Advanced #2: Recursion

SAMS SENIOR CS TRACK

Learning Goals

Understand what **recursion** means

Identify **base** and **recursive** cases in code

Write code using **recursive calls**

What is Recursion?

The core idea of recursion is that you can manipulate the control flow of a program by **calling a function from itself**.

This lets you repeat the same set of actions over and over again, similar to the behavior of a while loop. But since the repetition is managed in a function call, we can create control flow patterns that are difficult or even impossible to produce with loops.

How does Recursion work?

To write recursive code, we need to split up the problem we're solving into at least two parts:

The **recursive case** will solve the problem by calling the function again on a **slightly smaller** version of the input.

The **base case** will solve the problem for a defined input using no recursion at all.

If we set these up properly, the code will not run forever.

```
def recursiveFunction():
    if (this is the base case):
        do something non-recursive
    else:
        do something recursive
```

Seriously, how does it work?

This may seem like magic, but there's a logic behind how the code works.

Say you call the function to the right on the number 10. When it calls itself again, it will use the input 9, then 8, then 7... all the way down to 1.

At input 1, the function will return a value- also 1. That value will be passed back to the prior call, which will compute a result (2) and pass it back, etc. This eventually results in the original call getting a result, so it can compute its own result.

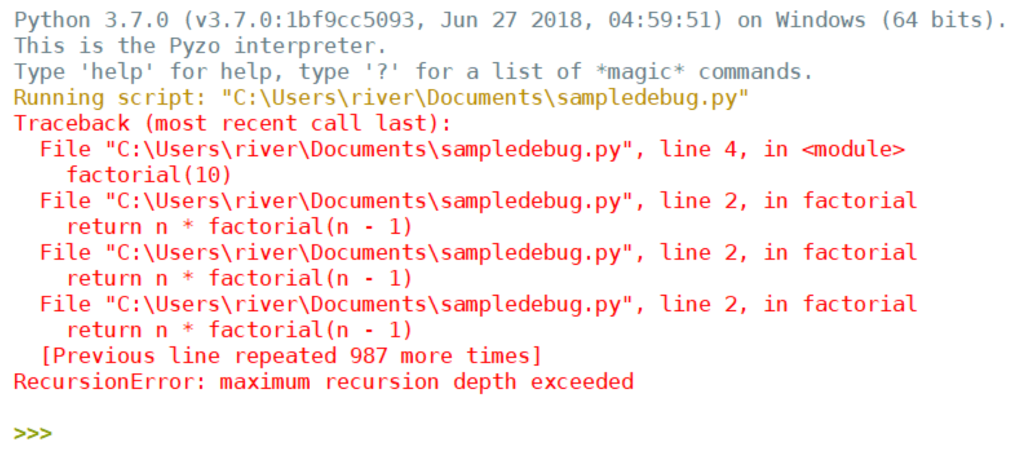
```
def factorial(n):
    if (n < 2):
        return 1
    else:
        return n * factorial(n - 1)
```

The base case is important!

The only reason why recursion can work is because of the **base case.** If the code recurses every time it is called, it will keep recursing until the computer runs out of memory. This is called a **RecursionError**.

```
def factorial(n):
    return n * factorial(n - 1)
```

