CSCI 134 Fall 2021: Introduction to Recursion

October 25, 2021

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

- Lab 6 due Wed/Thurs 10 pm
  - Uses dictionaries, tuples, sorting with lambda, plotting, CSV files
  - Analyzes supreme court data
  - Any questions?
- HW 6 will be posted on Wednesday

Do You Have Any Questions?
Last Time: Data Structure Review

• Wrapped up dictionaries and sets

• Data structure discussion: which to use when
  • List/tuples: when order matters, dictionaries/sets: when it does not

• Tuples or lists?
  • Do we need to add/remove items dynamically? If yes, use lists (they are mutable!); if data stays same (no changes), use tuples (more space efficient)

• Lists vs dictionaries?
  • Dictionaries/sets have huge performance benefits over a list as they store "hashes" of elements and thus support fast look ups/insert/deletes
  • This is why dictionaries are also referred to as hash tables in other programming languages

• If you want to learn about the implementation and trade off various data structures: take CSCI 136!
Today’s Plan: Introduction to Recursion

- Recursion is awesome: it is an important problem solving paradigm that can not only lead to beautiful, elegant code, it can also be used to do really cool things!
- By the end of the week, you'll be able to use recursion to draw these beautiful pictures (or see this video from S20)
Recursion

• We have seen many examples of functions calling other functions
• A recursive function is a function that calls itself
• Why would we want that?
  • To solve problems that can be broken down to smaller versions of the same problem
Recursion

• We have seen many examples of functions calling other functions
• A **recursive function** is a function that **calls itself**
• What do these images have in common?
  • They are made up of smaller versions of themselves
  • This idea is the core of recursion
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": assuming the **recursion fairy** will take care of the smaller problem
  • In CS136/256, the recursion fairy will introduced more formally as the inductive hypothesis
Let us discuss the difference between a recursive and iterative approach using examples.

**Example 1: countDown:** Write a function that prints integers from \( n \) down to 1.

**Iterative approach.** Using loops.

```
In [1]: def countDownIterative(n):
        for i in range(n, 0, -1):
            print(i)

In [2]: countDownIterative(5)
```

5
4
3
2
1
Recursive Approach: `countDown`

- **Base case(s)**. When the problem is so small, we can solve it directly, without having to reduce it any further.

```python
def countDown(n):
    '''Prints ints from n down to 1'''
    if n < 1:
        pass  # do nothing
```

```
In[1] countDown(0)
```

Nothing is printed
Recursive Approach: `countDown`

- **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**
- Can think of `countDown(5)` as `print(5)` followed by `countDown(4)`

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
countDown: Recursive Step

- **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*

- Can think of `countDown(5)` as `print(5)` followed by `countDown(4)`

```python
def countDown(n):
    '''Prints ints from n down to 1'''
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)
```
**countDown**: Recursive Step

- **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*

- Can think of `countDown(5)` as `print(5)` followed by `countDown(4)`

```python
def countDown(n):
    '''Prints ints from n down to 1'''
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)
```
countDown: Implicit Base Case

- It is possible to simplify our function by omitting the base case
- The following two versions are equivalent
- However, in recursion, we prefer that you write the base case explicitly for pedagogical reasons
  - Bad things happen if you forget the base case.... (we'll see)

```
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)
```
What is Going On?

- Notice that in our recursive function, **we did not use any loops!**
- And yet we were able to accomplish a similar repetitive behavior only using functions and conditionals
  - This is the beauty of recursion
- For now, you should have two questions:
  - Why are we doing this?
  - What is going on behind the scenes?
- 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

• 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)

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

• The function frame (the state of the local variables inside it) are destroyed after the `return`

• If a function does not have an explicit return statement, then we return `None` after all statements in the function body are executed
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)`?
def sumSquare(a, b):
    return square(a) + square(b)

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

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

```
return square(5) + square(3)
```

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

Base case reached!
```python
countDown(3)

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

3
2
1

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

Base case reached!
```
countDown(3)
```
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

countDown(3)
```
More Recursion: countUp
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?

\[
\text{In[1] countUp(5)}
\]

1
2
3
4
5

\[
\text{In[2] countUp(4)}
\]

1
2
3
4

\[
\text{In[3] countUp(3)}
\]

1
2
3
countUp(n)

- **Base case(s).** When the problem is **so small**, we can solve it directly, without having to reduce it any further.

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

- **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

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

countUp(3)
```

Base case reached!
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"
```
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?

```
countUp(n)

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

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

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.

Head recursion: the printing happens "on the way back"
```
More Recursion:

countDownUp
countDownUp(n)

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

\[
\begin{array}{l}
\text{In[1] countDownUp(5)}\\
5 \\
4 \\
3 \\
2 \\
1 \\
1 \\
2 \\
3 \\
4 \\
5 \\
\end{array}
\quad
\begin{array}{l}
\text{In[2] countDownUp(4)}\\
4 \\
3 \\
2 \\
1 \\
1 \\
2 \\
3 \\
4 \\
\end{array}
\quad
\begin{array}{l}
\text{In[3] countDownUp(3)}\\
3 \\
2 \\
1 \\
1 \\
2 \\
3 \\
\end{array}
\]
Write a recursive function that prints integers from \( n \) down to \( 1 \) and back up from \( 1 \) up to \( n \) (without using any loops)

```python
def countDownUp(n):
    """Prints integers from \( n \) down to \( 1 \) and then from \( 1 \) up to \( n \)."""
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDownUp(n-1)
        print(n)
```
Recursion **GOTCHAs!**
I'm going to write a recursive function

With a base case, right?
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 countDownGotcha(n):
    '''Prints ints from n down to 1'''
    if n < 1:
        pass # do nothing
    else:
        print(n)
        countDownGotcha(n)
```

Subproblem not getting smaller!
GOTCHA #2

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

```python
def printHalvesGotcha(n):
    if n > 0:
        print(n)
        printHalvesGotcha(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 with Return
(Fruitful Recursion)
Iterative Approach: \texttt{sumList}

\begin{verbatim}
def sumListIterative(numList):
    sum = 0
    for num in numList:
        sum += num
    return sum
\end{verbatim}

3, 4, 20, 12, 2, 20

\texttt{sum}
Iterative Approach: `sumList`

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

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

```
7
```
Iterative Approach: `sumList`

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

3, 4, 20, 12, 2, 20

61

sum
Recursive Approach

• Base case?
  • When is the list so small that the sum is trivial?
  • Empty list: return sum as 0

```python
def sumList(numList):
    """Returns sum of given list""
    if numList:
        return 0
    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

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

```
numList[0] + sumList(numList[1:])
```

**Smaller version of the same problem**
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

```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
Recursive Patterns
triangle(n, c)

>>> triangle(10, '*')

************
************
************
************
************
************
*****
****
***
**
*
triangle(n, c)

>>> triangle(10, '*')

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

```python
def triangle(n, c):
    if n < 1:
        # do nothing
        pass
    else:
        print( n * c)
        triangle(n - 1, c)
```
triangleAlt(n, c1, c2)

```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, '∗', '＾')
```
triangleAlt(3, '*', '^')

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

>>> triangleAlt(3, '*', '^')
***
^^
*

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

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

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

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

Try this at Home
triangleDownUp(n, c1, c2)

>>> triangleDownUp(8, '@', '#')

```
@@@@@@@@@
##########
@@@@@@@@
########
@@@@@
###
@@
#
#
@@
###
@@@@@
##########
@@@@@@@@@
```

Next Lecture

• Graphical recursion using \texttt{turtle} module