CS134: Recursion
Announcements & Logistics

• **Lab 6 due Wed/Thurs at 10 pm**
  • Uses dictionaries, plotting, CSV files

• **HW 6** will be available at noon today, due next Mon at 10pm

• **Lab 5** will be returned today
Last Time

• Worked through a simple example involving CSVs, dictionaries, and sets
• Discussed plotting with matplotlib
• Python is a powerful language for data processing and visualization
Where are We Going?

• First half of CS134: learned the **fundamentals of programming**
  • Functions, conditionals, loops, data types
  • Built-in data structures and methods, sorting, plotting
• 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
    • Building our own data types: **linked lists**
• **How does sorting really work?** what happens under the hood when Python is sorting?
• Continue developing our intuition regarding efficient vs inefficient code
Today’s Plan: Intro To Recursion

• What is recursion?
• Translating recursive ideas into recursive programs
• Examining the relation 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
Recursion In Art and Pop Culture

• You’re already familiar with the idea of recursion, whether you’ve referred to it by that name or not!
• The Droste effect was one of the first explicit uses of recursion in an advertising medium in 1904
• The cocoa tin shows an image of a woman holding a platter with a tin that has an image of the same woman holding platter with a tin that has an image of…
Recursion In Art and Pop Culture

- Computer scientists were of course writing nerdy poems about recursion long before it was cool (and before we had computers).

Great fleas have little fleas upon their backs to bite 'em,
And little fleas have lesser fleas, and so ad infinitum.
And the great fleas themselves, in turn, have greater fleas to go on;
While these again have greater still, and greater still, and so on.

— *Siphonaptera, A Budget Of Paradoxes*
by Augustus De Morgan (1874)
Why Learn About Recursion?

• Recursion is an important problem solving paradigm that can not only lead to **elegant** code, it can also be used to do cool things.

• By the end of lab next week, you'll be able to use recursion to draw these beautiful pictures.
So What Is Recursion?

- The easiest way to understand recursion is to first see examples of it.
- Let’s start by examining a familiar recursive definition in mathematics.
- The set of natural numbers can be defined as follows:
  - 0 is a natural number
  - If $n$ is a natural number, then $n+1$ is a natural number

- Building blocks of a recursive idea:
  1. **Base case(s):** 0 is a natural number
  2. **Recursive rule(s):** If $n$ is a natural number, then $n+1$ is a natural number
Exercise: Forming Base Case & Recursive Rules

• How would you define the concept of exponentiation $a^n$ as a base case and a recursive rule (assuming $n \geq 0$)

• A recursive definition:
  • Base case:
  • Recursive rule:
Exercise: Forming Base Case & Recursive Rules

• How would you define the concept of exponentiation $a^n$ as a base case and a recursive rule (assuming $n \geq 0$)

• A recursive definition:
  • **Base case:** $a^0 = 1$
  • **Recursive rule:** $a^n = a \ast a^{n-1}$
Exercise: Forming Base Case & Recursive Rules

• Similarly, how would you define the concept of factorial \( n! \) as a base case and a recursive rule (assuming \( n \geq 0 \))

• A recursive definition:
  • **Base case:** \( 0! = 1 \)
  • **Recursive rule:** \( n! = n \times (n-1)! \)
Exercise: Forming Base Case & Recursive Rules

- Let's examine a more complicated series known as the Fibonacci sequence.
- The Fibonacci sequence is a series of numbers that starts with $0$ and $1$, and where each successive number is the sum of the two preceding ones:
  \[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots\]

- A recursive definition:
  - **Base cases:** $F_0 = 0$ and $F_1 = 1$
  - **Recursive rule:** $F_n = F_{n-1} + F_{n-2}$
Translating Recursive Ideas To Programs

• The beauty of recursion is that once you’ve written down your recursive idea, the programming part is (relatively) easy

• Ideally, you spend more time with pen and paper and front-load all your thinking into coming up with an appropriate base case and recursive rule

• Once you have these two ingredients, the implementation of recursive programs is fairly formulaic

```python
def recursiveProgram(inputs):
    # if inputs correspond to base case apply base case rules
    # else apply recursive rule
```
Translating Recursive Ideas To Programs

• Recursive definition for \( a^n \):
  • Base case: \( a^0 = 1 \)
  • Recursive rule: \( a^n = a \times a^{n-1} \)

```python
def power(a, n):
    """
    Returns \( a^n \). Assumes \( n \geq 0 \).
    """
    if n == 0:
        return 1
    else:
        return a * power(a, n-1)
```

```python
print(power(5, 0))
print(power(5, 4))
```

1
625
Translating Recursive Ideas To Programs

• Recursive definition for Fibonacci:
  • **Base cases:** $F_0 = 0, F_1 = 1$
  • **Recursion:** $F_n = F_{n-1} + F_{n-2}$

```python
def fibonacci(n):
    """
    Returns nth Fibonacci number
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(5))
print(fibonacci(6))
print(fibonacci(7))
```

