CSCI 150: Exam 3 Practice

Friday, April 14, 2023

Directions: Exam #3 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, April 17. 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 on Monday, April 17. You should create a single `.py` file, and submit your file as instructed. You will be asked to write 5 separate functions. Your solutions are due by classtime on **Wednesday, April 19**. Like on the Codingbat homeworks 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
- code and any other material that Prof. Seme, Dr. Stanley, or Prof. Yorgey have posted to either the lecture or lab Teams pages
- any information 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, 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, 8:10am. Anyone who *gives* answers is equally in violation of the Academic Integrity Policy as one who *receives* them. All suspected violations will be reported to that committee.

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

Specifications Assessment: Exam #3 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 and heap work.
  - 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, dictionaries, and classes.
- 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 #3 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; at least one problem shows significant understanding of the heap.
  - By-hand code: At least one attempted solution would work in some circumstances; syntax errors are minor, and the work shown shows a basic graph of Python structure; at least one shows understanding of either dictionaries or classes
- 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 most cases, a **Missing** can only be "upgraded" to a **Partially Complete,** though the instructor reserves to right to allow **Missing** to become **Complete** in some circumstances.

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 Tracing:

1. Trace the following code, showing all interactions with the stack and the heap.

```
def main1():
    a = [7, 2, 6, 1]
    b = a
    c = [7, 2, 6, 1]

    for item in c:
        if item > 4:
            a.append(item * 2)
    print(b)
    print(c)

main1()
```

2. Trace the following code, showing all interactions with the stack and the heap.

```
def f1(s: str) -> int:
    if 'a' in s:
        return 0
    print('No a')
    return len(s)

def main2():
    lst = ['test', 'exam', 'ant', 'banana', 'smile']
    d = {}

    for item in lst:
        d[item] = f1(item)


    print(d)

main2()
```

3. Trace the following code, showing all interactions with the stack and the heap.

```
def g1(lst: List[int]):
    if len(lst) < 2:
        lst.append(17)
    else:
        print('Long!')

def g2(my_list: List[int]) -> int:
    g1(my_list)
    g1(my_list)
    return len(my_list)
    g1(my_list)

def main3():
    your_list = [6]
    z = g2(your_list)
    print(z)

    new_list = []
    g2(new_list)
    print(new_list)

main3()
```

In-class Coding:

4. Write a function `str_dictionary` which takes a string `s` as a parameter. `s` will only contain lowercase letters—although it could be empty. Your function should return a dictionary, using the characters of `s` as keys, with each having a value which is the number of times that the character appears in the string `s`.

   For example,

   - `str_dictionary('hello')` returns `{'h': 1, 'e': 1, 'l': 2, 'o': 1}`
   - `str_dictionary('')` returns `{}`
   - `str_dictionary('aababcabcdabcde')` returns `{'a': 5, 'b': 4, 'c': 3, 'd': 2, 'e': 1}`

```
str_dictionary(s: str) -> Dict[str, int]:
    # write your code here
```

5. On the next page, write a class `CokeBottle` which models a simple plastic bottle of Coke. A bottle should have two attributes, a boolean `open` which is `True` when the bottle is open and `False` otherwise (users can open or close the bottle as they choose), and an `amount` (an integer, in ounces), which is set to 20 initially. The bottle starts out closed.

   There should be two methods:

   - `drink(n: int)` which will drink n ounces – you can assume that n will always be positive. If you drink more than the bottle contains, just set `amount` to 0. However, you cannot drink from a closed bottle or from an empty bottle – if you try, a message should be printed to the user.
   - `toggle_cap()` which changes the cap from open to closed or vice-versa. There are no restrictions on when you can change the status of the cap.

   Consider the following two transcripts of a session in the console:

```
>>> a = CokeBottle()
>>> a.drink(4)
You cannot drink from a closed bottle

>>> a.toggle_cap()
>>> a.drink(4)
>>> a.toggle_cap()
>>> a.amount

16
>>> a.open
False
```

```
>>> b = CokeBottle()
>>> b.toggle_cap()
>>> b.drink(12)
>>> b.toggle_cap()
>>> b.amount
8

>>> b.toggle_cap()
>>> b.drink(11)
>>> b.drink(3)
You cannot drink from an empty bottle.

>>> b.amount
0
```

```
class CokeBottle:

    def __init__(self):
        # add code here




    def drink(self, n: int):
        # add code here




    def toggle_cap(self):
        # add code here
```

Take-Home

6. Write a function `in_order` which takes in a string `s` and returns `True` if the characters in the string occurs in alphabetical order and `False` otherwise. You can assume the string will only have lowercase letters and not any spaces, digits, or other special characters. Repeated letters are considered in order, and the empty string is considered to be in order.

