CSCI 134 Fall 2021:
More Recursion

October 27, 2021

Shikha Singh, 9AM
Jeannie Albrecht, 10AM
Announcements & Logistics

- **Lab 6 due today/tomorrow 10 pm**
  - Remember to test your hIndex function thoroughly
  - What are some good test cases?
    - Empty tuples, Singletons, tuples of zeroes
    - Tuples with duplicate citation counts
    - Tuples where if condition is True/False
    - Is there a default return value outside conditionals?
- **HW 6** will be posted at noon today
  - Covers lambda sorting, dictionaries, sets

Do You Have Any Questions?
Last Time

• Learned about **recursive functions**
  • Functions that **call themselves**

• Recursion is useful for:
  • Solving problems that can be broken down into smaller versions of the same problem
  • Writing simple and elegant code for seemingly complex problems
Recap: 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.
  - **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)
Correction: `countUp(n)`

- Write a recursive function that prints integers from 1 up to n (without using any loops)
- Easy to do iteratively (using loops)
- How would we solve this recursively, without loops?

```python
def countUp(n):
    # Prints out integers from 1 up to n''
    if n < 1:
        pass  # do nothing
    else:
        countUp(n-1)
        print(n)
```

**Head recursion:** the printing happens "on the way back"
Correction: countUp(n)

• Write a recursive function that prints integers from 1 up to n (without using any loops)

• Easy to do iteratively (using loops)

• How would we solve this recursively, without loops?

```python
CORRECTION: countUp(n)

• Head recursion: the printing happens "on the way back"

else:
    countUp(n-1)
    print(n)

Tail Recursion: A call is *tail recursive* if nothing has to be done after the call returns. That is, when the call returns, the returned value is immediately returned from the calling function.

Head Recursion: A call is *head recursive* when the first statement of the function is the recursive call.

CORRECTION: In class we called this tail recursion. This was a mistake! This is a corrected version of the slides.
```
Today’s Plan

• Continue discussing recursion!

• Look at graphical recursive examples using the turtle library in Python
Recap: Iterative Approach to `sumList`

```python
def sumListIterative(numList):
    sum = 0
    for num in numList:
        sum += num
    return sum
```

```
3, 4, 20, 12, 2, 20
```

```
61
sum
```
Recap: Recursive approach to `sumList`

- **Base case?**
  - Empty list: return sum as 0

- **Recursive step?**
  - 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*

```python
def sumList(numList):
    """Returns sum of given list""
    if not numList:
        return 0
    else:
        return numList[0] + sumList(numList[1:])
```

Example of *fruitful* recursion that returns a value
sumUp()

• Let’s consider another example
• Suppose we want to write a function to return the sum of numbers from 1 up to n?
• We already saw countUp() last time…
• How is this different?

def countUp(n):
    '''Prints out integers from 1 up to n'''
    if n < 1:
        pass  # do nothing
    else:
        countUp(n-1)
        print(n)
sumUp()

- Let's consider another example
- Suppose we want to write a function to return the sum of numbers from 1 up to n?
- We already saw countUp() last time...
- How is this different?
  - This time we want to return the sum rather than just printing the numbers
    - Base case?
    - Recursive case?
      - Can think of sum(5) as 5 + sumUp(4)…
sumUp()

- What changes when we switch from a function that only prints to a **fruitful** function that returns a value?

```python
def countUp(n):
    """Prints out integers from 1 up to n""
    if n < 1:
        pass  # do nothing
    else:
        countUp(n-1)
        print(n)
```

```python
def sumUp(n):
    """Returns sum of integers from 1 up to n""
    if n < 1:

    else:
```
What changes when we switch from a function that only prints to a **fruitful** function that returns a value?

```python
def countUp(n):
    '''Prints out integers from 1 up to n'''
    if n < 1:
        pass  # do nothing
    else:
        countUp(n-1)
        print(n)
```

```python
def sumUp(n):
    '''Returns sum of integers from 1 up to n'''
    if n < 1:
        return 0
    else:
        return n + sumUp(n-1)
```
• Can also name the “sub-results” separately
• Sometimes makes your code easier to understand

```python
def sumUp(n):
    """Returns sum of integers from 1 up to n""
    if n < 1:
        return 0
    else:
        subResult = sumUp(n-1)  # Result returned by recursive call
        result = n + subResult  # Result for whole call
        return result
```
def sumUp(n):
    """Returns sum of integers from 1 up to n"""
    if n < 1:
        return 0
    else:
        return n + sumUp(n-1)

sumUp(4)
⇒ 4 + sumUp(3)
⇒ 4 + (3 + sumUp(2))
⇒ 4 + (3 + (2 + sumUp(1)))
⇒ 4 + (3 + (2 + (1 + sumUp(0))))
⇒ 4 + (3 + (2 + (1 + 0)))
⇒ 4 + (3 + (2 + 1))
⇒ 4 + (3 + 3)
⇒ 4 + 6
⇒ 10
Fruitful Recursion: Base Case(s) Always Required!

```
def countUp(n):
    if n <= 0:
        pass
    else:
        countUp(n-1)
        print(n)
```

```
def countUp(n):
    if n > 0:
        countUp(n-1)
        print(n)
```

```
def sumUp(n):
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```

What happens if we eliminate the base case for `sumUp`?

Source: (http://cs111.wellesley.edu/spring19)
Fruitful Recursion: Base Case(s) Always Required!

