Computational Structures in Data Science

Recursio
n

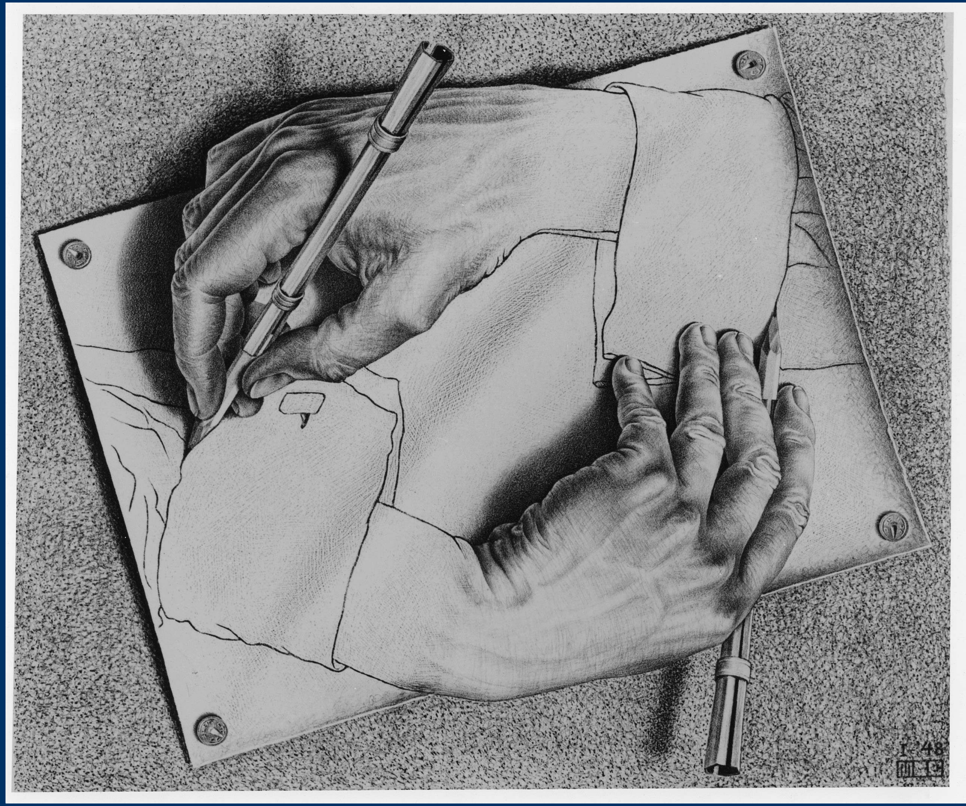
Announcements

- Midterm + Schedule updates
  - Midterm covers all material this week.
  - No lecture day after midterm
  - Next Monday: Review recursion + MT

Computational Structures in Data Science

Recursi
on

M. C. Escher : *Drawing Hands*



Demo: `vee` / Fractals

- `python3 -i 11-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)]
- [More Fractals]

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.
  - Fractals are definitely easily to think of recursively!

Today: Recursion

- Recursive function calls itself, directly or indirectly

re·cur·sion

/riˈkərZHən/

noun MATHEMATICS LINGUISTICS

the repeated application of a recursive procedure or definition.

- a recursive definition.
  plural noun: **recursions**

re·cur·sive

/riˈkərsiv/

adjective

characterized by recurrence or repetition, in particular.

- MATHEMATICS LINGUISTICS
  relating to or involving the repeated application of a rule, definition, or procedure to successive results.
- COMPUTING
  relating to or involving a program or routine of which a part requires the application of the whole, so that its explicit interpretation requires in general many successive executions.

Recursion In Practice

- We will use a function to solve smaller sub-problems
- Compared to a for-loop, while loop, we will *not* directly specify how many times we need to make a function call.

Demo: Countdown

```
def countdown(n):
    if n == 0:
        print('Blastoff!')
    else:
        # … what goes here?
```

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.

Computational Structures in Data Science

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?
  - `palindrome = lambda w: w == w[::-1]`
  - `[::-1]` is a slicing shortcut `[0:len(w):-1]` to reverse items.
- Let's write Reverse:

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

```
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
```

Fun Palindromes

- C88C
- racecar
- LOL
- radar
- a man a plan a canal panama
- aibohphobia 😈
  - The fear of palindromes.
- https://czechtheworld.com/best-palindromes/#palindrome-words

Writing Reverse Recursively

```
def reverse(s):
    if not s:
        return ''
    return 'TODO'

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

How should reverse work?

- Our algorithm in words:
  - Take the first letter, put it at the end
  - The beginning of the string is the reverse of the rest.

