CSCI 150: Exam 3 Practice Problems (Solutions)

November 14, 2018

Recursion

1. Below is a function that calculates the exponentiation function x^y. Assume that y is an integer greater than or equal to zero.

```
def power(x: float, y: int):
    result: float = 1
    for i in range(y):
        result *= x
    return result
```

For example:

- `power(2, 3)` returns `8` since $2^3 = 8$.
- `power(3, 0)` returns `1` since $3^0 = 1$.

Rewrite this function using recursion. Clearly comment the base case and recursive case. (*Hint*: you can make use the fact that $x^n = x \cdot x^{n-1}$. (There is also a more efficient way to do it—can you find it?))

```
def power_rec(x: float, y: int):
    if y == 0:
        # x^0 = 1 no matter what x is
        return 1
    else:
        # Otherwise return x * x^(y-1)
        return x * power_rec(x, y-1)
```

2. Write a function **using recursion** that takes a string as a parameter and returns the length of the string. You should **not** use a `while` loop or `for` loop in this function, nor the built in `len` function.

```
def length(s: str) -> int:
    if s == '':    # length of '' is 0
        return 0
    else:
        return 1 + length(s[1:])
```

3. Ada needs to find if all the elements of a list are divisible by either 3 or 5. She decides to use recursion to solve this problem, and writes the following function.

```
def div_three_five(t):
    candiv = div_three_five(t[1:])
    if (candiv or t[0] % 3 == 0 and t[0] % 5 == 0):
        return True
    else:
        return False
```

Is this function correct? Why or why not? How would you fix it?

Solution: It is not correct, for two reasons. First, the condition is not correct. The function should return `True` if all the rest of the elements in the list are divisible by 3 or 5, **and** the first element is divisible by 3 **or** 5. Second, there is no base case: as written, the function *always* calls itself recursively until it runs out of elements in the list and crashes. A corrected version might be as follows:

```
def div_three_five(t):
    if (len(t) == 0):
        return True
    else:
        candiv = div_three_five(t[1:])
        if (candiv and (t[0] % 3 == 0 or t[0] % 5 == 0)):
            return True
        else:
            return False
```

Note that the final `if-else` could also simply be written as a `return`, that is,

```
        return (candiv and (t[0] % 3 == 0 or t[0] % 5 == 0))
```

Classes and Objects

4. Write a class called `StoneSoup`. Objects of this class will have two component variables: a list of ingredients currently in the soup, and a threshold for how many ingredients it takes to make the soup taste good. The list of ingredients should start out empty; the `__init__` method should take just the threshold as a parameter. For example,

   ```
   soup = StoneSoup(6)
   ```

   will make a soup that currently has no ingredients and will be tasty when there are at least 6 ingredients.

   You will need two additional methods, `add(ingredient)` and `tasty()`. `add` will put a new ingredient into the list *if it is not already there*. An ingredient can only be added once. `tasty` will return `True` if the number of ingredients in the soup is greater than or equal to the threshold, and `False` otherwise.

   ```
   class StoneSoup:
       def __init__(self, threshold: int):
           self.threshold   = threshold
           self.ingredients = []

       def add(self, ingredient: str):
           if ingredient not in self.ingredients:
               self.ingredients.append(ingredient)

       def tasty(self) -> bool:
           return len(self.ingredients) >= self.threshold
   ```

5. Write a class called `RV`. `RV` objects will have four component variables representing

   - the gas mileage (measured in miles per gallon),
   - the size of the gas tank,
   - the current amount of gas in the tank, and
   - the current mileage,

   all of which will be integers. The `__init__` method should take two integer parameters, one that is the size of the gas tank and the other that is the miles-per-gallon value.

   You will need two additional methods, `cruise(distance)` and `fill(gallons)`. `cruise` will increase the mileage on the RV and decrease the gas according to the miles-per-gallon. Remember, an RV cannot drive when it is out of gas. `fill` will add gas to the tank. Remember that the amount of gas in the tank cannot exceed the size of the tank.

```
class RV:

    def __init__(self, tank_size: int, mpg: int):
        self.mpg = mpg
        self.tank_size = tank_size
        self.gas_amount = 0
        self.mileage = 0

    def cruise(self, distance: float):
        gas_used = min(self.gas_amount, distance/self.mpg)
        distance = min(distance, gas_used * self.mpg)
        self.mileage += distance
        self.gas_amount -= gas_used

    def fill(self, gallons: float):
        if gallons + self.gas_amount > self.tank_size:
            self.gas_amount = self.tank_size
        else:
            self.gas_amount += gallons
```

Reading

6. Trace the execution of the following Python program using the given template. Describe what is happening to `mylist` using English.

```
def bar(b, i, j):
    t = b[i]
    b[i] = b[j]
    b[j] = t

def oogie(a):
    p = 0
    while (p < len(a)):
        if p == 0 or a[p - 1] <= a[p]:
            p += 1
        else:
             bar(a, p, p - 1)
             p -= 1
             print(a)

def main():
    mylist = [2, 13, 5, 8]
    oogie(mylist)

main()
```

Calling `oogie` puts `mylist` into order from smallest to largest.

