `s` if all the letters of `t` appear in `s` in the order that they originally appear in `t`. There may be additional letters between the letters of `s`, interleaving with them.

Under this definition, every string hides inside itself: `'cat'` hides inside `'cat'`. For a more interesting example, the string `'cat'` hides inside the string `'afciurfdasctxz'` as the diagram below highlight the letters of `'cat'`:

`afciurfdasctxz`.

Importantly, these letters must appear in the order they originally appear. Hence, this means that `'cat'` does not hide inside the string `'xaytpc'`. In this case, although the letters `c`, `a`, and `t` individually appear, they don't appear in the correct order.

As another example, the string `'moo'` does not hide inside `'mow'`. This is because although we can find an `m` and an `o`, the second `o` doesn't appear in `'mow'`.

However, finding whether a string hides in another should ignore the case of the characters in those strings. For example, `'cat'` hides in `'Cat'`, and similarly `'Cat'` also hides in `'cat'`.

In this problem, you'll help Meow's friend by implementing a function `is_hidden(s, t)` that determines whether `t` hides inside `s`. The function returns a Boolean value: `True` if `t` hides inside `s` and `False` otherwise.

Here are some more examples:

```
is_hidden("welcometothehotelcalifornia","melon") == True
is_hidden("welcometothehotelcalifornia","space") == False
is_hidden("TQ89MnQU3IC7t6","MUIC") == True
```

```
is_hidden("VhHTdipc07","htc") == True
is_hidden("VhHTdipc07","hTc") == True
is_hidden("VhHTdipc07","vat") == False
```

Task 3: Writing A Cipher (12 points)

For this task, save your work in `cipher.py`

Hiding secrets from unwanted eyes has fascinated the humankind since ancient times. In the digital age, this is usually done by cipher algorithms. A cipher algorithm takes a string message called the *plaintext* and returns another string called the *ciphertext* that is (a) apparently hard to read, but (b) can be converted back into the original plaintext message. Turning the plaintext into the ciphertext is called encryption; reversing this process is called decryption.

In this task, you'll learn about a classic cipher called the transposition cipher and implement it.

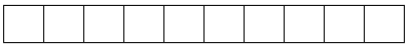
Transposition Cipher. As the name suggests, the basic idea of the transposition cipher is to take the message's symbols and jumble them up so that they become unreadable. The cipher decides how to move the symbols around using what's known as the *key*, denoted by k. To encrypt a text of length n, the key k is usually chosen to be a number between 2 and $k-1$ (inclusive). The exact encryption process is best described using an example.

Consider encrypting the text

```
Common␣sense␣in␣an␣uncommon␣degree␣is␣what␣the␣world␣calls␣wisdom
```

which contains 65 symbols, including spaces. The spaces are indicated by the ␣ symbol.

We'll use $k = 10$. First, we'll draw k boxes in a row—hence, drawing 10 boxes in our case.



Following that, we'll start filling the boxes from left to right using the symbols (letters and punctuation marks) from the input text. When you run out of boxes, you create a new row of k boxes and continue filling them until you accommodate the text entirely:

C	o	m	m	o	n		s	e	n
s	e		i	n		a	n		u
n	c	o	m	m	o	n		d	e
g	r	e	e		i	s		w	h
a	t		t	h	e		w	o	r
l	d		c	a	l	l	s		w
i	s	d	o	m	×	×	×	×	×

We include the ×'s in the last row only to remind ourselves that they aren't part of the input text and should be ignored in later processing.

To complete the encryption, we'll read the symbol in the first (leftmost) column from top to bottom, then the second column from top to bottom, so on so forth. Notice that when we hit an ×, we simply skip over it onto the next column. The resulting encrypted string is

```
Csngalioecrtdsm␣oe␣␣dmimetcoonm␣hamn␣oiel␣ans␣lsn␣␣wse␣dwo␣nuehrw
```

YOUR TASK: Implement a function `enc(msg: str, key: int) -> str` that takes a message string `msg` and a key `key`, $1 \leq$ `key` $<$ `len(msg)`, and returns a string representing the message encrypted using the transposition cipher (as explained above). Here are some more examples:

- `enc("abc",2) == 'acb'`
- `enc("monosodium glutamate", 7)=='mitouanmmo asgtoledu'`
- `enc("polylogarithmicsubexponential", 3)=='pygimseonaolatiuxntllorhcbpei'`

Task 4: Jogging Log (12 points)

For this task, save your work in `jogging.py`

Data geeks log their daily activities for no good reasons. In this problem, you are to process data in a logbook. The logbook is given to you as a list of strings. These are the lines from the logbook. Our goal is to derive this person's average jogging speed (ignoring data about all other activities).