```
reverse('ABC')
→ reverse('BC') + 'A'
→ reverse('C') + 'B' + 'A
→ 'C' + 'B' + 'A
→  'CBA'
```

reverse recursive

```
def reverse(s):
    if not s:
        return ''
    return
```

Recursive Case

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

Iteration vs Recursion: Sum Numbers

For loop:

```
def sum(n):
    s=0
    for i in range(0,n+1):
        s=s+i
    return s
```

Iteration vs Recursion: Sum Numbers

While loop:

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

Iteration vs Recursion: Sum Numbers

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. ☺

```
def sum(n):
    return (n * (n + 1)) / 2
```

The Recursive Process

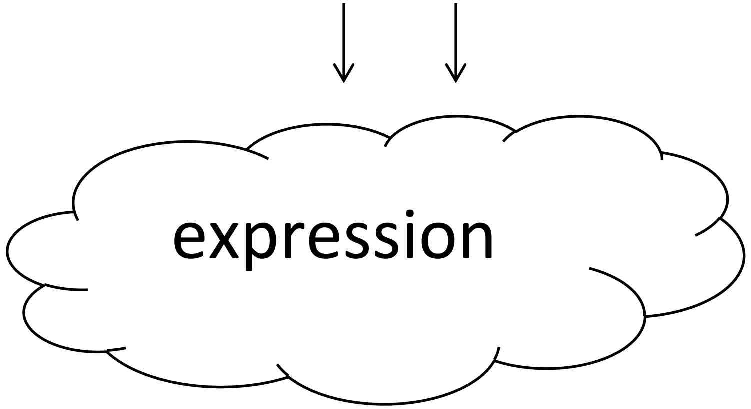
Recursive solutions involve two major parts:

- 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.

Review: Functions

def <function name> (<argument list>) :

return




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

concat("Hello","World")
```

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

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

Another Example

```
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
        Base Case
    else:
        Recursive Case
```

indexing an element of a sequence

Slicing a sequence of elements

- Recursion over sequence length

Recall: Iteration

1. Initialize the “base” case of no iterations

2. Starting value

3. Ending value

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

4. New loop variable value

Recursion Key concepts – by example

1. Test for simple “base” case

2. Solution in simple “base” case

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

3. Assume recusive solution to simpler problem

4. ”Combine” the simpler part of the solution, with the recursive case

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

```
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
```

