Announcements & Logistics

- **Lab 6** due Wed/Thurs at 10 pm
  - Uses dictionaries, plotting, CSV files
- **HW 6** will be out today, due Mon at 10pm
- Lab 7, 8, and 9 are **partner labs**
  - Fill out google form sent by Lida by **noon tomorrow (Thursday)!**
  - Pair programming is an important skill as well as a vehicle for learning
- Pick up your **graded midterm exam** at the end of class
  - Will use last few mins of lecture to discuss the midterm

Do You Have Any Questions?
Last Time

• Worked through an example involving CSVs, dictionaries, and sets
• Discussed plotting with matplotlib
  ‣ Python is pretty useful for data processing and visualization!
Today’s Plan

Intro To Recursion

• Discuss what we mean by the term recursion
• Practice translating recursive ideas into recursive programs
• Examining the relationship between recursive and iterative programs
  • That is, how do recursive ideas relate to the iterative ideas (for loops, while loops) we’ve covered so far?
Where are We Going?

• First half of CS134: learned some fundamental programming concepts
  • Functions, conditionals, loops, data types
  • Built-in data structures and operations

• Looking ahead to the second half: more emphasis on algorithmic and conceptual topics: more "computational thinking"
  • **Recursion** (~1 week)
  • Defining our own **data types** using **classes and objects** (~2 weeks)
    • Object oriented programming topics
  • Continue developing our intuition regarding efficient vs inefficient code
Why Learn About Recursion?

• Recursion is an important problem solving paradigm
  • An alternative to **iteration** for repeatedly performing a task
  • Process that lets us "divide, conquer, combine"
  • Useful to build and maintain data structures (like trees and lists)
• Provides a different lens to view the world
  • If you love procrastination — recursion is just the thing for you!
So What Is Recursion?

• An alternative to iteration (loops) for repetition

• General problem solving idea:
  • Break the problem down to a smaller version of itself
  • Keep doing this until the problem is so small, the answer is straightforward

• Let's take an example of this approach

• **Example.** Write a function `count_down(n)` that prints integers `n, n−1, \ldots, 1` (one per line)

• How would we solve this using a loop?
Iterative: count_down(n)

- **Example.** Write a function `count_down(n)` that prints integers `n, n-1, ..., 1` (one per line)

- How would we solve this using a loop?

```python
def count_down_iterative(n):
    '''Solution using loops'''
    for i in range(n):
        print(n - i)
```
Iterative: count_down(n)

- **Example.** Write a function `count_down(n)` that prints integers `n, n-1, ..., 1` (one per line)
- Now let's use recursion to do the same thing
- Recursion lets you solve this **without any loop**
  - Just using conditionals and functions

```python
def count_down_iterative(n):
    '''Solution using loops'''
    for i in range(n):
        print(n - i)
```
Recursive: count_down(n)

• **Example.** Write a function `count_down(n)` that prints integers n, n−1, ..., 1 (one per line)

• Key ideas to use recursion:
  • What's the smallest version of the problem we can *immediately* solve?
  • For larger versions of the problem, is there a small step we can take that brings us closer to the smallest version of the problem?
Recursive: `count_down(n)`

- **Example.** Write a function `count_down(n)` that prints integers `n, n-1, ..., 1` (one per line)

- Key ideas to use recursion:
  - What's the smallest version of the problem we can *immediately* solve?
    - `count_down(1)` just prints `1` and nothing else
  - For larger versions of the problem, is there a small step we can take that brings us closer to the smallest version of the problem?
    - to solve `count_down(n)`, printing `n` is the first step
    - the rest of the problem is the smaller version of the same problem!
Understanding Recursive Functions

• **Example.** Write a function `count_down(n)` that prints integers \( n, n-1, \ldots, 1 \) (one per line)

• Recursive definition of countdown:
  • **Base case:** \( n = 1 \), print(\( n \))
  • **Recursive rule:** print(\( n \)), call `count_down(n-1)`

Perform one step

Reduce the problem (or make the problem “smaller”)

A function calling itself!
Recursive: count_down(n)

