map, reduce, and
Recursion == Happiness
Getting help…

Linde Activities Center (aka “The LAC”) Computer Lab on 2nd floor
Piazza (short Q&A)
Grutoring Times...

Look for CS 5 Black
Starting tonight 6-8 PM!
My office hours are on the syllabus...

CS 5 Course Syllabus, Spring 2020

Welcome to CS 5! This course is designed to give you a broad introduction to the exciting field of computer science.

Instructors

CS 5 Gold Julie Medero
- **Office**: Olin 1269
- **Email**: julie@cs.hmc.edu
- **Phone**: x18072
- **Office Hours**: Come anytime! Officially, see Prof. Medero's schedule

CS 5 Black Ran "RON" Libeskind-Hadas
- **Office**: Olin 1253
- **Email**: ran@cs.hmc.edu
- **Phone**: x18976
- **Office Hours**: Mondays 10-12:30 PM, Fridays 3-4 PM in Olin 1253

This Friday, 3:30-4 PM
Today’s True Story
Mapping with Python...

```python
def dbl(x):
    """returns 2 * x """
    return 2 * x

>>> list(map(dbl, [0, 1, 2, 3, 4]))
[0, 2, 4, 6, 8]

def evens(n):
    """returns list of first n evens """
    myList = range(n)  # returns list[0, 1, ..., n-1]
    doubled = list(map(dbl, myList))
    return doubled

Alternatively…

def evens(n):
    return list(map(dbl, range(n)))
```
from functools import reduce

def add(x, y):
    """ returns x + y """
    return x + y

>>> reduce(add, [1, 2, 3, 4])
10
Google’s “Secret”

This is what put Google on the map!

MapReduce: Simplified Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat

Abstract

MapReduce is a programming model and an associated implementation for processing intermediate key/value pairs, and a reduce function that merges all intermediate values

Programs written in this functional style are automatically parallelized and executed on scheduling the program's execution across a set of machines, handling machine failure
Write a function called `span` that uses `reduce`, `min`, `max` and returns the difference between the maximum and minimum numbers *in a list*…

```python
>>> span([3, 1, 42, 7])
41
>>> span([42, 42, 42, 42])
0
```
Try this in your worksheet...

Write a function called `span` that returns the difference between the maximum and minimum numbers in a list...

```python
>>> span([3, 1, 42, 7])
41
>>> span([42, 42, 42, 42])
0
```

```python
from functools import reduce

def span1(myList):
    '''returns difference between largest and smallest element in list'''
    smallest = reduce(min, myList)
    largest = reduce(max, myList)
    output = largest - smallest
    return output

def span2(myList):
    '''returns difference between largest and smallest element in list'''
    return reduce(max, myList) - reduce(min, myList)
```
1. Write a python function called \texttt{gauss}(n) that accepts a positive integer \(n\) and returns the sum \(1 + 2 + \ldots + n\)

2. Write a python function called \texttt{sumSquares}(n) that accepts a positive integer \(n\) and returns the sum \(1^2 + 2^2 + 3^2 + \ldots + n^2\)

You can use \texttt{add} and write other “helper” functions!

\begin{verbatim}
def add(x, y):
    """ returns x + y """
    return x + y
\end{verbatim}
def gauss(n):
    """ return sum of numbers 1 through n """
    return reduce(add, range(n+1))

def sumSquares(n):
    return reduce(add, map(square, range(n+1)))

def square(x):
    return x * x

def add(x, y):
    return x+y
Life inside a function…

```python
def h(x):
    return f(x) + x

def f(x):
    x = x - 1
    return g(x) + 1

def g(x):
    return x * 2
```

Two key points…

- Functions return to where they were called from
- Each function keeps its own values of its variables
Factorial with loops...

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 1 \]

def factorial(n):
    result = 1
    for k in range(1, n+1):
        result = result * k
    return result

No recursion needed here, but some problems desperately need recursion!
Recursion…

\[ n! = n \times (n-1) \times (n-2) \times ... \times 1 \]

\[ n! = n \times (n-1)! \quad \text{“inductive definition”} \]

\[ 0! = 1 \quad \text{“base case”} \]

Why is \( 0! = 1 \)?
Math Induction = CS Recursion

**Math**

inductive definition

\[
\begin{align*}
0! & = 1 \\
n! & = n \times (n-1)! \\
\end{align*}
\]

**Python (Functional)**

recursive function

```python
# recursive factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```
Is Recursion Magic?

factorial(3):
    return 3 * factorial(2)
Is Recursion Magic?