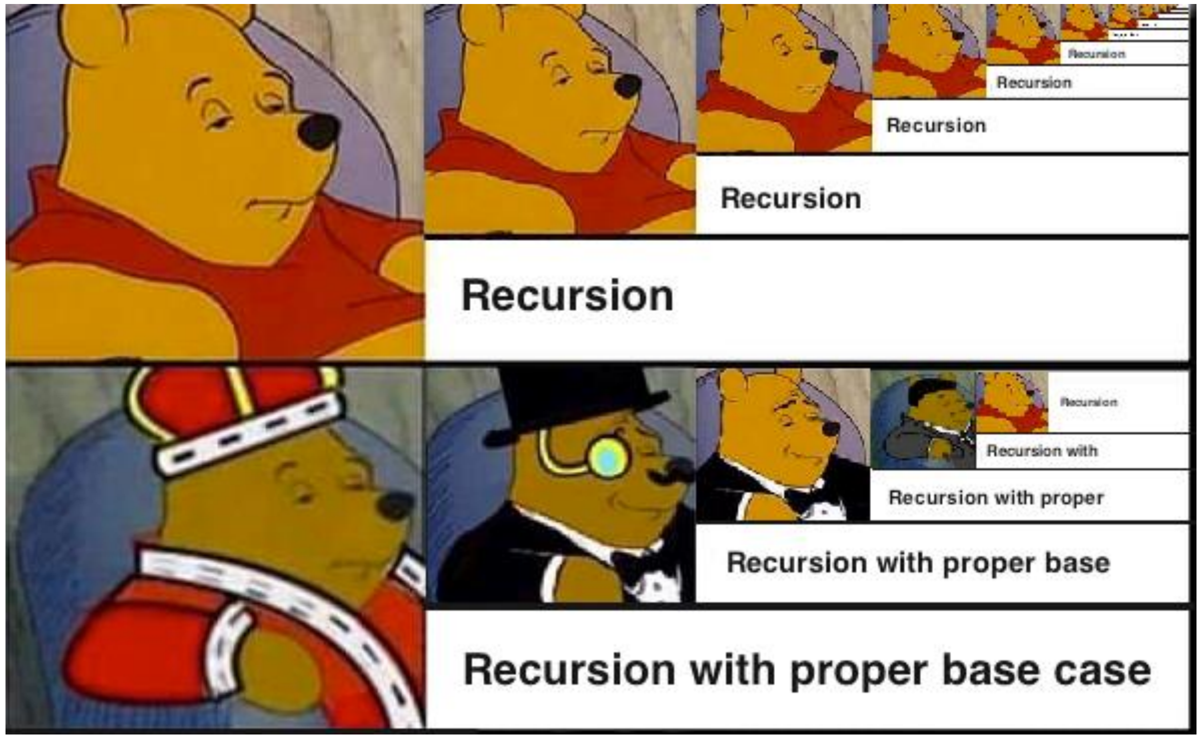
To avoid RecursionErrors, make sure you always have a base case, and make sure that you're always moving the input closer to that base case!



```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) on Windows (64 bits).
This is the Pyzo interpreter.
Type 'help' for help, type '?' for a list of *magic* commands.
Running script: "C:\Users\river\Documents\sampledebug.py"
Traceback (most recent call last):
  File "C:\Users\river\Documents\sampledebug.py", line 4, in <module>
    factorial(10)
  File "C:\Users\river\Documents\sampledebug.py", line 2, in factorial
    return n * factorial(n - 1)
  File "C:\Users\river\Documents\sampledebug.py", line 2, in factorial
    return n * factorial(n - 1)
  File "C:\Users\river\Documents\sampledebug.py", line 2, in factorial
    return n * factorial(n - 1)
  [Previous line repeated 987 more times]
RecursionError: maximum recursion depth exceeded

>>>
```

Infinite Recursion vs. Recursion w/ Base Case



Multiple Base Cases

We can include multiple base cases instead of one.

This is useful if we want to handle multiple possible stopping points.

Note: the code to the right uses lists, a data structure we'll learn about next week.

```
def interleave(list1, list2):
    if (len(list1) == 0):
        return list2
    elif (len(list2) == 0):
        return list1
    else:
        return [list1[0] , list2[0]] + \
            interleave(list1[1:], list2[1:])
```

Multiple Recursive Cases

We can also include multiple recursive cases when needed.

This is useful if we want to change the behavior of the function based on the input given.