```python
def countUp(n):
    if n <= 0:
        pass
    else:
        countUp(n-1)
        print(n)
```

```python
def countUp(n):
    if n > 0:
        countUp(n-1)
        print(n)
```

```python
def sumUp(n):
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```

sumUp(0) would return None! Can’t add anything to None.

Source: (http://cs111.wellesley.edu/spring19)
Another Example: Palindromes

EVE
CIVIC
MADAM
AVID DIVA
STEP ON NO PETS
STRESSED DESSERTS
ABLE WAS I ERE I SAW ELBA
LIVED ON DECAF FACED NO DEVIL
Recursive Approach

- We want to return True if string s is a palindrome
- **Base case?**
Recursive Approach

• We want to return True if string s is a palindrome

• **Base case?**
  • Strings of length 0 or 1
  • Return True!

```
palindrome(s)
```

```
s
```

```
palindrome(s')
```

```
0
```

```
-1
```
Recursive Approach

• We want to return True if string s is a palindrome

• **Base case?**
  - Strings of length 0 or 1
  - Return True!

```python
def palindrome(s):
    """Returns True if string s is a palindrome."""
    if len(s) == 0 or len(s) == 1:
        return True
    else:
```

Recursive Approach

• **Recursive step?**
  • 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**
Recursive Approach

- **Recursive step?**
  - Strip off first and last letter, compare if they are the same
  - Check if remaining string $s'$ is a palindrome
  - Return True only if both previous results are True
Recursive Approach

- **Recursive step?**
  - Strip off first and last letter, compare if they are the same
  - Check if remaining string $s'$ is a palindrome
  - Return True only if both previous results are True

```python
def palindrome(s):
    """Returns True if string s is a palindrome."""
    if len(s) == 0 or len(s) == 1:
        return True
    else:
        subresult = palindrome(s[1:-1])
        return s[0].lower() == s[-1].lower() and subresult
```
Takeaways: Fruitful Recursion

• Fruitful recursion is recursion that "computes and returns" values

• We always need to implement the base case!

• We always need to store the value returned by recursive calls!

• We can debug code using print statements
Debugging Recursive Code

- Adding print statements can help demystify the magic
- Make sure to print enough information to understand the state of the execution

```python
def palindromeDebug(s):
    """Returns True if string s is a palindrome."""
    print("Entering palindrome('{}').format(s))
    if len(s) == 0 or len(s) == 1:
        print('Reached base case. Returning True')
        return True
    else:
        subresult = palindromeDebug(s[1:-1])
        result = s[0].lower() == s[-1].lower() and subresult
        print("Returning {} from palindrome('{}').format(result, s))
        return result
```
def palindromeDebug(s):
    """Returns True if string s is a palindrome."""
    print("Entering palindrome('{}')".format(s))
    if len(s) == 0 or len(s) == 1:
        print('Reached base case. Returning True')
        return True
    else:
        subresult = palindromeDebug(s[1:-1])
        result = s[0].lower() == s[-1].lower() and subresult
        print("Returning {} from palindrome('{}')".format(result, s))
        return result

In [12]: palindromeDebug('shannahz')

Entering palindrome('shannahz')
Entering palindrome('hannah')
Entering palindrome('anna')
Entering palindrome('nn')
Entering palindrome('')
Reached base case. Returning True
Returning True from palindrome('nn')
Returning True from palindrome('anna')
Returning True from palindrome('hannah')
Returning False from palindrome('shannahz')

Out[12]: False
Recursive Patterns
triangle($n$, $c$)

- Write a recursive function called `triangle` that takes a length parameter $n$ and a character $c$ and draws a triangle pattern composed of lines of $c$, starting with length $n$ down to 1.

- For example:

  ```python
  >>> triangle(10, '*')
  
  ****************
  ************
  ***********
  **********
  *********
  ********
  ********
  *****
  ***
  *  
  ```

  **Base case?**

  **Recursive case?**
triangle(n, c)

• Write a recursive function called `triangle` that takes a length parameter `n` and a character `c` and draws a triangle pattern composed of lines of `c`, starting with length `n` down to 1.