```python
# recursive factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

factorial(3):
    return 3 * factorial(2)
    return 2 * factorial(1)
```

```python
# recursive factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```
Is Recursion Magic?

factorial(3):
    return 3 * factorial(2)

    return 2 * factorial(1)

    return 1 * factorial(0)

# recursive factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

# recursive factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
Is Recursion Magic?

factorial(3):

return 3 * factorial(2)

return 2 * factorial(1)

return 1 * factorial(0)

# recursive factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
Is Recursion Magic?

factorial(3):

return 3 * factorial(2)

return 2 * factorial(1)

return 1 * factorial(0)

# recursive factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
Is Recursion Magic?

```python
# recursive factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

factorial(3):
    return 3 * factorial(2)
    return 2 * factorial(1)
    return 1 * factorial(0)

# recursive factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

```
6  
↑   2
↑   1  
↑

return 3 * factorial(2)

return 2 * factorial(1)

return 1 * factorial(0)
```
A Tower of Fun!

Math

tower(3) = 2^2^2

tower(4) = 2^2^2^2

tower(5) = 2^2^2^2^2

inductive definition:

inductive case:
\[ \text{tower}(n) = 2^{\text{tower}(n-1)} \]
base case:
\[ \text{tower}(0) = ? \]

Python (Functional)

recursive function

```python
# recursive tower
def tower(n):
```

Try this on your worksheet!
(Note that \(x^y\) is computed using \(x**y\))
A Tower of Fun!

**Math**

tower(3) = \( 2^{2^2} \)
tower(4) = \( 2^{2^{2^2}} \)
tower(5) = \( 2^{2^{2^{2^2}}} \)

**Python (Functional)**

```python
# recursive tower
def tower(n):
    ''' Tower of n '''
    if n==0:
        return 1
    else:
        # recursive definition
        tower(n) = 2^{tower(n-1)}

inductive definition:
    inductive case:
        tower(n) = 2^{tower(n-1)}
    base case:
        tower(0) = 1
```

recursive function
A Tower of Fun!

**Math**

\[
tower(3) = 2^{2^2}
\]

\[
tower(4) = 2^{2^{2^2}}
\]

\[
tower(5) = 2^{2^{2^{2^2}}}
\]

**Python (Functional)**

```
# recursive function
def tower(n):
    ''' Tower of n '''
    if n==0:
        return 1
    else:
        return 2**tower(n-1)
```

**Inductive definition:**

**Inductive case:**

\[
tower(n) = 2^{tower(n-1)}
\]

**Base case:**

\[
tower(0) = 1
\]

Demo!
Lists!

Python starts counting from 0…

```python
>>> M = [2, "alien", 42, ["spam","spamity","spam"] ]
>>> len(M)
4
>>> M[2]
42
>>> M[3]
['spam', 'spamity', 'spam']
>>> M[3][0]
???
>>> M[1:]
["alien", 42, ["spam","spamity","spam"] ]
```
Recursion on lists

```python
>>> len([1, 42, "spam"])
3
>>> len([1, [2, [3, 4]]])
2

def len(L):
    """ returns the length of list L """
```
Computing the length of a list

>>> len([1, 42, "spam"])
3

>>> len([1, [2, [3, 4]]])
2

def len(L):
    """ returns the length of list L """
    if L == []: return 0  # base case
Computing the length of a list

```python
>>> len([1, 42, "spam"])
3
>>> len([1, [2, [3, 4]]])
2

def len(L):
    """returns the length of list L """
    if L == []: return 0  # base case
    else: return 1 + len(L[1:])
```

What happens if I run `len(42)`?
Summing up the numbers in a list

```python
>>> sum([3, 42, 7])
52
>>> sum([42])
42
>>> sum([])
0

def sum(L):
    """returns the sum of numbers in the list"""
```

Python has this built-in too!
Summing up the numbers in a list

```python
>>> sum([1, 42, 7])
50
>>> sum([42])
42
>>> sum([])
0
```

```python
def sum(L):
    """returns the sum of numbers in the list""
    if L == []: return 0
```

The base case takes care of the simplest case we could get.
Summing up the numbers in a list

```python
>>> sum([1, 42, 7])
50
>>> sum([42])
42
>>> sum([])
0
```

def sum(L):
    """returns the sum of numbers in the list""
    if L == []: return 0
    else: return ???
```