   For example:

   - `in_order('cot')` should return `True`
   - `in_order('boot')` should return `True`
   - `in_order('cook')` should return `False` (since 'k' comes before 'o')
   - `in_order('')` should return `True`
   - `in_order('cat')` should return `False`

7. Write a function `word_hist` that takes a sentence `s` as a parameter and returns a dictionary where the keys are words and the values are the number of times each word was found in the sentence. All words should be treated as lowercase–that is, *you* might have to `.lower()` them. Also, the word `"lemurs"` is awesome, so it should be counted 3 times whenever seen. There will be no punctuation or other non-space, non-alphabetic characters in `s`.

   For example, supposing that

   `s1 = "She sells what she sells and what she sells is lemurs"`

   `s2 = "Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo"`[1]

   then `word_hist(s1)` yields

[1] `https://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo`

`{'she': 3, 'sells': 3, 'what': 2, 'and': 1, 'is': 1, 'lemurs': 3}`,

and `word_hist(s2)` yields

`{'buffalo': 8}`.

Hint: Use the `split()` function on a string to turn it into a list of words.

8. Write a function `pref_dict` which will take parameters `lst`, a list of strings, and `n`, a non-negative integer. The value of `n` will be a prefix length.The function will return a dictionary, keyed on the prefix, and with value the number of strings in the list which start with that prefix.

   The `n` will always be less than or equal to the length of the shortest word.

   For example:

   - `pref_dict(['car', 'cat', 'hat', 'mouse', 'more'], 2)` will return `{'ca': 2, 'ha': 1, 'mo': 2}`
   - `pref_dict(['threw', 'at', 'attention', 'through', 'attempt', 'thrush'], 2)` will return `{'th': 3, 'at': 3}`
   - `pref_dict(['car', 'cat', 'hat', 'mouse', 'more'], 0)` will return `{'': 5}`

   [For the two classes below, you are not required to include `__repr__()` or `__str__()` methods, but are welcome to if you find them useful.]

9. Create an `Animal` class. Each animal will have two attributes, a string `color` (such as `'red'` or —'blue'—) and a positive integer `size`. These will be assigned upon creation of each object in the class. You will then include two methods:

   - `grow`, which takes in an integer parameter (which could be negative). This will increase the size of the animal by that amount (or decrease it, if the parameter was negative). However, no animal can ever have a size smaller than 1.
   - `is_friendly` which takes no outside parameters, but returns a boolean deciding if the animal is friendly by the following rule: Small animals (those with a size smaller than 10) are friendly. Big animals (with a size greater than 24) are never friendly. Medium animals (with a size between 10 and 24, inclusive) are friendly only if they are yellow or blue. Finally, however, red animals are never friendly, no matter their size. (Note that animals can have other colors besides red, blue, and yellow.) [The details of this might look familiar!]

   To check, consider the following console transcripts:

```
>>> a = Animal('blue', 12)
>>> a.is_friendly()
True

>>> a.grow(10)
>>> a.is_friendly()
True

>>> a.grow(20)
>>> a.is_friendly()
False



>>> b = Animal('red', 20)
>>> b.is_friendly()
False
```

```
>>> b.grow(7)
>>> b.size
27

>>> b.grow(-20)
>>> b.size
7

>>> b.is_friendly()
False

>>> b.grow(-10)
>>> b.size
1
```

10. Create a `Character` class, which represents a character in the world's most boring video game. When instantiated, a Character object has the following attributes:

    - `energy` – An energy level, always a positive integer, set initially as 20 units.
    - `dist` – A total distance travelled, also always an integer, and initially set to 0.

    A Character can do four things, each of which should be a method:

    - `walk(n: int)` – the character moves a distance of n, which takes one unit of energy for each unit moved. If the character runs out of energy, they stop moving (that is, suppose a character has 4 units of energy. If you call `move(7)`, the character's energy should drop to 0 and they should only move a total of 4 units.
    - `run(n: int)` – like walk, except that you use three times as much energy. (If you end up with an energy of 2 or less, notice that you cannot run anymore)
    - `eat(n: int)` – increases energy by n units
    - `rest()` – increase energy by 5 units

    Consider the following console transcript:

```
>>> a = Character()
>>> a.walk(3)
>>> a.energy
17

>>> a.dist
3

>>> a.run(4)
>>> a.energy
5

>>> a.dist
7

>>> a.rest()
>>> a.energy
10

>>> a.run(5)
```

```
>>> a.run(3)
>>> a.dist
10

>>> a.energy
1

>>> a.rest()
>>> a.rest()
>>> a.energy
11

>>> a.eat(3)
>>> a.energy
14

>>> a.dist
10

>>> a.walk(9)
>>> a.dist
19

>>> a.energy
5

>>> a.walk(8)
>>> a.dist
24

>>> a.energy
0
```