CSCI 150: Exam 2 Practice

October 17, 2023

Directions: Exam #2 occurs in two parts, an in-class and take-home.

In-class: The in-class portion will take place during regular class time on Monday, October 23. You will have 50 minutes to complete the exam. For the in-class portion, you may not use a computer, calculator, or other device. It will consist of tracing the execution of three pieces of code and writing two straightforward functions by hand. You must work the in-class without reference to notes or any other resource.

Take-home: The take-home will be given to you as you leave the in-class exam. You should create a single `.py` file, and upload your file via CodeGrade. You will be asked to write 5 separate functions. Your solutions are due at the start of class on **Wednesday, October 25**. Like on the Codingbat conditional homework and the practice exam below, example input and output will be provided. For the take home, the *only* resources you may use are the following:

- any code you have created for class, including homework and labs
- any notes you have taken during class
- any information, code, or other material directly linked from the class homepage, `https://hendrix-cs.github.io/csci150/`
- anything in the official Python documentation, `https://docs.python.org/3/`

You may not talk to a classmate, friend (real-life or Facebook), Siri, ChatGPT, or anyone other than me about this exam until you turn it in, nor search the Internet or library or any reference other than those listed above for assistance. You may not even mention anything about how long it took you to complete the exam, that you found problem #2 particularly difficult (or easy), or in fact talk at all about the exam or Computer Science with anyone other than your instructor from Monday, 8:10am–Wednesday 12:00pm. Anyone who *gives* answers is equally in violation of the Academic Integrity Policy as one who *receives* them.

If you have ANY questions about whether something you would like to do is allowed, please ASK!

Important! The use of a `for` loop is not required on any problem on this exam, but you are welcome to use one where it is appropriate. A `while` loop will also work where repetition is needed.

Specification: Exam #2 will be marked **Complete** provided:

- Each problem is attempted
- On the in-class:
  - At least one tracing problem is completely correct, one other has a single mistake, and the third makes clear that you understand how the stack works.
  - Excepting minor syntax issues, at least one by-hand code writing problem is correct, and the other misses a single case; both answers exhibit clear evidence that the student understands the basic structure of Python functions, conditionals, and loops.
- On the take-home:
  - No syntax errors in any code
  - No runtime errors with any of the examples given
  - 3 out of 5 produce the right answer in all circumstances, 1 produces the right answer in all examples given, and the last produces right answers in at least two of the given examples

Exam #2 will be marked **partially complete** provided:

- At most one problem on the in-class and one on the take-home are not attempted
- On the in-class:
  - At least one tracing problem has all calls to the stack correct; loop variables are mostly updated correctly on at least one tracing problem.
  - By-hand code: At least one attempted solution would work in some circumstances; syntax errors are minor, and the work shown shows a basic grasp of Python structure
- On the take-home:
  - No syntax errors in any code
  - At least 3 functions produce no runtime errors with the examples given and 2 produce no runtime errors at all
  - At least 1 function produces the right answers in all circumstances, and one other produces the right answer on the examples given.

The instructor will return your exam along with a rubric detailing your progress and assessment. If you earned less than a **Complete** instructions will be provided on that rubric about how to attempt to raise your assessment.

In general, the student will be expected to correct all errors, complete additional problems as assigned, and meet with the instructor in office hours to discus their work.

In-Class Practice

1. Trace the following code:

```
def main1():
    a = 4
    b = 7
    s = 1

    while a < b:
        s *= a
        a += 1

    print(s)

main1()
```

2. Trace the following code:

```
def f1():
    print('Hello there')

def f2(x: int):
    if x < 2:
        f1()
        x = 7
    elif x == 7:
        f1()
        f1()
    else:
        f1()
        print('Goodbye!')
        f1()

def main2():
    f2(5)
    f2(0)


main2()
```

3. Trace the following code:

```
def top(s: str) -> bool:
    i = 0
    k = 0
    while i < len(s):
        if s[i] <= 'm':
            k +=1
        i += 1
    return k

def main3():
    print(top('exam'))
    print(top('hello'))

main3()
```

For the next two, write code which will accomplish the following task. As practice for the in-class part of the exam, you should do this without any use of a Python interpreter – that is, you should do this entirely by hand.

4. Write a function `same_char` which takes in two string parameters, `s` and `t`, which are guaranteed to be of the same length (you do not have to check or worry about this). It should return an integer which counts the number of times that the corresponding characters in the two strings are equal.

   For example, `same_char('smart','start')` should return `4`, since the `'s', 'a', 'r', 't'` all match. Likewise, `same_char('aaaa', 'bbab')` would return `1`.