• For example:

```python
>>> triangle(10, '*')

************
************
************
************
************
*****
****
***
**
*  
```

Base case?

\[ n < 1 \]:

do nothing

Recursive case?
Write a recursive function called `triangle` that takes a length parameter `n` and a character `c` and draws a triangle pattern composed of lines of `c`, starting with length `n` down to 1.

For example:

```python
>>> triangle(10, '*')
```

```
***********
**********
*********
********
******
*****
****
***
**
*  
```

**Base case?**

n < 1:

do nothing

**Recursive case?**

print (c * n)

triangle(n-1, c)
triangle(n, c)

• Write a recursive function called \texttt{triangle} that takes a length parameter \texttt{n} and a character \texttt{c} and draws a triangle pattern composed of lines of \texttt{c}, starting with length \texttt{n} down to \texttt{1}.

• For example:

```python
def triangle(n, c):
    if n < 1:
        pass # do nothing
    else:
        print(n * c)
        triangle(n - 1, c)
```

```plaintext
**********
**********
******
****
***
**
*
```
triangleAlt(n, c1, c2)

• Write a recursive function called triangleAlt that takes int n and two characters c1 and c2 and draws a triangle pattern similar to the previous example but now the rows must alternate between c1 and c2 (starting with c1).

>>> triangleAlt(10, '@', '#')

@@@@@@@@@@
#############
@@@@@@@@@@
#############
@@@@@@@@@@
#############
@@@@@@@@@@
#############
@@@@@@@@@@
#############
@@@@@@@@@@
#############
Write a recursive function called `triangleAlt` that takes integers `n` and two characters `c1` and `c2` and draws a triangle pattern similar to the previous example but now the rows must alternate between `c1` and `c2` (starting with `c1`).

```python
def triangleAlt(n, c1, c2):
    if n < 1:
        pass  # do nothing
    else:
        print(n * c1)
        triangleAlt(n-1, c2, c1)
```
Function Frame Model:
triangleAlt(3, '*', '^')
```python
triangleAlt(3, '*', '^')

n 3 c1 '*' c2 '^'

if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)

triangleAlt(2, '*', '^')

n 2 c1 '^' c2 ' '*

if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)

triangleAlt(1, '*', '^')

n 1 c1 '*' c2 '^'

if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)

>>> triangleAlt(3, '*', '^')
***
^^
*```
Moving on…
Recursion with Turtle Graphics
Graphical Recursion with Turtle

- Graphical recursion using the Python `turtle` module
- The `turtle` leaves behind a trail allowing us to draw pretty pictures!
Turtle

• Python has a built-in module named `turtle`. 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>pu()</code></td>
<td>(pen up) turtle raises pen in belly</td>
</tr>
<tr>
<td><code>pd()</code></td>
<td>(pen down) turtle lower pen in belly</td>
</tr>
<tr>
<td><code>pensize(width)</code></td>
<td>sets the thickness of turtle's pen to <code>width</code></td>
</tr>
<tr>
<td><code>pencolor(color)</code></td>
<td>sets the color of turtle's pen to <code>color</code></td>
</tr>
<tr>
<td><code>shape(shp)</code></td>
<td>sets the turtle's shape to <code>shp</code></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 <code>width</code> and <code>height</code></td>
</tr>
</tbody>
</table>

Source: [http://cs111.wellesley.edu/spring19](http://cs111.wellesley.edu/spring19)
Basic Turtle Movement

- **forward** or **fd**, **left** or **lt**, **right** or **rt**, **backward** or **back**

```python
In [1]: from turtle import *
   # setup(400,400, 0, 0) # open the Turtle graphics window
   fd(100) # move turtle forward 100 pixels

In [2]: lt(90) # change direction by 90 degrees left
   fd(100) # move forward 100 pixels

In [3]: # completing the square
   lt(90)
   fd(100)

In [4]: lt(90)
   fd(100)
```
Useful Turtle Commands

- **clear()** - Erase the turtle's canvas. Doesn't change turtle's properties (pen size, color, etc)

- **reset()** - Return the turtle in its original state, including pen size and color

- **speed()** - Control the speed with which the turtle moves. The fastest speed setting is 10, slowest is 1. A speed of 0 is a special number. It turns off animation and will draw the graphics instantly.
Drawing Polygons

- We can use loops to draw shapes, such as polygons

```python
def polygon(numSides, sideLength):
    """Draws a polygon with the specified number of sides, each with the specified length."""
    for side in range(numSides):
        fd(sideLength)
        lt(360/numSides)
```
Drawing Polygons

In [5]:
# Draw a triangle, allow user to click on canvas to close window
# 0,0 sets the position of the canvas at the top-left corner
reset()
pensize(3)
polygon(3, 200)

In [6]:
# Draw a square, click on canvas to close window
# setup(400, 400, 0, 0)
reset()
pensize(5)
polygon(4, 200)

In [7]:
# Draw an approximate circle, click on canvas to close window
reset()
pensize(5)
polygon(100, 7)
More next time!