CIS1100.py — Fall 2024 — Exam 1

Full Name: _____

PennID (e.g. 12345678): _____

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

_____ _____

Signature Date

Instructions are below. Not complying will lead to a 0% score on the exam.

- Do not open this exam until told by the proctor.
- You will have exactly 60 minutes to take this exam.
- Make sure your phone is turned OFF (not on vibrate!) before the exam starts.
- Food and gum are not permitted—don't be noisy or messy.
- You may not use your phone or open your bag for any reason, including to retrieve or put away pens or pencils, until you have left the exam room.
- This exam is closed-book, closed-notes, and closed computational devices.
- If you get stuck on a problem, it may be to your benefit to move on to another question and come back later.
- All code must be written in proper Python format.
- Do not separate the exam pages. Do not take any exam pages with you. The entire exam packet must be turned in as is.
- Only answers on the FRONT of pages will be graded. There are two blank pages at the end of the exam if you need extra space for any graded answers.
- Use a pencil, or blue or black pen to complete the exam.
- If you have any questions, raise your hand and a proctor will come to you.
- When you turn in your exam, you may be required to show your PennCard. If you forgot to bring your ID, talk to an exam proctor immediately.
- We wish you the best of luck!

Q1	Q2	Q3	Q4	Q5	Q6 (bonus)

Q1. Types Fill In The Blank

In the column marked **“Type,”** choose the type (int, float, bool, str, list, set, tuple, range) for the final expression in the snippet, or write **"error"** if there is an error in the expression. You do not need to write the value of the expression. In cases where multiple answers are possible, any of them will be accepted.

Statement	**Type**
3 in {1, 2, 3, 4}	
13.4 / 18.43	
[1, 2] + [3, 4]	
"CIS 1100" - 0	
"CIS 1100" + "0"	
len(sys.argv)	
l = [3, 4, 5, 6, 7, 10, 12] l[0]	
l = [3, 4, 6, 7, 10, 12] l[0:2]	
17 % 4	

Q2. Values Fill In the Blank

Write the value that gets printed, or write **"error"** if there is an error during the execution of these lines of the program. **Hint:** remember that **None** is a value.

Question 2.1

```
print(int("4") // 3)
```

Answer:

Question 2.2

```
x = [1, 2, 3]
x[len(x) - 2] = x[len(x) - x[1]] - x[1]
print(x[1])
```

Answer:

Question 2.3

```
s = {1, 2, 3}
print(s[0])
```

Answer:

Question 2.4

```
print(f"{10} + {20}")
```

Answer:

Question 2.5

```
nums = [1100, 1200, 1210, 1600]
result = nums.append(2620)
print(result)
```

Answer:

Q3. Debugging

Help fix this buggy implementation of **mealswipe_predictor**. This function should return the number of days *before* the end of the semester that you will run out of meal swipes. Provide your answers in the table on the next page—identify each of the five lines with errors and provide corrections for those lines. (There are actually six errors, but the first error is solved for you.)

If the swipe history is empty or you always use 0 meal swipes per day, it should print an error message and return None. If you have more swipes left than you need, the result will be negative, indicating how many extra days you could survive on meal swipes after the semester ends.

```
"""
Input: num_days -> int, number of days in a full semester
       num_swipes -> int, number of swipes in your mealplan
       swipe_history -> list of ints, swipe_history[i] represents
                        the number of swipes used on day i, where the first
                        day of the semester is day 0
Examples:
       mealswipe_predictor(365, 365, [1]) -> 0
       mealswipe_predictor(365, 365, [365]) -> 364
       mealswipe_predictor(365, 380, [1]) -> -15
       mealswipe_predictor(30, 12, [1, 2, 3]) -> 24
       mealswipe_predictor(30, 12, [0, 0, 0, 0, 0, 0]) -> None
       mealswipe_predictor(30, 12, []) -> None
"""
1.  define mealswipe_predictor(num_days, num_swipes, swipe_history):
2.
3.      if len(swipe_history) == 0:
4.          print("Not enough data to make a prediction.")
5.          None
6.
7.      days_so_far = swipe_history.len()
8.      total_swipes_used = sum(swipe_history)
9.      avg_uses_per_day = days_so_far // total_swipes_used
10.
11.     if avg_uses_per_day != 0:
12.         print("On average, you are not eating. Please reconsider.")
13.         return None
14.
15.     remaining_swipes = num_swipes - total_swipes_used
16.     days_remaining = remaining_swipes // avg_uses_per_day
17.
18.     return num_days + days_so_far - days_remaining
```

Line Number	Incorrect Code	Replacement Code
1	`define`	`def`

Q4 Complete the Program: *calendar.py*

The following program models a person's calendar of monthly recurring events. The program maintains two lists: **days** and **events**. The lists always have the same lengths. For each index **i**, we store in **days[i]** the day on which a particular event takes place. The name of that event is **events[i]**. The program should read a day (as a number from 1-31, inclusive) as the first command line argument and then search through the **days** list, printing out all of the names of events that take place on the input day. The program should also print out a warning about events that are scheduled to take place on the following day. Keep in mind that the day following **31** would be **1**! Finally, if the day provided as a command line argument is not a number between 1 and 31 inclusive, print an error message. There are some example executions below; the code to complete and the table for writing answers are on the next page.