5. Write a function `positive_only` which takes in a list of integers, `lst`, as its parameter. It should return a list which contains exactly the entries in `lst` which are positive.

   For example, `positive_only([4, -5, 0, 1, 2])` should return `[4, 1, 2]`, and `positive_only([-3, 0, -1])` should return `[]`.

Take-Home Practice

1. Write a function `word_count` which asks the user to enter a word, and continues to ask this question until the user enters 'exit' at which point the function returns the number of words entered (not counting 'exit' itself). The function `word_count` should take no parameters, but return an `int`.

   Consider the following example transcripts:

   ```
   Please enter a word ('exit' to quit): apple
   Please enter a word ('exit' to quit): high school
   Please enter a word ('exit' to quit): apple
   Please enter a word ('exit' to quit): exit
   ```

   The function should return `3`. (Notice that it does not care that 'apple' was given twice, nor that 'high school' has a space. It simply counts the number of entries made by the user until 'exit' is input). Here are two more example transcripts:

   ```
   Please enter a word ('exit' to quit): sdsdfsdf
   Please enter a word ('exit' to quit): car
   Please enter a word ('exit' to quit): banana
   Please enter a word ('exit' to quit): treadffd
   Please enter a word ('exit' to quit): exit
   ```

   will return `4`. (Again, it does not care or check in any way whether the entries are "real" words.)

   Finally,

   ```
   Please enter a word ('exit' to quit): exit
   ```

   will return `0`.

   Note: The function you write will require an `input` statement as part of the code! (You do not need to try to "validate" that input in any way.)

2. Write a function `my_replace` which takes in three string parameters, `s`, `t`, and `u`, where `s` can be of any length and `t` and `u` are each always a single character, and returns `s`, but with each occurrence of `t` in `s` replaced with `u`. You must do this using a `while` loop or a `for` loop and without using the built-in Python `replace` method.

   For example:

   - `my_replace('hello','l','m')` should return `'hemmo'`
   - `my_replace('hello','h','m')` should return `'mello'`

- `my_replace('hello','x','m')` should return `'hello'`
- `my_replace('','l','m')` should return `''`
- `my_replace('testing','t','q')` should return `'qesqing'`

3. Write a function `suffix_to_prefix` which takes two parameters, a string `s`, and a non-negative integer `n`. It should take the final n characters of the string (the suffix) and return a string with them as the prefix to the original string. (The value of n will always be less than or equal to the length of the string.

   For example:

   - `suffix_to_prefix('string', 3)` should return `'ingstr'` (the final three characters of 'string' have moved to the front)
   - `suffix_to_prefix('traffic', 2)` should return `'ictraff'` (the final two characters of 'traffic' have moved to the front)
   - `suffix_to_prefix('hendrix', 4)` should return `'drixhen'`
   - `suffix_to_prefix('hendrix', 0)` should return `'hendrix'` (since 0 characters were moved)
   - `suffix_to_prefix('hendrix', 7)` should return `'hendrix'` (since all of the characters were moved!)

4. Write a function `sum_between` which takes in two integer parameters `a` and `b` and returns the sum of all of the integers between a and b (inclusive). If $a > b$, it should return 0.

   For example:

   - `sum_between(3,5)` should return 12
   - `sum_between(-2,4)` should return 7
   - `sum_between(6,6)` should return 6
   - `sum_between(7,2)` should return 0 (since $7 > 2$).
   - `sum_between(-3,3)` should return 0 (since $(-3)+(-2)+(-1)+0+1+2+3=0$)

5. Write a function `make_num` which takes in a list of non-negative, single-digit integers `lst` and returns a single integer which is the result of concatenating the integers together. The list will always contain at least one entry.

   For example:

   - `make_num([7, 3, 0, 1])` should return `7301`
   - `make_num([5])` should return `5`
   - `make_num([1, 2, 3, 4, 5])` should return `12345`

Hint: There are at least two distinct ways to do this:

- Convert the entries to strings, glue them together, and convert back to an integer
- Use "place arithmetic" – that is, in the first example above, `[7, 3, 0, 1]` corresponds to

$$7 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$$

Either method is valid.

6. Write a function `list_min` which takes in a list `nums` of integers and returns the *index* of the smallest integer in the list. You can assume that `nums` will always be non-empty (that is, it is OK if your function crashes when given an empty list as input). If the smallest integer appears more than once, the first index (i.e. lowest number) should be returned.

   For example:

   - `list_min([3, 9, 1, 4, 3, 1, 7])` should return `2` (since `1` is the smallest integer and its first occurrence is in index 2.)
   - `list_min([7])` should return `0`
   - `list_min([-2, -5, 0, 1, -2])` should return `1`
   - `list_min([3, 3, 3])` should return `0`