COMP 1900 - Fall 2021

Lab 6: Functions

Total Points: 36

Due: Fri., Nov. 5, by 2359 CDT
Please carefully read the submission instructions at the end of the assignment. As with all other assignments, this should be done *individually*.

Grader: The lab TA who supervises your registered lab section will grade your submission. Grades will generally be posted within 1-2 weeks of the assignment due date. Questions about grading? Please contact them first.

1. You're throwing a wild, socially distanced party for yourself and 9 of your closest friends. Since this party is *wild*, you expect each person (including yourself) to drink 14 cans of uh... kombucha. You don't want to disappoint any of your friends (or yourself), so you need to make sure there's enough kombucha available: 10 people $\times$ 14 cans/person = 140 cans. Suppose that you can't buy kombucha in individual cans – it's sold only in 6-packs. Thus, you need to buy at least 24 of those 6-packs (giving you a total of 144 cans) to satisfy everyone. The extra 4 cans can be saved for next time, or perhaps used as an incentive to encourage your guests to help you clean up afterward.

After throwing several of these parties, you're getting tired of doing the math yourself every time. Let's do the computer scientist thing and generalize the problem so you can write some code to solve it. You want to throw a party for yourself and f friends, each of whom will drink c cans of a wholesome, healthy beverage which is sold only in packs of p cans each. (As before, assume that you'll be drinking yourself too.)

Within a file named `party_planner.py`, do the following:

(a) (3 points) Write a function `plan_party(f, c, p)` that computes and displays the number of p-packs needed to supply a party for you and f of your friends, each of whom will drink c cans. You may assume that all the parameters are integers, $f \geq 0$, $c \geq 0$, and $p \geq 1$. The function should also compute and display how many extra cans you'll have. Note that this function should only *print* its results on the screen; it does not *return* any value.

Here are some example arguments for the `plan_party` function and their expected output on the screen:

Arguments	Output on Screen
(9, 14, 6)	`24 6-pack(s) needed` `4 extra can(s)`
(4, 6, 3)	`10 3-pack(s) needed` `0 extra can(s)`
(4, 6, 4)	`8 4-pack(s) needed` `2 extra can(s)`

(b) (3 points) Write a function `plan_party2(f, c, p)` that performs the same calculations as the previous part, but *returns* the number of p-packs needed. `plan_party2` does not need to consider the number of extra cans, and it should not print anything.

Here are some example arguments for the `plan_party2` function and their expected return values:

Arguments	Return Value
(9, 14, 6)	24
(4, 6, 3)	10
(4, 6, 4)	8

(c) (2 points) Below your function definitions in `party_planner.py`, write a program that gets user input for number of friends, number of cans per person, and number of cans per pack. The program should then call both `plan_party` and `plan_party2`. Display the return value from `plan_party2` at the end.

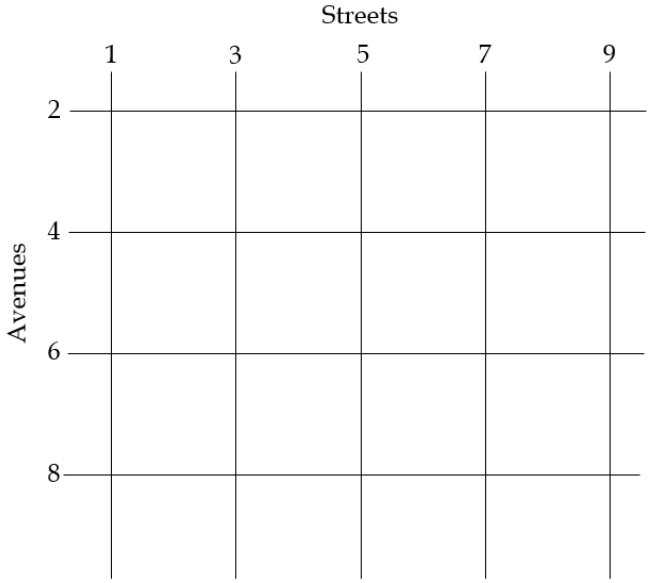
Example program run (underlined parts indicate what the user enters)

```
How many friends are you throwing this party for? 9
How many cans will each person drink? 14
How many cans are in each pack? 6

24 6-pack(s) needed
4 extra can(s)
Return value from plan_party2: 24
```

2. Suppose you live in a city where the roads form a grid. **Streets** run north-south and are odd-numbered: 1st St., 3rd St., 5th St., etc. from west to east. **Avenues** run west-east and are even-numbered: 2nd Ave., 4th Ave., 6th Ave., etc. from north to south.

   The map below shows the first several streets and avenues. Note that the city is substantially larger than what's shown here; this partial map is just to give you a visual aid for how the roads are laid out.



   The roads are all equally spaced, 1000 ft apart. The city is well developed, so you cannot cut across any blocks when traveling. For example, to get from the corner of 1st St. and 6th Ave. to the corner of 3rd St. and 4th Ave., you would have to travel one block north on 1st and one block east on 4th, or one block east on 6th and one block north on 1st. Both routes have a total traveling distance of 2000 ft.

   Within a file named `mapping_app.py`, do the following:

   (a) (3 points) Write a function `get_input_st()` that reads user input for a street and returns the value that was read. You may assume that the user will enter an integer, but the function should include input validation to ensure that the input is a positive odd integer.

