CS 134 Lecture 20: More Recursion
Announcements & Logistics

- **HW 6** on GLOW due Mon at 10pm
  - Good practice for short-code questions on exam
  - Practice on pencil and paper first
- Lab 7, 8, and 9 are **partner labs**
  - Pair programming is an important skill as well as a vehicle for learning
- Colloquium Today: **Tim Randolph ’18**
  - Theoretical computer science talk on the Subset Sum problem (a problem you may use a “brute-force” approach to solve recursively in a future assignment!)

Do You Have Any Questions?
Last Time

- Introduction to recursion
- Alternative to iteration
- New problem solving paradigm
- Function frame model to understand recursion behind the scenes
A recursive function is a function **that calls itself**

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

- **Recursive step.** Does the following things:
  - Performs an action that contributes to the solution
  - Reduces the problem to a smaller version of the same problem, and calls the function on this **smaller subproblem**

- The recursive step is a form of "wishful thinking" (also called the inductive hypothesis)
More Recursion: count_up
count_up(n)

• Write a recursive function that prints integers from 1 up to n and then prints "DONE!"

• Recursive definition of count_up:
  • Base case: n <= 0, pass # print nothing
  • Recursive rule: call count_up(n-1), print(n)

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

>>> count_up(4)
1
2
3
4

>>> count_up(3)
1
2
3
count_up(n)

- Unlike `count_down(n)` the print statement is **after** the recursive function call (*why?*)
- 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 <= 0:
        pass  # can omit this if of course
    else:
        count_up(n-1)
        print(n)
```

```bash
>>> count_up(5)
1
2
3
4
5
```
Function Frame Model to Understand count_up
```python
def count_up(n):
    if n <= 0:
        pass
    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!

```python
def print_halves_gotcha(n):
    """Prints n, n/2, down to ... 1""
    if n > 0:
        print(n)
        return print_halves_gotcha(n/2)
```

Always true!
"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
Recursion vs. Iteration: \texttt{sum\_list}
sum_list

• **Goal:** Write a function to sum up a list of numbers
• Iterative approach? (i.e., using loops?)
Iterative Approach to `sum_list`

- **Goal:** Write a function to sum up a list of numbers
- Iterative approach:

```python
def sum_list_iterative(num_lst):
    sum = 0
    for num in num_lst:
        sum += num
    return sum
```

```python
>>> sum_list_iterative([3, 4, 20, 12, 2, 20])
61
```
sum_list

• **Goal:** Write a function to sum up a list of numbers
• Recursive approach?
Recursive approach to \texttt{sum\_list}

- **Base case:**
  - \texttt{num\_lst} is empty, return 0

- **Recursive rule:**
  - Return first element of \texttt{num\_lst} plus result from calling \texttt{sum\_list} on rest of the elements of the list.

- **Example:** Suppose \texttt{num\_lst} = [6, 3, 6, 5]
  - \texttt{sum\_list([6, 3, 6, 5])} = 6 + \texttt{sum\_list([3, 6, 5])}
  - \texttt{sum\_list([3, 6, 5])} = 3 + \texttt{sum\_list([6, 5])}
  - \texttt{sum\_list([6, 5])} = 6 + \texttt{sum\_list([5])}
  - \texttt{sum\_list([5])} = 5 + \texttt{sum\_list([])}
  - For the base case we have \texttt{sum\_list([])} returns 0
Recursive approach to **sum_list**

- **Base case:**
  - `num_lst` is empty, return **0**

- **Recursive rule:**
  - Return first element of `num_lst` plus result from calling `sum_list` on rest of the elements of the list.

- **Example:** Suppose `num_lst = [6, 3, 6, 5]`
  - \(\text{sum_list}([6, 3, 6, 5]) = 6 + \text{sum_list}([3, 6, 5])\)
  - \(\text{sum_list}([3, 6, 5]) = 3 + \text{sum_list}([6, 5])\)
  - \(\text{sum_list}([6, 5]) = 6 + \text{sum_list}([5])\)
  - \(\text{sum_list}([5]) = 5 + \text{sum_list}([])\)
  - For the base case we have `sum_list([])` returns **0**
Recursive approach to \texttt{sum\_list}

\begin{verbatim}
def sum_list(num_lst):
    \"Returns sum of given list\"
    if not num_lst:
        return 0
    else:
        return num_lst[0] + sum_list(num_lst[1:])