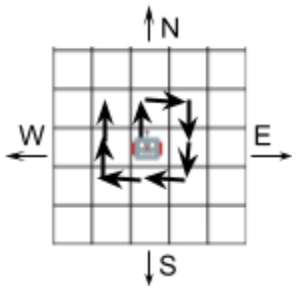
$ python calendar.py 15 TODAY: Bowling TODAY: Movie Night TOMORROW: Pay Bills	$ python calendar.py 31 TOMORROW: Pay Rent TODAY: Deep Clean	$ python calendar.py 0 Invalid day!

```
import ___BLANK_0___

days = [1, 5, 15, 15, 5, 16, 31]
events = ["Pay Rent", "Team Meeting", "Bowling", "Movie Night",
          "Grocery Shopping", "Pay Bills", "Deep Clean"]

input_day = ___BLANK_1___                    # read from the command line
if input_day < 1 ___BLANK_2___ input_day > 31:  # check if input is invalid
    print("Invalid day!")
else:
    next_day = ___BLANK_3__ + 1  # find the next day, wrap around if needed
    for idx, day in enumerate(days):
        if ___BLANK_4___:                    # found an event for input day
            print("TODAY: " + events[idx])
        elif day == nextDay:                 # found an event for next day
            print("TOMORROW: " + ___BLANK_5___)
```

Blank #	Code
0	
1	
2	
3	
4	
5	

Q5. Coding: FarmBot

FarmBot is a little robot responsible for inspecting crops in a field. He emits a string of moves that he took on his travels, called a **history**.



In a history, a "0" indicates one move north, a "1" indicates one move east, a "2" indicates one move south, and a "3" indicates one move west. For example, the history "01223300" would indicate the path on the right, assuming that FarmBot starts at the location marked by 🤖.

First, implement a function that converts a history into a **summary**, which is a list that counts the total number of moves north, east, south, and west. The summary for the history above would be [3, 1, 2, 2]. (A summary list always has length four.)

Question 5.1

```
"""
Input: a history of FarmBot's moves
Output: a summary of the moves, as defined above
Examples: history_to_summary("") => [0, 0, 0, 0]
          history_to_summary("0") => [1, 0, 0, 0]
          history_to_summary("3333") => [0, 0, 0, 4]
          history_to_summary("01223300") => [3, 1, 2, 2]
"""
def history_to_summary(history):
```

Second, implement a function that determines from a summary whether FarmBot is **burnt out**, which means he moved more than five times in any direction. This is not based on FarmBot's final location but instead on the total number of times he moved in each direction.

Question 5.2

```
"""
Input: a summary of FarmBot's moves
Output: whether FarmBot is burnt out, as defined above
Examples: is_burnt_out([1, 2, 3, 4]) => False
          is_burnt_out([5, 5, 5, 5]) => False
          is_burnt_out([3, 4, 5, 6]) => True
          is_burnt_out([6, 0, 0, 6]) => True
"""
def is_burnt_out(summary):
```

Finally, implement a function (on the next page) that determines from a history whether FarmBot **needs to be picked up**. He needs to be picked up if he ever took three or more steps north in a row (three or more consecutive **0** moves in his history) or he is burnt out. Use the previous two functions you wrote to complete this one in **three or fewer lines.**

Question 5.3

```
"""
Input: a history of FarmBot's moves
Output: whether FarmBot needs to be picked up, as defined above
Examples: needs_pickup("01230123") => False
          needs_pickup("11100333222") => False
          needs_pickup("111000333222") => True, there's a sequence of 000
          needs_pickup("0100010011") => True, six 0s
"""
def needs_pickup(history):
```

Q6. (Bonus)

Create a piece of art (e.g. drawing, poem, short program, anything you like)!
If you are unsure of what to make, select one member of the course staff and make art about that person. All exams will get full credit for this question even if you leave it blank.

Extra Answers Page (This page is intentionally blank)

You may use this page for additional space for answers; keep it attached to this exam. Clearly note on the original question page that your answer is on this extra page, and clearly note on this page what question you are answering.