Week 11 - Wednesday

COMP 1800

Last time

- What did we talk about last time?
- Exam 2
- Before that:
  - Review
- Before that:
  - Regular expressions

Questions?

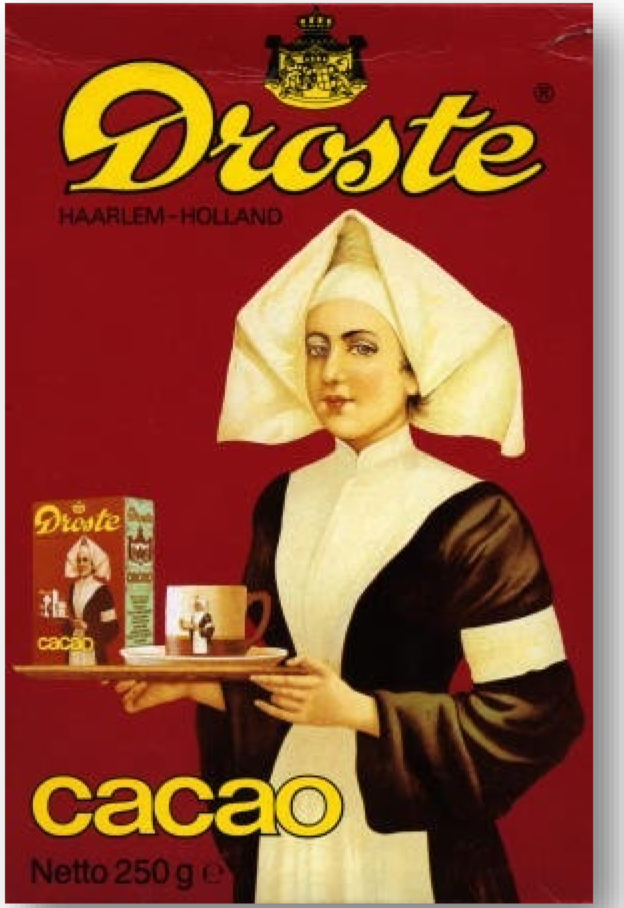
Assignment 8

Recursion

To understand recursion, you must first understand recursion.

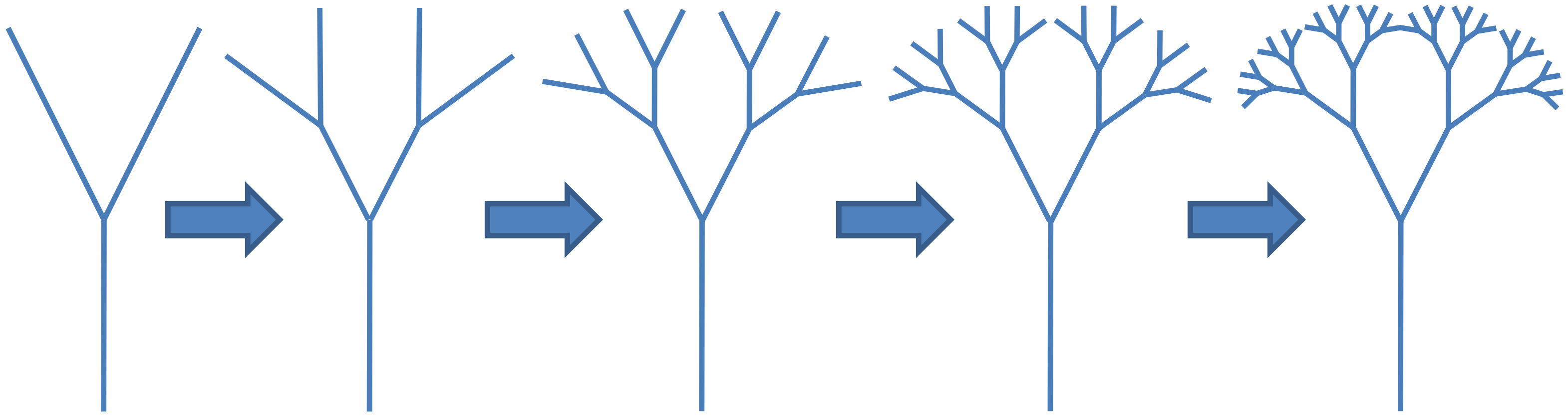
What is Recursion?

- Defining something in terms of itself
- To be useful, the definition must be based on progressively simpler definitions of the thing being defined



Bottom Up

- It's possible to define something recursively from the bottom up
- We start with a simple pattern and repeat the pattern, using a copy of the pattern for each part of the starting pattern



Top Down

Explicitly:

- $\boldsymbol{n}! = (\boldsymbol{n})(\boldsymbol{n} - 1)(\boldsymbol{n} - 2) \ldots (2)(1)$

Recursively:

- $\boldsymbol{n}! = (\boldsymbol{n})(\boldsymbol{n} - 1)!$
- $1! = 1$

- $6! = 6 \cdot 5!$
  - $5! = 5 \cdot 4!$
    - $4! = 4 \cdot 3!$
      - $3! = 3 \cdot 2!$
        - $2! = 2 \cdot 1!$
          - $1! = 1$
- $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$

Useful Recursion

Two parts:

- Base case(s)
  - Tells recursion when to stop
  - For factorial, $n = 1$ or $n = 0$ are examples of base cases
- Recursive case(s)
  - Allows recursion to progress
  - "Leap of faith"
  - For factorial, $n > 1$ is the recursive case

Solving Problems with Recursion

Approach for Problems

- Top down approach
- Don't try to solve the whole problem
- Deal with the next step in the problem
- Then make the "leap of faith"
- Assume that you can solve any smaller part of the problem

Walking to the Door

- Problem: You want to walk to the door
- Base case (if you reach the door):
  - You're done!
- Recursive case (if you aren't there yet):
  - Take a step toward the door

Problem

Implementing Factorial

- Base case ($\boldsymbol{n} \leq 1$):
  - $1! = 0! = 1$

- Recursive case ($\boldsymbol{n} > 1$):
  - $\boldsymbol{n}! = \boldsymbol{n}(\boldsymbol{n} - 1)!$

Code for Factorial

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

Base Case

Recursive Case

Recursion and loops are the same

- Any program that uses loops can be done with recursion
- Any program that uses recursion can be done with loops
- Sometimes it's easier to use loops
- Sometimes it's easier to use recursion
- A base case is necessary in recursion to tell the process when to stop
  - This is like a condition for while loop or the amount of iteration for a for loop
- A recursive case is necessary so that recursion can continue
  - This is similar to how a loop jumps back up to the top when it gets to the bottom

Adding up the numbers in a list

- Base case (Empty list):
  - 0

- Recursive case (At least one thing left in the list):
  - The value of the first thing plus the sum of the rest of the list

Code for Sum

```
def recursiveSum(list):
  if len(list) == 0:
     return 0
  else:
     return list[0] + recursiveSum(list[1:])
```

Base Case

Recursive Case

Finding the biggest number in a list

- Base case (List with one thing in it):
  - The first (and only) thing in the list
- Recursive case (More than one thing left in the list):
  - The maximum of the first thing in the list and whatever is the biggest thing in the rest of the list

Code for biggest number in list

```
def recursiveMax(list):
  if len(list) == 1:
     return list[0]
  else:
     return max(list[0], recursiveMax(list[1:]))
```

Base Case

Recursive Case

Tips for recursion

- Use it only in special circumstances, since it's usually slower than loops
- Recursive solutions are often impressive for how short the code is
- Some people love it, but it can be hard to think about
- Instead of trying to solve the entire problem, we think about unwrapping one layer of the problem
  - Don't think too much about what's going on in the other recursive calls since you can't access those variables
- You usually don't want to change the values of variables with **=** since that can make the recursion harder to think about

Drawing Recursively

Complex shapes

- Many natural things have recursive shapes:
  - Trees
  - Spiral shells
  - Blood vessels
  - Mountains
  - Snowflakes
- Using recursion, we can draw some complex, organic-looking shapes with only a little code

Drawing squares

- Let's start with a simple (non-recursive) function that draws a square with a turtle called **`yertle`** and a side length called **`side`**

```
def drawSquare(yertle, side):
    for i in range(4):
        yertle.forward(side)
        yertle.right(90)
```

- It works by going clockwise around the square
- It (importantly) returns **`yertle`** to the starting point

Nested squares

- We can use the **`drawSquare()`** function repeatedly to draw a series of nested squares with progressively smaller sides
- Base case (Side length < 1):
  - Do nothing (Seems odd but is not an unusual base case)
- Base case (Side length ≥ 1):
  - Draw a square with the given side length
  - Continue drawing nested squares with a side length that's 5 units smaller

Nested squares function

- Here is that function implemented in Python:

```
def nestedSquares(yertle, side):
    if side >= 1: # hidden base case
        drawSquare(yertle, side)
        nestedSquares(yertle, side - 5)
```

- This function is called like any normal function:

```
nestedSquares(someTurtle, 200)
```

Trees

- Squares are fine, but they're not very exciting (or very organic looking)
- We can extend the idea into drawing a tree shape
- A tree looks kind of like a capital Y
- But then, instead of straight lines, we can replace the two branches of the Y with smaller Y's
  - And so on …
    - And so on …

Recursion for tree drawing

- Base case (Trunk length < 5):
  - Do nothing
- Recursive case (Trunk length ≥ 5):
  - Move forward trunk length
  - Turn right 30°
  - Draw a tree (recursively) with a trunk length 15 units shorter
  - Turn left 60° (which turns back to the original heading plus another 30°)
  - Draw a tree (recursively) with a trunk length 15 units shorter
  - Turn right 30° (which turns back to the original heading)
  - Move backward the trunk length (returning to the starting point)

Tree function

- Here is that function implemented in Python:

```
def tree(yertle, trunkLength):
    if trunkLength >= 5: # hidden base case
        yertle.forward(trunkLength)
        yertle.right(30)
        tree(yertle, trunkLength - 15)
        yertle.left(60)
        tree(yertle, trunkLength - 15)
        yertle.right(30)
        yertle.backward(trunkLength)
```

Upcoming

Next time…

- Finish recursion
- Work time for Assignment 8
  - **Assignment 8 is hard!**

Reminders

- Work on Assignment 8