Try finishing this in your notes (not worksheet)…
Summing up the numbers in a list

```python
>>> sum([1, 42, 7])
50
>>> sum([42])
42
>>> sum([])
0

def sum(L):
    """returns the sum of numbers in the list""
    if L == []: return 0
    else: return L[0] + sum(L[1:])
```
More tricks with lists...

```python
>>> myList = [42, 10, 5, "spam"]
    0  1  2  3
    -4 -3 -2 -1

>>> myList[0]
42

>>> myList[-4]
42

>>> myList[-1]
"spam"

>>> myList[100]
IndexError: list index out of range
```
More tricks with lists...

```python
>>> myList = [42, 10, 5, "spam"]
```

```plaintext
0   1   2   3
```

```plaintext
-4  -3  -2  -1
```

```python
>>> myList[0:3]  # stops one short of 3
[42, 10, 5]
>>> myList[0:-1]  # stops one short of -1
[42, 10, 5]
>>> myList[1:]  # from 1 to the end
[10, 5, "spam"]
>>> myList[:2]  # from start to one short of 2
[42, 10]
```
More tricks with lists...

```python
>>> myList = [42, 10, 5, "spam"]
```

```plaintext
-4  -3  -2  -1
```

```python
>>> myList[100]
IndexError: list index out of range
```

```python
>>> myList[1:100]
[10, 5, "spam"]
```

You can’t get hurt when you slice!*

(*In Python)
Adding lists

```python
>>> myList = [42, 47, 23]
>>> newList = myList + 100
BARF!
>>> newList = myList + [100]
>>> newList
[42, 47, 23, 100]
>>> myList
[42, 47, 23]
>>> newList = newList + newList
>>> newList
[42, 47, 23, 100, 42, 47, 23, 100]
```
Adding lists

```python
>>> myList = [42, 47, 23]
>>> newList = myList + 100
BARF!
>>> newList = myList + [100]
>>> newList
[42, 47, 23, 100]
>>> myList
[42, 47, 23]
>>> newList = newList + newList
>>> newList
[42, 47, 23, 100, 42, 47, 23, 100]
```
Adding lists

```python
>>> myList = [42, 47, 23]
>>> newList = myList + 100
BARF!
>>> newList = myList + [100]
>>> newList
[42, 47, 23, 100]
>>> myList
[42, 47, 23]
>>> newList = newList + newList
>>> newList
[42, 47, 23, 100, 42, 47, 23, 100]
```
Reversing a list

```python
>>> reverse([1, 2, 3, 4])
[4, 3, 2, 1]

def reverse(L):
```

Reversing a list

```python
>>> reverse([1, 2, 3, 4])
[4, 3, 2, 1]

def reverse(L):
    if L == []: return ???  # base case
```
Reversing a list

```python
>>> reverse([1, 2, 3, 4])
[4, 3, 2, 1]

def reverse(L):
    if L == []: return []  # base case
else:
    return ???
```

Try this in your notes (not worksheet)
Reversing a list

```python
>>> reverse([1, 2, 3, 4])
[4, 3, 2, 1]

def reverse(L):
    if L == []: return []  # base case
    else:
        return reverse(L[1:]) + [L[0]]
```

Why the red brackets?

Notice the indentation here…
Something that’s (very) hard to do \textit{without} recursion…

```python
>>> reverse([1, [2, [4, 5], 6], 7])
[7, [2, [4, 5], 6], 1]

>>> deepReverse([1, [2, [4, 5], 6], 7])
[7, [6, [5, 4], 2], 1])
```

```python
>>> removeAll(42, [67, 42, [42, [42, 43]], 47])
[67, [42, [42, 43]], 47]

>>> deepRemoveAll(42, [67, 42, [42, [42, 43]], 47])
[67, [[43]], 47]
```
Recursion = :^)\

“To understand recursion, you must first understand recursion” - anonymous Mudd alum
recursion

Web Images Videos Shop

About 1,860,000 results (0.22 seconds)

Did you mean: recursion
Strings!

```python
>>> myDNA = "AATGCCGTGCTT"
>>> len(myDNA)  # built-in len function!
12

>>> myDNA[0]
'A'

>>> myDNA[3]
'G'

>>> myDNA[20]
IndexError: string index out of range
```

Single quotes and double quotes work equally well, as long as we don’t mix-and-match!
Strings...

```python
>>> myDNA = "AATGCCGTGCTT"

>>> myDNA[0:4]
'AATG'

>>> myDNA[3:7]
'GCCG'

>>> myDNA[1:]
'ATGCCGTGCTT'

>>> myDNA[:4]
'AATG'

>>> myDNA[10:42]
'TT'

>>> foo = ''
>>> foo[1:]
''
```
Palindrome?

>>> pal("radar")
True
>>> pal("amanaplanacanalpanama")
True
>>> pal("spam")
False

def pal(myString):
    
    
    # base case?
    # general case?

A solution is on the next slide, but try to avert your eyes for a moment!
>>> pal("radar")
True
>>> pal("amanaplanacanalpanama")
True
>>> pal("spam")
False

def pal(myString):
    """ returns True if myString is a palindrome and False otherwise """
    if len(myString) <= 1: return True # base case
    elif myString[0] != myString[-1]:
        return False
    else: return pal(myString[1:-1])