The author logged data into her book carefully following a specific format. Hence, your input looks similar to the following example:

```
log_book = [
   "cycling;time:1,49;distance:2",
   "jogging;time:40,11;distance:6",
   "swimming;time:1,49;distance:1.2",
   "jogging;time:36,25;distance:5.3",
   "hiking;time:106,01;distance:8.29"
 ]
```

There can be as many lines (entries) as the logbook may contain. But each line always contains *three* pieces of information:

(1) the type of activity;
(2) the duration of that activity (time in minutes); and
(3) the distance involved in that activity (in km).

These three pieces are semicolon(`;`)-separated, and there will not be extra spaces anywhere. Therefore, for example, entry 0 in the example shows a cycling activity that lasted for 1 minute and 49 seconds involving 2 km of distance.

The author always uses a comma (`,`) to separate minutes from seconds. The seconds will always have 2 digits. For the distance component, she uses a dot (`.`) between the integral and the fraction parts. This dot is omitted if there is only the integral part. The jogging activity—the only activity relevant to us—is every line where the activity field is exactly `"jogging"`. This means, if the activity field is, for example, `'joggingandsprinting'`, it will not be counted.

YOUR TASK: It helps to break this big task down into smaller functions, representing different logical units. Your main goal is to write a function `jogging_average(activities: list[str]) -> float` that takes a list of strings similar to the example above and returns a floating-point number that equals the author's average jogging speed in m/sec (meter per second)[1]. To be very precise, your answer only has to be correct up to ± 0.0001. We promise also that there will be at least one jogging entry with a positive duration and distance.

Now, to implement this function, you may wish to implement the following helper functions:

- a function `parse_time(line: str) -> int` that takes a string (a line in the logbook) and returns the duration mentioned in this line (in seconds). For example,

[1] We know this is an usual unit for jogging speed, but we hope it helps everyone's sanity.

```
parse_time('jogging;time:40,11;distance:6')
```

should return $60 \times 40 + 11 = 2411$.

- a function `parse_dist(line: str) -> float` that takes a string (a line in the logbook) and returns the distance mentioned in this line (in km). For example,

  ```
  parse_dist('jogging;time:40,11;distance:7.1')
  ```

  should return 7.1.

EXAMPLES: Running `jogging_average` on the input described above, we should get 2.4586597. This is because there were *two* jogging activities:

- `"jogging;time:40,11;distance:6"`, which translates to 2411 seconds and 6 km; and
- `"jogging;time:36,25;distance:5.3"`, which translates to 2185 seconds and 5.3 km.

Therefore, the author has covered a distance of 11.3 km—that is, 11300.0 meters, in $2411 + 2185 = 4596$ seconds. Hence, the average jogging speed is

$$\text{Avg. Speed} = \frac{\text{Distance [m]}}{\text{Time [s]}} = \frac{11300.0\,[\text{m}]}{4596\,[\text{s}]} = 2.4586597\,[\text{m/s}],$$

so the function returns the floating-point number 2.4586597.

Task 5: Word Sleuth (12 points)

For this task, save your work in `sleuth.py`

Word sleuth (also known under various other names) is a popular word game found in newspapers and puzzle books. In such a puzzle, letters of words are placed in a grid, and the objective is to find and mark all the words hidden inside the grid.

You can read words:

- horizontally from left to right;
- vertically from top to bottom;
- diagonally from top-left to bottom-right; or
- diagonally from bottom-left to top-right.

A word w appears in a grid if w can be found along one of these directions of reading, with the letters of w appearing consecutively along the path of reading.

In this task, you'll implement a function

```
word_sleuth(grid: list[list[str]], words: list[str])
```

that takes as parameters:

- `grid` — a 2-dimensional list representing the input grid, and
- `words` — a list of words whose presence you look for in `grid`.

Your function will return a list of *all the words* in the `words` list that appear in the grid. The words may appear in any order in the output list but may *not contain duplicates*.

How to Get Started? You will want to break this problem into a few functions, each representing a logical unit of task. We suggest writing at least the following helper functions:

- `contains_word(grid: list[list[str]], w: str) -> bool` takes the grid from above and one *single* word and checks if the grid contains the word `w`. To implement this function, you may

wish to further break it down into (i) a function that walks the grid in a direction you specify and the string read-out that it encounters; and (ii) a function that checks if the word appears in the read-out string.