- **Example.** Write a function `count_down(n)` that prints integers 1, 2, ..., n (one per line)

```python
def count_down(n):
    '''Prints numbers from n down to 1'''
    if n == 1:  # Base case
        print(n)
    else:  # Recursive case: n > 1:
        print(n)
        count_down(n-1)
```

Recursion: A function calling itself!
Understanding Recursive Functions

• Recursive functions seem to be able to reproduce looping behavior without writing any loops at all

• To understand what happens behind the scenes when a function calls itself, let’s review what happens when a function calls another function

• Conceptually we understand function calls through the function frame model
Review: Function Frame Model
Review: Function Frame Model

- Consider a simple function `square`
- What happens when `square(5)` is invoked?

```python
def square(x):
    return x**x
```
Review:
Function Frame Model

```python
>>> square(5)
```

```
x
25
```

```
square(5)
```

```
x
5
```

```
return x * x
```
Summary:
Function Frame Model

• When we `return` from a function frame "control flow" goes back to where the function call was made

• Function frame (and the local variables inside it) **are destroyed after the return**

• If a function does not have an explicit return statement, it returns `None` after all statements in the body are executed

```python
>>> square(5) + 4
25
```
Review:
Function Frame Model

• How about functions that call other functions?

```python
def sum_square(a, b):
    return square(a) + square(b)
```

• What happens when we call `sum_square(5, 3)`?
def sum_square(a, b):
    return square(a) + square(b)

>>> sum_square(5, 3)
def sum_square(a, b):
    return square(a) + square(b)

>>> sum_square(5, 3)
25
```python
def sum_square(a, b):
    return square(a) + square(b)
```

```python
>>> sum_square(5, 3)
25
```

```
sum_square(5, 3)
```

```
| a | 5 | b | 3 |
```

```
return square(25) + square(b)
```

```
square(5)
```

```
x  5
```

```
return x * x
```

```
square(3)
```

```
x  3
```

```
return x * x
```
def sum_square(a, b):
    return square(a) + square(b)

>>> sum_square(5, 3)
	square(5)
	|
	|
	|
	|
	|
	|
	|
	square(3)
	|
	|
	|
	|
	|
	|
	n return x \times x

25 + 9
```python
def sum_square(a, b):
    return square(a) + square(b)

>>> sum_square(5, 3)
34
```

Function Frame Model to Understand count_down
def count_down(n):
    '''Prints ints from n down to 1'''
    if n == 1:
        print(n)
    else:
        print(n)
        count_down(n-1)

>>> val = count_down(5)
  5
  4
  3
  2
  1

>>> val = count_down(4)
  4
  3
  2
  1
```python
count_down(4)

n = 4

if n == 1:
    print(n)
else:
    print(n)
    count_down(n-1)

>>> count_down(4)
4
3
2
1
```

**Base case reached!**
```python
count_down(4)
```

```python
n 4
if n == 1:
    print(n)
else:
    print(n)
count_down(n-1)
```

```python
count_down(3)
```

```python
n 3
if n == 1:
    print(n)
else:
    print(n)
count_down(n-1)
```

```python
count_down(2)
```

```python
n 2
if n == 1:
    print(n)
else:
    print(n)
count_down(n-1)
```

```python
>>> count_down(4)
4
3
2
1
Base case reached!
```
```python
count_down(4)

n = 4

if n == 1:
    print(n)
else:
    print(n)
    count_down(n-1)

count_down(3)

n = 3

if n == 1:
    print(n)
else:
    print(n)
    count_down(n-1)

countDown(2)

n = 2

if n == 1:
    print(n)
else:
    print(n)
    count_down(n-1)

countDown(1)

n = 1

if n == 1:
    print(n)
else:
    print(n)
    count_down(n-1)

Base case reached!

>>> count_down(4)
4
3
2
1
```
```python
def count_down(n):
    if n == 1:
        print(n)
    else:
        print(n)
        count_down(n-1)

>>> count_down(4)
4
3
2
1
```
```python
>>> count_down(4)
4
3
2
1
Base case reached!
```
Recursive functions may seem like magic at first glance, but they follow from the principles that we’ve been building all semester.