7. Trace the execution of the following Python program using the given template. Describe in English what the function `snerf` does.

```
def snerf(lst):
    if len(lst) == 1:
        return lst[0]
    else:
        m = snerf(lst[1:])
        if lst[0] > m:
            return lst[0]
        else:
            return m

print(snerf([1,5,2]))
```

This function finds and returns the minimum value in `lst`.

Dictionaries

8. Write a function `prev_up` that takes an alphabetic string as a parameter and returns a dictionary, where the keys are letters of the alphabet and the values are the number of times each letter was immediately preceded by the previous letter in the alphabet (where `"s"` is preceded by `"r"`, etc., and we count `"a"` as being preceded by `"z"`) if this number is 1 or more. For example:

   - `prev_up("defenders")` → `{'s': 1, 'f': 1, 'e': 2}`
   - `prev_up("misunderstanders")` → `{'t': 1, 's': 2, 'e': 2}`
   - `prev_up("understudy")` → `{'t': 1, 's': 1, 'e': 1, 'u': 1}`

```
import string

# You could also just define alpha = "abcdef...."
alpha: str = string.ascii_lowercase

def prev_up(s: str) -> Dict[str, int]:
    counts: Dict[str, int] = {}

    for i in range(len(s) - 1):
        cur  = s[i+1]
        prev = s[i]
        if (alpha.find(prev) + 1) % 26 == alpha.find(cur):
            if cur not in counts:
                counts[cur] = 0

            counts[cur] += 1

    return counts
```

9. Write a function that takes a sentence 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. Also, the word `"lemurs"` is awesome, so it should be counted 3 times whenever seen.

   For example, supposing that[1]

   ```
   s1 = "She sells what she sells and what she sells is lemurs"
   s2 = "Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo"
   ```

   then `word_hist(s1)` yields

   `{'she':3, 'what':2, 'sells':3', '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.

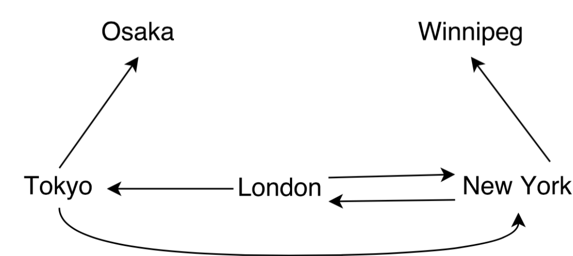
   ```
   def word_hist(s: str) -> Dict[str, int]:
       words: List[str] = s.lower().split()

       counts: Dict[str, int] = {}

       for word in words:
           if word not in counts:
               counts[word] = 0

           if word == 'lemurs':
               counts[word] += 3
           else:
               counts[word] += 1

       return counts
   ```

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

10. Consider using a dictionary to represent a network of flights. The keys of the dictionary are the names of cities, and the values are lists of cities connected to the key city by a single plane flight. For example, the dictionary

```
flights =\
  { "London"   : ["Tokyo", "New York"]\
  , "Tokyo"    : ["Osaka", "New York"]\
  , "New York" : ["London", "Winnipeg"]\
  }
```

represents a network that looks like this:



where each arrow represents a plane flight. In this (admittedly unrealistic) scenario, there are no flights leaving Osaka or Winnipeg, so it is possible to get "stuck" in those cities.

(See next page.)

Write a function `stuck` which takes a dictionary of flights, as described on the previous page, and returns a list of all the cities in which it is possible to “get stuck”, that is, cities for which there are no outgoing flights. For example, given the above example, `stuck(flights)` should return `["Osaka", "Winnipeg"]` (or it could return the items in the other order; the order of items in the output list does not matter).

```
def stuck(d: Dict[str, List[str]]) -> List[str]:
    stuck_cities: List[str] = []

    for city in d:
        for dest in d[city]:
            if dest not in d and dest not in stuck_cities:
                stuck_cities.append(dest)

    return stuck_cities
```