CSCI 150 (Spring 2016): Exam 3 Prep

Instructor: Brent Yorgey

In preparing your solutions to the exam, you are **allowed to use any sources** including your textbook, each other, previous homeworks and labs, or any sources on the Internet. The only restriction is that I will not help you with exam questions, although you are welcome to ask me general questions that do not specifically relate to a problem on the exam. I am also happy to answer clarifying questions about exam problems.

On exam day, you will have 50 minutes to complete the exam. During the exam, **you will not be allowed to use your notes, textbook, phone or computer, though you may use a calculator.** You will be given a fresh copy of the exam and must write your solutions from memory.

1 Practice reading problems

The first three problems on the exam will be code reading problems, where you are given some Python code and asked what it does. One problem will involve recursion, one will involve dictionaries, and one will involve classes and objects. The problems below are **for practice**; the actual problems on the exam will be **different** than the problems below.

Remember that you can only receive partial credit if you show your work.

1. (10 points) Consider the following function definition.

```
def zzz(a):
    if a == []:
        return ""
    else:
        return a[0] + ":" + zzz(a[1:])
```

Given the definition

```
thelist = ["the", "cat", "in", "the", "hat"],
```

what is the output of `zzz(thelist)`?

2. (10 points) Consider the following function definition.

```
def yyy(a):
    d = {}
    i = 0
    for s in a:
        if s not in d:
            d[s] = []
            d[s].append(len(s))

        d[s].append(i)
        i += 1
    return d
```

Given the definition

```
thelist = ["the","cat","in","the","hat"],
```

What is the output of `yyy(thelist)`?

3. (10 points) Consider the following Python definitions.

```
class Heffalump:
    def __init__(self, name):
        self.name = name
        self.count = 1

    def __repr__(self):
        return str(self.name) + " (" + str(self.count) + ")"

    def heff(self):
        self.count += 1
        if (self.count % 3 == 0):
            self.count += 1

    def lump(self, other):
        if (self.count > other.count):
            for i in range(self.count - other.count):
                print "Heff!"

def main():
    ned = Heffalump("Ned")
    fred = Heffalump("Fred")
    fred.heff()
    ned.lump(fred)
    ned.heff()
    ned.heff()
    ned.lump(fred)
    print ned
```

What is printed by `main()`?

2 Exam questions

The remaining questions will appear on the exam **exactly** as they appear below (subject to corrections). Again, you may use any resources in preparing your solutions, but during the exam you must write your solutions from memory.

Once you have prepared your solutions, I **highly recommend** putting them aside for a while and then testing yourself by trying to write them out from memory!

4. (15 points) Write a **recursive** function `maxlist` which returns the maximum element in a list, or `None` if the list is empty.

    - `maxlist([1,2,3,2,5,3])` $\to$ `5`
    - `maxlist(["hi", "there", "world"])` $\to$ `"world"`
    - `maxlist([])` $\to$ `None`

    ```
    def maxlist(the_list):
    ```

5. (15 points) Write a **recursive** function `power` which performs exponentiation. That is, `power(`*base*, *exp*`)` should return $base^{exp}$. For example:

   - `power(3,4)` $\rightarrow$ `81`
   - `power(17,0)` $\rightarrow$ `1`
   - `power(2,20)` $\rightarrow$ `1048576`

   You may assume that `exp` is a nonnegative integer. Recall that $x^0 = 1$ for any x (even $x = 0$).

   For 5 points **extra credit**, write your function in such a way that it can successfully compute `power(2,2000)` without exceeding the maximum recursion depth.

```
def power(base, exp):
```

6. (20 points) Write a class called `Fountain`. Objects of this class will have three component variables: the capacity of the fountain, the amount of water currently in the fountain (both measured in gallons), and whether the fountain is currently running or not.

   Fountains start out empty, in the "off" state, with a given capacity. For example,

   ```
   fountain = Fountain(200)
   ```

   should create an empty fountain object in the "off" state with a capacity of 200 gallons of water.