5
8
13
Recursive Functions

• We have seen many examples of functions calling other functions
• A **recursive function** is a function that **calls itself**
• Recursive functions consist of one or more **base cases** and a set of **recursive rules** that successively **simplify** (or reduce) the problem **until we reach one of the base cases**
• Recursive rules **must** eventually take you to one of the base cases, else we end up with the recursive equivalent of an infinite loop
• We will compare recursive implementations to iterative implementations soon, but for now let’s take a deeper look into how recursion works
Infinite Recursion

- Recursive definition for $a^n$:
  - **Base case**: $a^0 = 1$
  - **Recursive rule**: $a^n = a \times a^{n-1}$

This causes a **RecursionError**: maximum depth exceeded in comparison — notice we are no longer simplifying the problem in our recursive rule.

- What does this error mean?

- So far, we've simply believed in the magic of recursion but let's take a closer look at what goes on in recursive function calls.

```python
def infinitePower(a, n):
    """
    Returns a\^n
    """
    if n == 0:
        return 1
    else:
        return a * infinitePower(a, n)

print(infinitePower(5, 4))
```
Understanding Recursive Functions

• Let’s review a simple recursive function that gives us some intermediate feedback through print statements.
• Write a recursive function that prints integers from n down to 1
• Recursive definition of countdown:
  • Base case: n = 0, do nothing
  • Recursive rule: print(n), call countDown(n–1)
Understanding Recursive Functions

- Recursive definition of countdown:
  - **Base case:** \( n = 0 \), do nothing
  - **Recursive rule:** \( \text{print}(n), \text{call=countDown}(n-1) \)

```python
def countDown(n):
    '''Prints numbers from n down to 1'''
    if n < 1:  # Base case
        pass  # Do nothing
    else:  # Recursive case: n >= 1:
        print(n)
        countDown(n-1)

countDown(5)
```

5
4
3
2
1
Side Note: Implicit Base Case

- It is possible to simplify our function by omitting the base case
- The following two versions are equivalent
- Version 2 is preferred in general
- Version 1 shows the base case explicitly (and uses pass, which is a little weird), but for now we'll focus on that version
- Note: Bad things happen if you forget the base case.... (we'll see)

```python
def countDown(n):
    '''Version 1'''
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

def countDown(n):
    '''Version 2'''
    if n > 0:
        print(n)
        countDown(n-1)
```
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

>>> square(5)

```
x = 5

return x * x

25
```
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
def square(x):
    return x**2

>>> square(5) + 4
25
```

Return value replaces the function call.
Review:
Function Frame Model

• How about functions that call other functions?

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

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

```python
>>> sumSquare(5, 3)
```

```python
sumSquare(5, 3)
```

```python
a  5  b  3
```

```python
return square(a) + square(b)
```

```python
square(5)
```

```python
x  5
```

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

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

>>> sumSquare(5,3)

25
def sumSquare(a, b):
    return square(a) + square(b)

>>> sumSquare(5,3)
25

square(5)
x
return x * x

square(3)
x
return x * x
def sumSquare(a, b):
    return square(a) + square(b)

>>> sumSquare(5, 3)
34

sumSquare(5, 3)
---
\[ a = 5 \quad b = 3 \]

\[
\text{return square}(a) + \text{square}(b)
\]

square(5)
---
\[ x = 5 \]

\[
\text{return } x \times x
\]

square(3)
---
\[ x = 3 \]

\[
\text{return } x \times x
\]
Function Frame Model to Understand `countDown`
def countDown(n):
    '''Prints ints from n down to 1'''
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

>>> countDown(5)
5
4
3
2
1

>>> countDown(4)
4
3
2
1
countDown(3)

n 3

if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(2)

n 2

if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(1)

n 1

if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(0)

n 0

if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

Base case reached!

>>> countDown(3)
3
2
1

Implicit return
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)
```

```python
>>> countDown(3)
3
2
1
```

**Base case reached!**

Implicit return
```python
countDown(3)

n 3

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

>>> countDown(3)
3
2
1

countDown(2)

n 2

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

countDown(1)

n 1

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

Base case reached!

countDown(0)

n 0

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

Implicit return
```

Implicit return
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)
```

**Base case reached!**

```python
>>> countDown(3)
3
2
1
```

Implicit return

Implicit return

Implicit return
```python
countDown(3)

n = 3
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(2)

n = 2
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(1)

n = 1
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(0)

n = 0
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

>>> countDown(3)
3
2
1

Base case reached!
```

The function `countDown(n)` prints numbers in ascending order until reaching the base case where `n < 1`, at which point it stops and returns. Here, we start with `n = 3` and print `3`, then `2`, then `1` until it reaches the base case of `0`, where it stops and returns.

The diagram illustrates the recursive calls and the flow of execution, showing how the function calls itself with decreasing values of `n` until the base case is reached.
The end!