It often takes several exposures to recursion before it “clicks”, so we’ll keep revisiting recursion in the coming lectures.

- Drawing pictures and practicing are two tools that can help.
- Our next lab is a partner lab so you can bounce your ideas off of a classmate and work through recursion stumbles.
Recursive Approach to Problem Solving

- A recursive approach to problem solving has two main parts:
  
  - **Base case(s).** When the problem is **so small**, we solve it directly, without having to reduce it any further (this is when we stop)
  
  - **Recursive step.** Does the following things:
    
    - Performs an action that contributes to the solution (take one step)
    
    - **Reduces** the problem to a smaller version of the same problem, and calls the function on this **smaller subproblem** (break the problem down into a slightly smaller problem + one step)

- The recursive step is a form of "wishful thinking": assume the unfolding of the **recursion** will take care of the smaller problem by eventually reducing it to the base case

- In CS136/256, this form of wishful thinking will be introduced more formally as the **inductive hypothesis**
Counting with Recursion

• Recall the function `count_appearances(elem, l)`
  • Returns the number of times `elem` appears in `l`
• What the iterative way to implement this?

```python
def count_occurrences(elem, l):
    count = 0
    for item in l:
        if item == elem:
            count = count + 1
    return count
```

Examples today are easily written iteratively, but we'll be looking at problems on Friday where that may not be the case!
Recursive: count_occurrences

• One of the keys to thinking recursively:
  • What's the smallest version of the problem we can immediately solve?
  • For larger versions of the problem, is there a small step we can take that brings us closer to the smallest version of the problem?

```python
def count_occurrences(elem, l):
    '''recursive version'''
    # base case (empty list)
    if len(l) == 0:
        return 0
    else:
        # is first item same as elem?
        # if so, we can add 1
        # else, we add zero
        # now we have a smaller problem:
        # count # occurrences in smaller list
```
Recursive: \texttt{count\_occurrences}

\begin{verbatim}
def count_occurrences(elem, l):
    '''recursive approach'''

    if len(l) == 0: # base case
        return 0

    else: # recursive case
        first = 1 if elem == l[0] else 0
        rest = count_occurrences(elem, l[1:])

        return first + rest
\end{verbatim}
Midterm Discussion
More Recursion: count_up
count_up(n)

• Write a recursive function that prints integers from 1 up to n

• Recursive definition of countUp:
  • Base case: n = 1, print(n)
  • Recursive rule: call count_up(n−1), print(n)

We swapped the order of recursing (calling count_up) and printing
countUp(n)

• Note that unlike count_down(n) we moved our print statement to be after the recursive function call

• By printing after the recursive call, the print statement gets executed “on the way back” from recursive calls

```python
def count_up(n):
    '''Prints out integers from 1 up to n'"
    if n == 1:
        print(n)
    else:
        count_up(n-1)
        print(n)

>>> count_up(5)
 1
 2
 3
 4
 5```
Function Frame Model to Understand count_up
```python
if n == 1:
    print(n)
else:
    count_up(n-1)
    print(n)
```

```
>>> count_up(4)
1
2
3
4
```

Base case reached!
Recursion GOTCHAs!
GOTCHA #1

• If the problem that you are solving recursively is not getting smaller, that is, you are not getting closer to the base case --- infinite recursion!

• Never reaches the base case

```python
def count_down_gotcha(n):
    '''Prints ints from 1 up to n'''
    if n == 1:  # Base case
        print(n)
    else:       # Recursive case
        print(n)
        count_down_gotcha(n)
```

Subproblem not getting smaller!
GOTCHA #2

- Missing base case/unreachable base case--- another way to cause infinite recursion!

def print_halves_gotcha(n):
    """Prints n, n/2, down to ... 1""
    if n > 0:
        print(n)
        return print_halves_gotcha(n/2)
"Maximum recursion depth exceeded"

- In practice, the infinite recursion examples will terminate when Python runs out of resources for creating function call frames, leads to a "maximum recursion depth exceeded" error message
Next Lectures

• Intro to **turtle** module and graphical recursion
• Comparing iterative and recursive programs