CS 30 Discussion

Week 10

Welcome back to CS30 Discussion

- HW #5 grades have been posted.
- HW #6 grades will be posted next week.
- **Final exam on next Wednesday (12.16).**
- Full solutions to the practice problems we discuss today will be posted on CCLE after the discussion.
- I'll move my last office hour to next Tuesday 4-6pm.

Final Exam Review

Dictionary

Dictionaries are used to store data values in **key:value** pairs.

A dictionary is a collection which is **unordered**, **changeable** and does **not** allow **duplicates**.

Dictionaries are written with curly brackets, and have keys and values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

```
def count(l):
    counter = {}
    for x in l:
        if x not in counter:
            counter[x] = 1
        else:
            counter[x] += 1
    return counter
```

For Loops

for loop: executes some block of code once for each element of a given list

Loop recipe:

1. Introduce an accumulator variable to store the value to return.

2. Initialize the accumulator to the value that should be returned if the list is empty.

3. How to write the body of a loop of the form "for x in l" ?
   Consider the case where the loop is on its last iteration, so x is the last element of the list.
   Assume that the accumulator contains the right answer for the prefix of the list up to x.
   Update the accumulator to incorporate x into the result.

Example

```
def sumList(l):
    result = 0
    for x in l:
        result = result + x
    return result
```

```
def sumListRec(l):
    if l == []:
        return 0
    else:
        head = l[0]
        tail = l[1:]
        tailRes = sumListRec(tail)
        return head + tailRes
```

While loops

while loop: executes some block of code as long as the loop condition is true (in other words, until the loop condition becomes false)

Loop recipe:

1. Introduce an accumulator variable to store the value to return.

2. Initialize the accumulator to the value that should be returned if the loop never runs

3. Write a loop condition

4. In the body of the loop accumulate results

5. Update the loop condition variable

Example

```
# 1. Return a list of odd numbers
in a given range start, end
```

```
def oddList(start, end):
    result = []
    i = start
    while i < end:
        if i % 2 != 0:
            result = result + [i]
        i+=1
    return result
```

```
def contains_ie(string):
  found = False
  i = 0
  while not found and i < len(string):
       if string[i] == 'i' and string[i+1] == 'e':
            found =  True
       i += 1
  return found
```

Q1. There's a bug in this code; on certain inputs it will crash! Identify the bug and give an example of some inputs that will reveal the buggy behavior.

Correct code

```
def contains_ie(string):
  found = False
  i = 0
  while not found and i < len(string)-1:
       if string[i] == 'i' and string[i+1] == 'e':
            found =  True
       i += 1
  return found
```

Q2. What value does "i" have just before the function returns?

```
contains_ie("bierhall")
contains_ie("wonderbubbles")
contains_ie("weirdo")
```

Tips and Tricks

1. Use this when you only need to touch every element in a list or string once.

```
for x in my_list:
  doSomething(x)
```

2. Use this when you need to touch every element in a list or string and also need the indexes:

```
for i in range(len(my_list)):
  doSomething(my_list[i])
```

3. Use it when you need to loop an unknown number of times, until a certain condition becomes false,

```
i = 0
while(i < n):
  doSomething with n, i etc
  i+=1
```

Map review

- **map** is a function takes two arguments:
    the first argument is the **function `f` to use to transform each list element**
    the second argument is **the list `l` to transform**
- **map** always returns a transformed list `l'`
- `len(l) = len(l'),` order of elements in `l` is the same as `l'` but values can be different
- Function `f` can be a lambda or a defined function, `f` should always return a value

Filter Review

- **filter** is a function takes two arguments:
  - the first argument is the **function `f` to use to filter a list element**
  - the second argument is **the list `l` to filter**
- **filter** always returns a list l' which is a subset of the list `l`
- `len(l') <= len(l),` order of elements in the smaller l' is preserved
- **The return value of a function `f` is always boolean (True/False)**
- `list(filter(f, l)) returns a list of all elements e of l such that f(e) = True`

Lambda review

- To avoid writing small function definitions, we can use lambda
- Lambda is a keyword that means that we're defining an **anonymous function**.
- Example lambda:
  - `(lambda x: x**2 + 2*x - 5)(5)= 30`
  - `(lambda x: x if x > 0 else 0)(-1) = 0`

Reduce Review

- **reduce** is a function takes two arguments (three in a general usage):
  - the first argument is the **function `f` that takes two arguments where the first argument is the result of reducing the list so far and the second argument is the element of the list**
  - the second argument is **the list `l` to reduce**
  - *the third argument is the **initial value**
- `reduce(f, [x1,x2,x3]) performs the computation  f(f(x1, x2), x3)`

Recursion

Remember that recursive functions are just functions that call themselves.

You first **call yourself recursively on a slightly smaller version** of the argument, before doing anything else.

Then the key is that you get to **assume that the recursive call does the right thing**, and now your job is to figure out **how to use that result to produce the overall result** that is desired.

General Recipe

the general 'recipe' of defining a recursive function:

- 1 or more *base cases*
    - handle the "smallest" arguments
- 1 or more *recursive cases*
    - use the results of one or more *recursive calls* with "smaller" arguments
    - the magic of recursion:
        - you get to *assume* that these recursive calls do the right thing!

Recursion Recipe for list based problems

- You always need a base case: small input with obvious answer
- break into the first item (head) and the rest (the tail)
  - `head = l[0]`
  - `tail = l[1:]`
- assume your function gives you the right answer for the tail
  - `tailResult = recursiveCall(tail)`

If-elif-else Syntax

```
if expression1:
    statement(s)
elif expression2:
    statement(s)
elif expression3:
    statement(s)
else:
    statement(s)
```

Remember: if-elif-else implies that each "case" is *exclusive*.

What would the following print out if x were 5?

```
if x < 10:
    return "I'm under ten!"
elif x == 5:
    return "I'm five!"
else:
    return "I give up"
```

Function review

Syntax:

```
def functionName (arguments):
        statement(s)
        return something
```

- **Function arguments are just variables.**
- **They take different data values**


GOOD NEWS EVERYONE
PROGRAMMING IS NOT HARD
imgflip.com

Instructor Evaluations due soon.

WorkSheet

Please work together on the problem set.

- Q3: crosswordFind_loops
- Every question in Part 2