   Besides `__init__`, you will need three additional methods: `fill(gallons)`, `toggle()`, and `look()`.

   - When `fill(gallons)` is called:
     - If `gallons` is positive, that many gallons of water are added to the fountain. Note that the fountain cannot hold more than its capacity: if the total contents of the fountain exceeds its capacity, the excess gallons overflow and the fountain contains exactly its capacity.
     - If `gallons` is negative, that many gallons of water are drained from the fountain, except that the contents of the fountain cannot become negative. If the fountain becomes empty then it should automatically turn off.
   - When `toggle()` is called:
     - If the fountain is on, it will become off.
     - If the fountain is off but it is empty, it cannot turn on. Instead, an error message should be printed.
     - If the fountain is off and contains some water, it will turn on.
   - When `look()` is called, it prints a string explaining what the fountain looks like. The string should say whether the fountain is on or off, and how much water it contains.

For example, here is a sample session at the Python prompt. Your implementation does not have to produce exactly the same messages, as long as they contain the same information.

```
>>> f = Fountain(200)
>>> f.look()
The fountain is off. It contains 0 gallons of water.
>>> f.toggle()
Cannot start a dry fountain!
>>> f.look()
The fountain is off. It contains 0 gallons of water.
>>> f.fill(120)
>>> f.look()
The fountain is off. It contains 120 gallons of water.
>>> f.toggle()
>>> f.look()
Sploosh! The fountain is on. It contains 120 gallons of water.
>>> f.fill(397)
>>> f.look()
Sploosh! The fountain is on. It contains 200 gallons of water.
>>> f.toggle()
>>> f.look()
The fountain is off. It contains 200 gallons of water.
>>> f.toggle()
>>> f.look()
Sploosh! The fountain is on. It contains 200 gallons of water.
>>> f.fill(-90)
>>> f.look()
Sploosh! The fountain is on. It contains 110 gallons of water.
>>> f.fill(-200)
>>> f.look()
The fountain is off. It contains 0 gallons of water.
```

Please implement `Fountain` in the space below. Do **not** write a `main` method, just the `Fountain` class!

```
class Fountain:
```

```
# Extra space for Fountain class, if needed
```

7. (20 points) Your eccentric half-great-uncle-thrice-removed has left you a lot of money, and you decide to spend some of it on a trip. The only problem is that you cannot decide where to go. So, of course, you decide to write a program to plan your trip for you.

   Your function should take as input

   - A dictionary whose keys are strings representing the names of cities, and whose values are lists of strings, representing the destination cities you can reach by a plane flight from each key city, and
   - An integer `n` specifying the number of plane flights you want to take.

   Your function should return a randomly selected list of `n+1` cities, where the starting city is `"Little Rock"` and each subsequent city can be reached from the previous one by a plane flight. (You do not need to end up back in Little Rock; also, it's OK to visit the same city more than once.)

   For example, given

```
flight_map = \
  { "Little Rock" : ["Baltimore", "Chicago", "New York"]
  , "New York" : ["Paris", "Copenhagen", "London"]
  , "Chicago" : ["Los Angeles", "Toronto"]
  , "Toronto" : ["Chicago", "New York"]
  , "Paris" : ["Copenhagen", "New York"]
  , "Copenhagen" : ["Reykjavik", "Paris"]
  , "Reykjavik" : ["Copenhagen"]
  , "Los Angeles" : ["Tokyo", "Chicago"]
  , "London" : ["New York", "Paris", "Copenhagen", "Tokyo"]
  , "Tokyo" : ["London", "Los Angeles"]
  , "Baltimore" : ["Little Rock", "New York", "Chicago"]
  }
```

   a call to `plan_trip(flight_map, 8)` might return

```
[ 'Little Rock', 'New York', 'Paris', 'New York',
  'London', 'Tokyo', 'London', 'Copenhagen', 'Reykjavik']
```

```
import random

def plan_trip(flights, n):
```