Questions

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

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

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

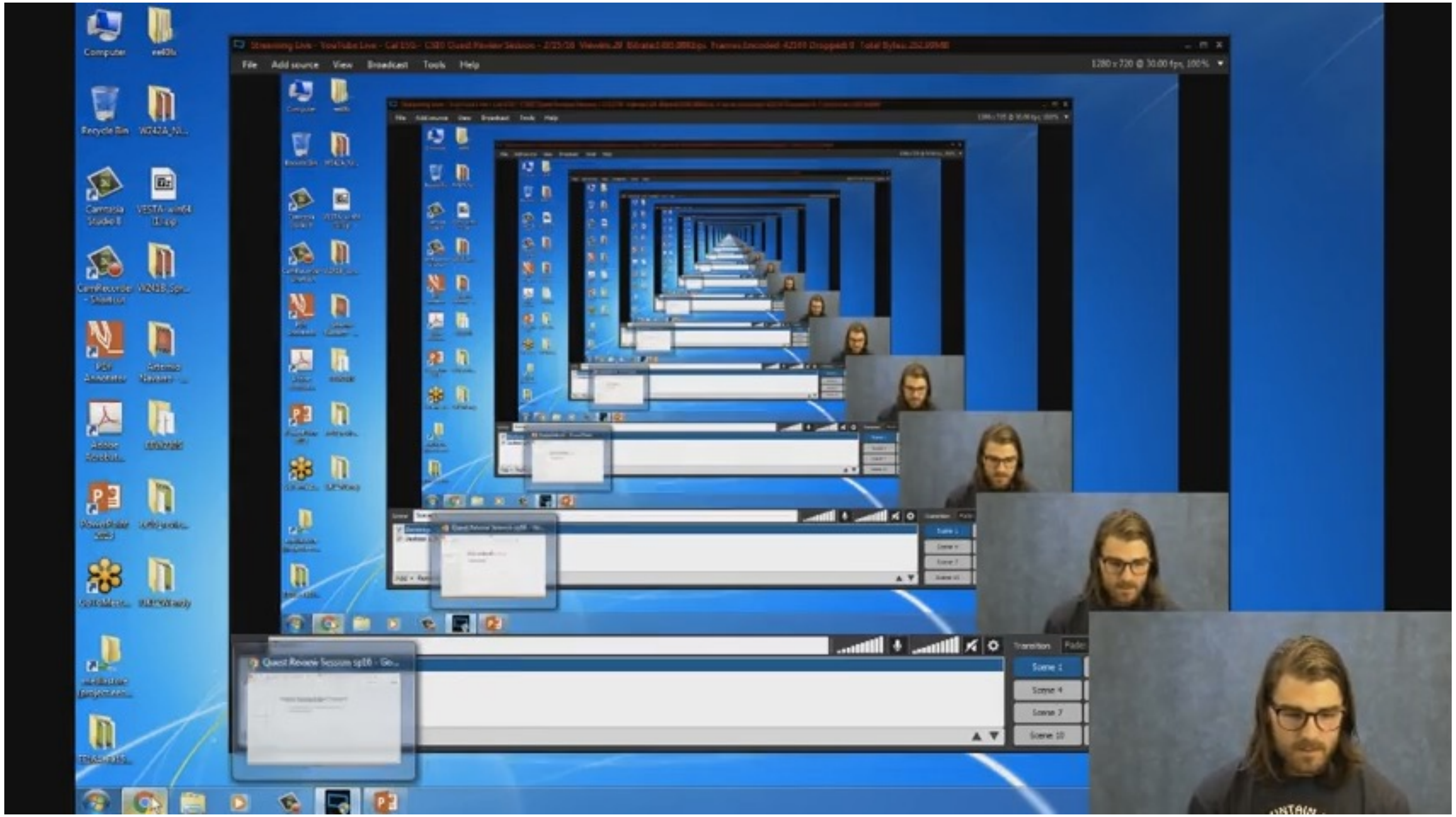
```
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return n**2 + sum_of_squares(n-1)
```

Trust …

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

- What happens if we don’t?

Recursion (unwanted)

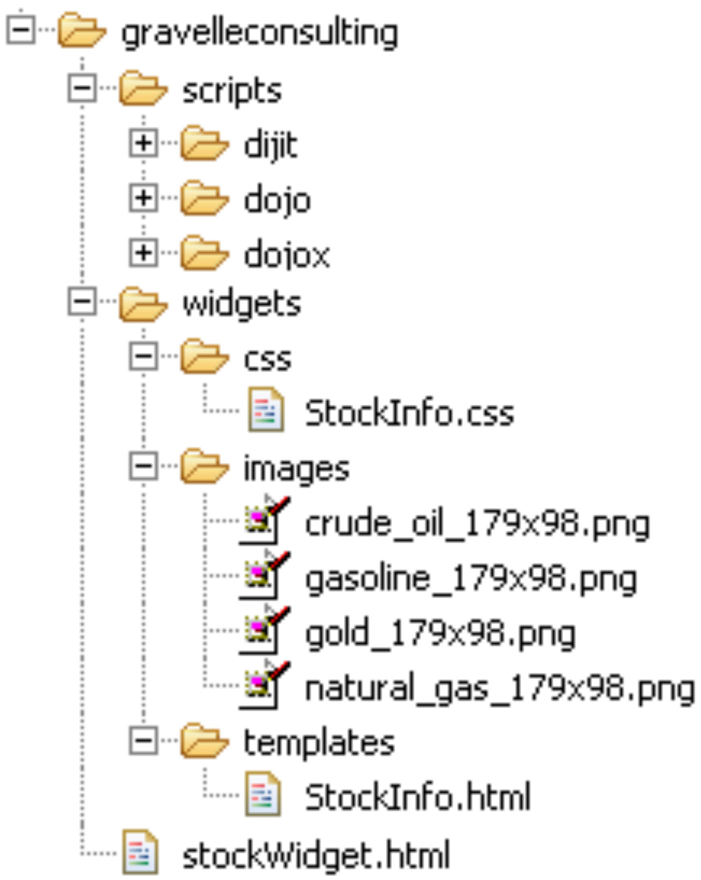


Why Recursion?

- “After Abstraction, Recursion is probably the 2nd 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”
- …

Example I

List all items on your hard disk



- Files
- Folders contain
  - Files
  - Folders

Recursion!

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.