(b) (3 points) Write a function `get_input_ave()` that reads user input for an avenue and returns the value that was read. You may assume that the user will enter an integer, but the function should include input validation to ensure that the input is a positive even integer.

(c) (5 points) Write a function `num_to_ordinal(n)` that returns a string representing the ordinal version of the number n. You may assume that n is a positive integer, but the function should work correctly for *any* positive integer value of n.

Here are some example arguments for the `num_to_ordinal` function and their expected return values:

Argument	Return Value
1	'1st'
2	'2nd'
3	'3rd'
4	'4th'
2011	'2011th'
2021	'2021st'

Hints:

- There are only four possible endings that can be added to the end of the number: *st*, *nd*, *rd*, and *th*. The last two digits of n are enough to determine which ending to use.
- Make sure you consider the unconventional behavior of 11, 12, and 13. These use the *th* ending; we say "eleventh," "twelfth," and "thirteenth" instead of "elevenst," "twelvend," and "thirteenrd."

(d) (15 points) Write a function `get_directions(start_st, start_ave, end_st, end_ave)` that shows step-by-step directions for traveling between the specified start and end corners. This function only needs to print the directions; it should not return anything. You may assume that the function has valid parameters (i.e., all the street and avenue numbers make sense). Call your previously written `num_to_ordinal` function as needed to show the road numbers nicely.

For example, calling `get_directions(1, 6, 3, 4)` should print something like this:

```
Directions from 1st and 6th to 3rd and 4th:
Take 6th Ave. east for 1000 ft until you get to 3rd St.
Turn left onto 3rd St.
Take 3rd St. north for 1000 ft until you get to 4th Ave.
Yay, you've arrived!
```

This function likely seems very challenging at first glance, but don't panic! Pretend you're a sloth just chilling in the rain forest, and take it slowly step by step. Here are some hints to get you started:

- The trip can be broken down into one horizontal component and one vertical component.

- For the horizontal component, the distance traveled depends on the numbers of the starting and ending streets. The direction traveled (west or east) depends on whether the number of the starting street is larger than or smaller than the number of the ending street. The vertical component can be handled very similarly, using the starting and ending avenues.
- If a turn is involved, there are only two possible directions: left or right. What combinations of horizontal and vertical directions produce each kind of turn?
- There are a few situations in which no turn is needed: a horizontal-only trip, a vertical-only trip, or a case where the starting and ending points are the same (in that case, you're already at your destination). Your function should handle all of these!

(e) (2 points) Below your function definitions in `mapping_app.py`, write a program that gets user input for a starting and ending corner, then displays directions between those points. Call your previously written functions as needed. Because the functions are already doing most of the work required, this part should be quite short!

Example program run (underlined parts indicate what the user enters)

```
Starting corner:
Street: -1
Street must be a positive odd integer!  Try again: 5
Avenue: 14

Ending corner:
Street: 101
Avenue: 54

Directions from 5th and 14th to 101st and 54th:
Take 14th Ave. east for 48000 ft until you get to 101st St.
Turn right onto 101st St.
Take 101st St. south for 20000 ft until you get to 54th Ave.
Yay, you've arrived!
```

Example program run (underlined parts indicate what the user enters)

```
Starting corner:
Street: 5
Avenue: 2

Ending corner:
Street: 5
Avenue: 10

Directions from 5th and 2nd to 5th and 10th:
Take 5th St. south for 4000 ft until you get to 10th Ave.
Yay, you've arrived!
```

Code Guidelines

Points can be deducted for not following these guidelines!

- Most importantly, your code *must* run. Code that does not run due to syntax errors may receive zero credit, at the TA's discretion.
- Follow Python capitalization conventions for `variable_and_function_names`.
- Use consistent indentation throughout your code. (Python kind of forces you to do this!)
- Include a reasonable amount of comments in your code. "Reasonable" is somewhat subjective, but at the very least include:

  1. A comment at the top of each source code file summarizing what it does.
  2. Comments that indicate the major steps taken by the program. There are generally at least two or three of these, such as collecting user input or making calculations.

Need Help?

- Attend your weekly lab meeting. Your lab TA can give you live help during this session.
- Contact your lecture instructor, and/or attend office hours.
- The UofM offers free online tutoring through the Educational Support Program (ESP): `https://www.memphis.edu/esp/onlinetutoring.php`

Submission Instructions

- Create a zip file containing your Python source files. You can use any Python development environment you like, as long as you submit the source files.
- Upload your zip file to the appropriate dropbox folder on eCourseware. The dropbox *will* cut off submissions at precisely the stated deadline, so please submit with some time to spare. Late submissions are not accepted. You may submit as many times as you want up to the deadline. Each submission overwrites the previous one.
- Contact your lab TA to schedule a one-on-one code review via Zoom. During this short meeting (15-30 minutes), your TA will ask you to run your code. They may also ask you to explain your code and/or run different test cases. The code review must be completed for you to get a grade for the lab.

  Note that since the code review happens after your submission, its purpose is *not* for you to get help with writing your code! Getting help should be done during the scheduled lab sessions, or by reaching out to your lecture instructor/online tutors *before* the due date.