>>> sum_list([3, 4, 20, 12, 2, 20])
61
\end{verbatim}
Compare **sum_list** approaches

- Compare/Contrast:

```python
def sum_list_iterative(num_lst):
    sum = 0
    for num in num_lst:
        sum += num
    return sum

def sum_list(num_lst):
    if num_lst == []:  # Fixing the comparison to an empty list
        return 0
    else:
        return num_lst[0] + sumList(num_lst[1:])
```
Graphical Recursion
The Turtle Module

- Turtle is a **graphics module** first introduced in the 1960s by computer scientists Seymour Papert, Wally Feurzig, and Cynthia Solomon.
- It uses a programmable cursor — fondly referred to as the “turtle” — to draw on a Cartesian plane (x and y axis.)
Turtle In Python

- **turtle** is available as a built-in module in Python. See the [Python turtle module API](#) for details.
- Basic turtle commands:

Use `from turtle import *` to use these commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd(dist)</code></td>
<td>turtle moves forward by <code>dist</code></td>
</tr>
<tr>
<td><code>bk(dist)</code></td>
<td>turtle moves backward by <code>dist</code></td>
</tr>
<tr>
<td><code>lt(angle)</code></td>
<td>turtle turns left <code>angle</code> degrees</td>
</tr>
<tr>
<td><code>rt(angle)</code></td>
<td>turtle turns right <code>angle</code> degrees</td>
</tr>
<tr>
<td><code>up()</code></td>
<td>(pen up) turtle raises pen in belly</td>
</tr>
<tr>
<td><code>down()</code></td>
<td>(pen down) turtle lowers pen from belly</td>
</tr>
<tr>
<td><code>shape(shp)</code></td>
<td>sets the turtle's shape to <code>shp</code></td>
</tr>
<tr>
<td><code>speed(spd)</code></td>
<td>sets the turtle's speed 1-10 (slow-fast). 0 skips animation.</td>
</tr>
<tr>
<td><code>home()</code></td>
<td>turtle returns to (0,0) (center of screen)</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>delete turtle drawings; no change to turtle's state</td>
</tr>
<tr>
<td><code>reset()</code></td>
<td>delete turtle drawings; reset turtle's state</td>
</tr>
<tr>
<td><code>setup(width, height)</code></td>
<td>create a turtle window of given width and height</td>
</tr>
</tbody>
</table>
Basic Turtle Movement

- `forward(dist)` or `fd(dist)`,
- `left(angle)` or `lt(angle)`,
- `right(angle)` or `rt(angle)`,
- `backward(dist)` or `bk(dist)`

```python
# set up a 400x400 turtle window
setup(400, 400)
reset()

fd(100)  # move the turtle forward 100 pixels
lt(90)   # turn the turtle 90 degrees to the left
fd(100)  # move forward another 100 pixels

# complete a square
lt(90)
fd(100)
lt(90)
lt(90)
fd(100)
done()
```
Drawing Basic Shapes With Turtle

- We can write functions that use turtle commands to draw shapes.
- For example, here's a function that draws a square of the desired size

```python
def draw_square(length):
    # a loop that runs 4 times
    # and draws each side of the square
    for i in range(4):
        fd(length)
        lt(90)
done()
```

```python
setup(400, 400)
reset()
draw_square(150)
```
Drawing Basic Shapes With Turtle

• How about drawing polygons?

```python
def draw_polygon(length, num_sides):
    for i in range(num_sides):
        fd(length)
        lt(360/num_sides)
    done()
```

draw_polygon(80, 3)  
draw_polygon(80, 10)
Adding Color!

- What if we wanted to add some color to our shapes?

```python
def draw_polygon_color(length, num_sides, color):
    # set the color we want to fill the shape with
    # color is a string
    fillcolor(color)

    begin_fill()
    for i in range(num_sides):
        fd(length)
        lt(360/num_sides)
    end_fill()
    done()
```

draw_polygon_color(80, 10, "gold")
draw_polygon_color(80, 10, "purple")
Next Time: Recursive Figures With Turtle

- Next time we will explore how to draw recursive pictures with Turtle