

## COMP2720: Automating Tools for New Media

“Do we have to write all those lines?”  
Recursion

## A very powerful idea: Recursion

- Recursion is writing functions that call *themselves*.
- When you write a recursive function, you write (at least) two pieces:
  - What to do if the input is the smallest possible item (piece of data),
  - What to do if the input is larger so that you:
    - (a) process one piece of the data,
    - (b) call the function to deal with the rest.

## First, a reminder of lists

```
>>> fred=[1,2,3,4,5]
>>> fred[0]
1
>>> fred[1:]
[2, 3, 4, 5]
```

- In functional programming languages, there are usually functions called *head* and *rest* for these two operations.
  - They're very common in recursion.

```
>>> print fred[:-1]
[1, 2, 3, 4]
```

## A recursive decreaseRed

```
def decreaseRed(alist):
    if alist == []: #Empty
        return
    setRed(alist[0],
           getRed(alist[0])*0.8)
    decreaseRed(alist[1:])
```

This actually won't work for reasonable-sized pictures — takes up too much memory in Java.

- If the list (of pixels) is empty, don't do anything.
  - Just return
- Otherwise,
  - decrease the red in the first pixel,
  - call decreaseRed on the rest of the pixels.
- Call it like:  
decreaseRed(getPixels(pic))

## Recursion can be hard to get your head around

- It really relies on you *trusting* your functions.
  - They'll do what you tell them to do.
  - So if a function decreases red on a list of pixels, just let it do that!
- Let's try some different ways to think about recursion.
  - But first, let's take a smaller problem.

## DownUp

- Let's define a function called downUp

```
>>> downUp("Hello")
Hello
ello
llo
lo
o
lo
llo
ello
Hello
```

## 3 ways to understand recursion

1. Procedural abstraction.
2. Trace it out (use a small problem like downUp to do this).
3. Little people method.

## 1. Procedural abstraction

- Break the problem down into the smallest pieces that you can write down easily as a function.
- Re-use as much as possible.

## downUp for one character words

```
def downUp1(word):  
    print word
```

- Obviously, this works:

```
>>> downUp1("I")  
I
```

## downUp for two-character words

- We'll reuse downUp1 since we have it already.

```
def downUp2(word):  
    print word  
    downUp1(word[1:])  
    print word  
>>> downUp2("it")  
it  
t  
it  
>>> downUp2("me")  
me  
e  
me
```

## downUp for three-character words

```
def downUp3(word):  
    print word  
    downUp2(word[1:])  
    print word  
>>> downUp3("pop")  
pop  
op  
p  
op  
pop  
>>> downUp3("top")  
top  
op  
p  
op  
top
```

Are we seeing a  
pattern yet?

## Let's try our pattern

```
def downUpTest(word):  
    print word  
    downUpTest(word[1:])  
    print word
```

## It starts right!

```
>>> downUpTest("hello")
hello
ello
llo
lo
o
```

A function can get called so much that the memory set aside for tracking the functions (called the *stack*) runs out, called a *stack overflow*.

I wasn't able to do what you wanted.  
The error java.lang.StackOverflowError has occurred  
Please check line 58 of C:\Documents and Settings\Mark  
Guzdia\My Documents\funcplay.py

## How do we stop?

- If we have only one character in the word, print it and STOP!

```
def downUp(word):
    if len(word)==1:
        print word
        return
    print word
    downUp(word[1:])
    print word
```

## That works

```
>>> downUp("Hello")
Hello
ello
llo
lo
o
lo
llo
ello
Hello
```

## 2. Let's trace what happens

- >>> downUp("Hello")
  - The len(word) is not 1, so we print the word
- Hello
  - Now we call downUp("ello")
  - Still not one character, so print it
- ello
  - Now we call downUp("llo")
  - Still not one character, so print it
- llo
  - downUp("lo")
  - Still not one character, so print it
- lo

## Still tracing

- lo
  - Now call `downUp("o")`
  - THAT'S ONE CHARACTER! PRINT IT AND RETURN!
- o

## On the way back out

- `downUp("lo")` now continues from its call to `downUp("o")`, so it prints again and ends.
- lo
  - `downUp("llo")` now continues (back from `downUp("lo")`)
  - It prints and ends.
- llo
  - `downUp("ello")` now continues.
  - It prints and ends.
- ello
  - Finally, the last line of the original `downUp("Hello")` can run.
- Hello

## 3. Little elves

- Some of the concepts that are hard to understand:
  - A function can be running multiple times and places in memory, with different input.
  - When one of these functions end, the rest still keep running.
- A great way of understanding this is to use the metaphor of a function call (a function invocation) as an *elf*.
  - (We'll use students in the class as elves.)

## Elf instructions:

1. Accept a word as input.
2. If your word has only one character in it, write it on the screen and you're done! Stop and sit down.
3. Write your word down on the "screen"
4. Hire another elf (student) to do these same instructions and give the new elf your word *minus* the first character.
  1. *Wait until the elf you hired is done.*
5. Write your word down on the "screen" again.
6. You're done!

## Exercises

- Try writing upDown

```
>>> upDown("Hello")
Hello
Hell
Hel
He
H
He
Hel
Hell
Hello
```

## Why use functional programming and recursion?

- You can do a lot in very few lines.
- Very useful techniques for dealing with hard problems.
- ANY kind of loop (FOR, WHILE, and many others) can be implemented with recursion.
  - It's the most flexible and powerful form of looping.

## Fibonacci sequence

- The ordered sequence of numbers:

$[a_1, a_2, a_3, \dots, a_{n-1}, a_n, a_{n+1}, \dots]$

- $a_1 = 1, a_2 = 1, a_3 = a_1 + a_2 = 1 + 1 = 2,$   
 $a_4 = a_3 + a_2 = 1 + 2 = 3,$   
.....  
 $a_{n+1} = a_n + a_{n-1}$

## Fibonacci sequence: procedural version

- The function to calculate first n elements of the F. list

```
def fib1(n):
    """Procedural version, returns list"""
    i,a,b = 0,0,1
    list = []
    while i < n:
        list.append(b)
        i,a,b = i+1,b,a+b
    return list

>>> fib1(0)
[]
>>> fib1(1)
[1]
>>> fib1(12)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

## Fibonacci sequence: functional version

- This is a 'hack' (no lazy evaluation in Python), uses an auxiliary function and **list comprehension**

```
def fib2(n):
    """Functional version, returns list"""
    if n < 1:
        return []
    elif n == 1:
        return [1]
    l = [1,1]
    list = [1,1] + [oneStep(l) for x in range(1,n-1)]
    return list #map(lambda a: oneStep(l),range(1,n-1))

def oneStep(l):
    if len(l) == 2:
        a = l.pop(0)
        b = l.pop(0)
        l.append(b)
        l.append(a+b)
    return l[1]
```

## Fibonacci sequence: recursive version

- The **recursive call** inside the function body

```
def fib3(n):
    """Recursive version, returns list"""
    if n < 1:
        return []
    elif n == 1:
        return [1]
    elif n == 2:
        return [1,1]
    else:
        list = fib3(n-1)[: ]
        list.append(list[n-2] + list[n-3])
        return list
```

## What to do this week

- Read chapter 14 in text book (covers lectures CSTopics-3 to CSTopics-6)
- Work on home work 2
- The second lecture on Thursday (Oct. 23, 10am) will be a guest lecture by Peter Christen, *"Python in the Real World"*
- Complete and submit assignment 2
- Work on portfolio
- Start exam preparation (review next week)