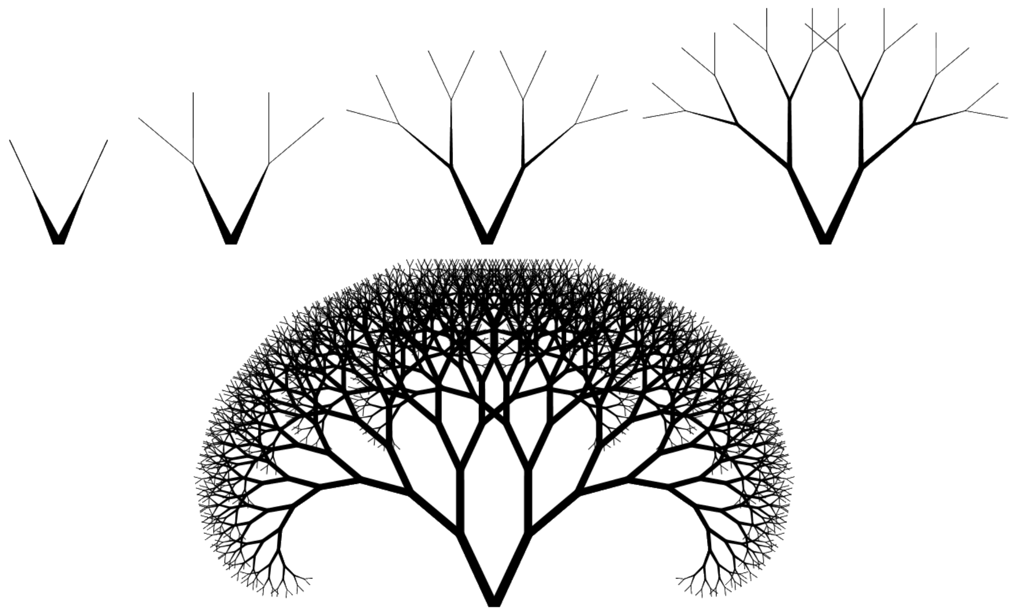
```
def power(base, expt):
    if (expt == 0):
        return 1
    elif (expt < 0):
        return 1.0 / power(base, abs(expt))
    else:
        return base * power(base, expt-1)
```

Multiple Recursive Calls

Finally, we can include **more than one recursive call** in the function to change the function's behavior.

Functions with one recursive call can usually be mimicked by loops. But multiple recursive calls let you manage control flow in more advanced ways, by repeating code at as many levels as you need.

Multiple recursive calls can be used to create cool images, like **fractals**, by repeating graphic algorithms at multiple points.



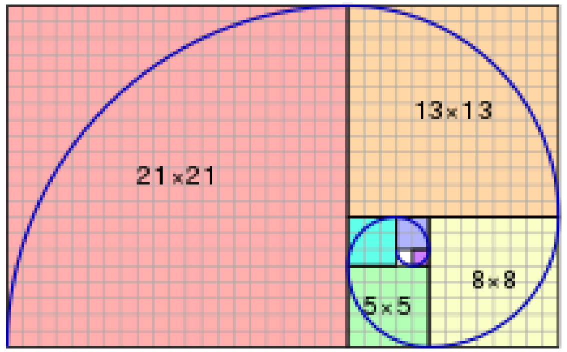
Multiple Call Example: Fibonacci

The **Fibonacci Sequence** is a famous example of a multiple-call recursive algorithm.

In this sequence, each number is generated by adding the two numbers that came before it. The base case of the sequence is the starting two numbers- 1 and 1.

This sequence is known for its relation to the **golden ratio**, which commonly appears in nature, mathematics, and architecture.

```
def fib(n):
    if (n < 2):
        return 1
    else:
        return fib(n-1) + fib(n-2)
```



Multiple Call Example: Mergesort

Another common example is **mergesort**, an algorithm which is used to sort a list of elements.

Mergesort works by recursively sorting the two halves of the list, then merging the two back together. This algorithm is known for being **more efficient** then many simpler sorting algorithms.

You can find a visualization of the algorithm here: http://math.hws.edu/eck/js/sorting/xSortLab.html

```
def merge(A, B):
    if ((len(A) == 0) or (len(B) == 0)):
        return A+B
    else:
        if (A[0] < B[0]):
            return [A[0]] + merge(A[1:], B)
        else:
            return [B[0]] + merge(A, B[1:])

def mergeSort(L):
    if (len(L) < 2):
        return L
    else:
        mid = len(L)//2
        left = mergeSort(L[:mid])
        right = mergeSort(L[mid:])
        return merge(left, right)
```

Learning Goals

Understand what **recursion** means

Identify **base** and **recursive** cases in code

Write code using **recursive calls**

Want to learn more? Try reading the notes here and here

Bonus Task

Bonus Task

Write the function powerSum(n, k) that takes two possibly-negative integers n and k and returns the so-called power sum: 1**k + 2**k + ... + n**k. If n is negative, return 0. Similarly, if k is negative, return 0.

You must use recursion to solve this problem; for loops, while loops, and the function sum() are not allowed.

Submit your answer to the bonus2 assignment on Autolab by **noon on Friday 7/12**.

Hint: Don't forget your base case!