Announcements

• Maps is out
  – No slip days for the checkpoint, but slip days for the rest of the project.
• Midterm, 3/16 7-9pm
  – Through OOP, but no inheritance
  – Make sure you’ve filled out the alternate times sheet.
Recursion

M. C. Escher: *Drawing Hands*
Demo: vee

• python3 –i 10-Recursion.py
• This uses Turtle Graphics.
  – The turtle module is really cool, but not something you need to learn
• vee is the one recursive problem that doesn't have a base case
  – But fractals in general are a fun way to visualize self-similar structures
• Use the following keys to play with the demo
  – Space to draw
  – C to Clear
  – Up to add “vee” to the functions list
  – Down to remove the “vee” functions from the list.
• Some cool variations on vee, seen in Snap! (the language of CS10)
Why Recursion?

- Recursive structures exist (sometimes hidden) in nature and therefore in data!
- It’s mentally and sometimes computationally more efficient to process recursive structures using recursion.
- Sometimes, the recursive definition is easier to understand or write, even if it is computationally slower.
Today: Recursion

- Recursive function calls itself, directly or indirectly
Demo: Countdown

def countdown(n):
    if n == 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n - 1)
The Recursive Process

Recursive solutions involve two major parts:

- **Base case(s)**, the problem is simple enough to be solved directly
- **Recursive case(s)**. A recursive case has three components:
  - **Divide** the problem into one or more simpler or smaller parts
  - **Invoke** the function (recursively) on each part, and
  - **Combine** the solutions of the parts into a solution for the problem.
Recursion
Learning Objectives

• Compare Recursion and Iteration to each other
  – Translate some simple functions from one method to another

• Write a recursive function
  – Understand the base case and a recursive case
palindromes

- Palindromes are the same word forwards and backwards.
- Python has some tricks, but how could we build this?
  - lambda p: p == p[::-1]
- Let’s write Reverse:

```python
def reverse(s):
    result = ''
    for letter in s:
        result = letter + result
    return result
```

```python
def reverse_while(s):
    ""
    >>> reverse_while('hello')
    'olleh'
    ""
    result = ''
    while s:
        first = s[0]
        s = s[1:] # remove the first letter
        result = first + result
    return result
```
reverse recursive

```python
def reverse(s):
    if not s:
        return ''
    return reverse(s[1:]) + s[0]

def palindrome(word):
    return word == reverse(word)
```

Recursive Case
Fun Palindromes

- racecar
- LOL
- radar
- a man a plan a canal panama
- aibohphobia 😈
  - The fear of palindromes.
- https://czechtheworld.com/best-palindromes/#palindrome-words
Iteration vs Recursion: Sum Numbers

For loop:

```python
def sum(n):
    s=0
    for i in range(0,n+1):
        s=s+i
    return s
```
Iteration vs Recursion: Sum Numbers

While loop:

```python
def sum(n):
    s=0
    i=0
    while i<n:
        i=i+1
        s=s+i
    return s
```
Recursion:

def sum(n):
    if n == 0:
        return 0
    return n+sum(n-1)
Iteration vs Recursion: Cheating!

Sometimes it’s best to just use a formula! But that’s not always the point. 😊

```python
def sum(n):
    return (n * (n + 1)) / 2
```
The Recursive Process

- Recursive solutions involve two major parts:
  - **Base case(s)**, the problem is simple enough to be solved directly
  - **Recursive case(s)**. A recursive case has three components:
    - **Divide** the problem into one or more simpler or smaller parts
    - **Invoke** the function (recursively) on each part, and
    - **Combine** the solutions of the parts into a solution for the problem.
Recall: Iteration

```python
def sum_of_squares(n):
    accum = 0
    for i in range(1, n+1):
        accum = accum + i*i
    return accum
```

1. Initialize the “base” case of no iterations
2. Starting value
3. Ending value
4. New loop variable value
Recursion Key concepts – by example

1. Test for simple “base” case
2. Solution in simple “base” case
3. Assume recursive solution to simpler problem
4. ”Combine” the simpler part of the solution, with the recursive case

```python
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```
**In words**

- The sum of no numbers is zero
- The sum of $1^2$ through $n^2$ is the
  - sum of $1^2$ through $(n-1)^2$
  - plus $n^2$

```python
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```
Why does it work

sum_of_squares(3)

# sum_of_squares(3) => sum_of_squares(2) + 3**2
# => sum_of_squares(1) + 2**2 + 3**2
# => sum_of_squares(0) + 1**2 + 2**2 + 3**2
# => 0 + 1**2 + 2**2 + 3**2 = 14
**Review: Functions**

- Generalizes an expression or set of statements to apply to lots of instances of the problem
- A function should *do one thing well*

```python
def def <function name> (<argument list> ) :
    return expression

def concat(str1, str2):
    return str1+str2;

concat("Hello","World")
```
How does it work?

• Each recursive call gets its own local variables
  – Just like any other function call
• Computes its result (possibly using additional calls)
  – Just like any other function call
• Returns its result and returns control to its caller
  – Just like any other function call
• The function that is called happens to be itself
  – Called on a simpler problem
  – Eventually stops on the simple base case
Questions

• In what order do we sum the squares?
• How does this compare to iterative approach?

```python
def sum_of_squares(n):
    accum = 0
    for i in range(1,n+1):
        accum = accum + i*i
    return accum
```

```python
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```

```python
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return n**2 + sum_of_squares(n-1)
```
Trust ...

- The recursive “leap of faith” works as long as we hit the base case eventually

What happens if we don’t?
Why Recursion?

• “After Abstraction, Recursion is probably the 2\textsuperscript{nd} biggest idea in this course”
• “It’s tremendously useful when the problem is self-similar”
• “It’s no more powerful than iteration, but often leads to more concise & better code”
• “It’s more ‘mathematical’”
• “It embodies the beauty and joy of computing”
• ...

UC Berkeley | Computer Science 88 | Michael Ball | http://cs88.org
Recursion (unwanted)
Example I

List all items on your hard disk

- Files
- Folders contain
  - Files
  - Folders

Recursion!
Another Example

- Recursion over sequence length, rather than number magnitude

```python
def first(s):
    """Return the first element in a sequence.""
    return s[0]

def rest(s):
    """Return all elements in a sequence after the first""
    return s[1:]

def min_r(s):
    """Return minimum value in a sequence.""
    if len(s) == 1:
        return first(s)
    else:
        return min(first(s), min_r(rest(s)))
```
Why Recursion? More Reasons

- Recursive structures exist (sometimes hidden) in nature and therefore in data!
- It’s mentally and sometimes computationally more efficient to process recursive structures using recursion.