- `make_unique(lst: list[str]) -> list[str]` takes a list of words `lst` (with possible duplicates) and returns a list of words from `lst`, each with exactly one copy.

Example: Suppose you call `word_sleuth` with

```
words = ["bog", "moon", "rabbit", "the", "bit", "raw"]
```

and the following grid:

```
[["r","a","w","b","i","t"],
["x","a","y","z","c","h"],
["p","q","b","e","i","e"],
["t","r","s","b","o","g"],
["u","w","x","v","i","t"],
["n","m","r","w","o","t"]]
```

This should return a list with the following members `["raw", "bit","rabbit", "bog", "the"]`. Your program must output the same set of words but may (and probably will) output them in a different order. Note: even though `bit` appears twice in the grid, we only list it once in the output.

Ground Rules: For this task, you must **not** import any package. You must build everything from scratch.

Performance Expectations: While in this class, we aren't crazy about speedy code, your code must *not* be excessively slow. For this problem, it means finishing under 1 second on a 75×75 grid with about 50 words in the input.

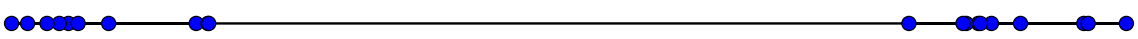
Task 6: Self Grouping (12 points)

For this task, save your work in `sgroup.py`

How can we automatically split seemingly-random data points into meaningful groups? For example, consider the heights of 20 people (in cm) presented in the ascending order:

```
71.4, 72.73, 74.36, 75.38, 76.15, 76.96, 79.51, 86.82, 87.81, 87.87, 146.38, 150.89,
151.16, 152.18, 152.36, 153.27, 155.7, 160.99, 161.36, 164.55
```

Say we happen to know that there are *two* age groups in this population—kids and adults. *How can we tell them apart?* It may not be clear at first glance, but plotting this reveals an interesting trend:



With the plot, it is visually trivial to split this dataset into two groups: there is a big gap in the middle, separating the heights into two logical sides. The more involved question—and the basis for this problem—is, **how can we discover this gap automatically?**

Answering this and similar questions has been a topic of extensive research, fueled by real-world problems ranging from deciding letter grades to giving suggestions on Facebook (e.g., who are your close friends?).

What we did in our brain was essentially locating the widest gap visually. This makes intuitive and mathematical sense because the widest gap is where our data points are the most dissimilar.

Generalizing this idea gives us a simple heuristic that often works okay on real data. Say we want to identify k groups. We can proceed as follows:

Step 1: Find the largest gap in the data.
Step 2: Split the data at that point.

This gives us 2 groups. To further split the data, we find the largest gap that hasn't been split and split it there. This gives us one more group. We can repeat this process until we get k groups.

Let us illustrate this idea with a simple example. Suppose $k = 3$ and our input is the following numbers:

10, 12, 45, 47, 91, 98, 99.

They look as follows in plot:

As before, we assume the input is sorted from small to large. And we wish to split it into $k = 3$ groups. To begin, we'll treat the input as one big group:

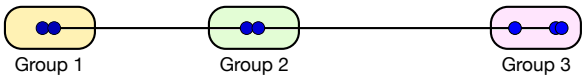
[10, 12, 45, 47, 91, 98, 99]

Then, we look for the largest gap between two consecutive elements. In this example, the largest gap is in between 47 and 91 (you can check that). That is where we want to split. After that, the list becomes:

[10, 12, 45, 47] [91, 98, 99]

Next, we find the largest gap between two successive elements in all groups. This time, the largest gap is in the first group (to know this, we'll have to check the other group, too). The overall largest gap is between 12 and 45. Hence, we further split it there, leading to the following groups (illustration on right):

[10, 12] [45, 47] [91, 98, 99]



We now have 3 groups, so we can stop and output them.

REMARKS: Mathematically, this process partitions a list into k sublists so that the *total gap*—the sum of all gaps—between sublists is maximized. It makes sense intuitively and aligns with out intuition that different groups should be dissimilar, so the process attempts to maximize how different they are.

YOUR TASK: You will write a function

```
separate(data: list[int], k: int) -> list[list[int]]
```

that partitions `data` into k sublists that maximize dissimilarity using the process described above. **Importantly:** for this task, your implementation *must* find the largest gap and split the dataset at the gap, repeating this process until you have k groups. If there are ties, pick the left-most gap. To keep things simple, we will assume that the input list is always ordered from small to large.

Hence, for example,

```
separate([10, 12, 45, 47, 91, 98, 99], 3)
```

should return

```
[[10, 12], [45, 47], [91, 98, 99]].
```