Lecture 07
A First Look at Recursion

It's a baby holding a smaller baby! If that ain't recursion idk what is. Oh wait.. I don't know what recursion is…. what's that?

based in part on notes from the CS-for-All curriculum developed at Harvey Mudd College and Boston University

Last Time (lecture 06)

- Methods (functions attached to objects)
  - `string.lower()`
  - `file.close()`
- File Processing
  - Opening and Closing Text-based Files
  - Reading 1 line vs. whole file
- Splitting strings
  - `s.split()  # for whitespace`
  - `s.split(',')  # for CSV files`
- Dictionaries
  - `data = {'key1': value1, 'key2': value2}`
  - `data[key1] = value1`

- Markov models and Project 1
  - **due Thursday, Feb. 28 @ midnight**

Lecture 07 Goals

1. Introduce recursion design process
   a. function calling itself!
2. Application of Test Driven Design (TDD) to recursion

3. As time allows:
   a. **lambda** functions
   b. **filter**, **map**, & **reduce**
   c. debugging using **pdb** (Python debugger)

Remember this slide?

recursion

```
def fac(n):
    if n == 0:
        return 1
    else:
        rest = fac(n-1)
        return n * rest
```

More on these ~~later~~ RIGHT NOW!

for loop

```
def fac(n):
    result = 1
    for x in range(1, n+1):
        result *= x
    return result
```

map

```
def fac(n):
    return reduce(lambda x,y : x*y,\
       range(1,max(2,n+1)))
```

while loop

```
def fac(n):
    result = 1
    while n > 0:
        result *= n
        n = n - 1
    return result
```

Functions Calling Themselves: *Recursion!*

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

Remember this function? factorial?

- Recursion solves a problem by reducing it to a *simpler* or *smaller* problem *of the same kind*.
  - the function calls itself to solve the smaller problem!

- We take advantage of *recursive substructure*.
  - the fact that we can define the problem *in terms of itself*

```
n! = n * (n-1)!
```

Functions Calling Themselves: *Recursion!* (cont.)

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

base case

recursive case

- One recursive call leads to another...

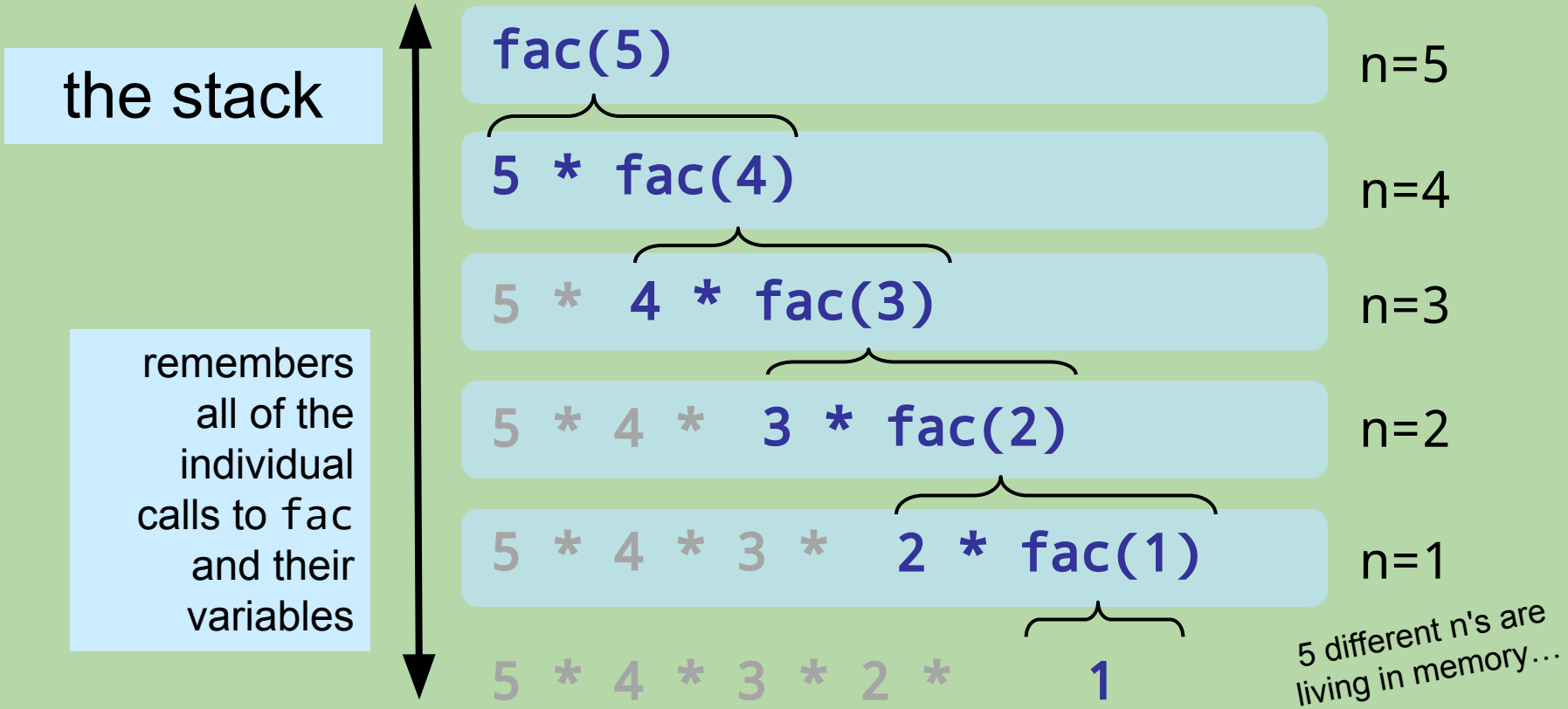
```
fac(5) = 5 * fac(4)
       = 5 * 4 * fac(3)
       = ...
```

- We eventually reach a problem that is small enough to be solved directly – a *base case.*
  - stops the recursion
  - make sure that you always include one!

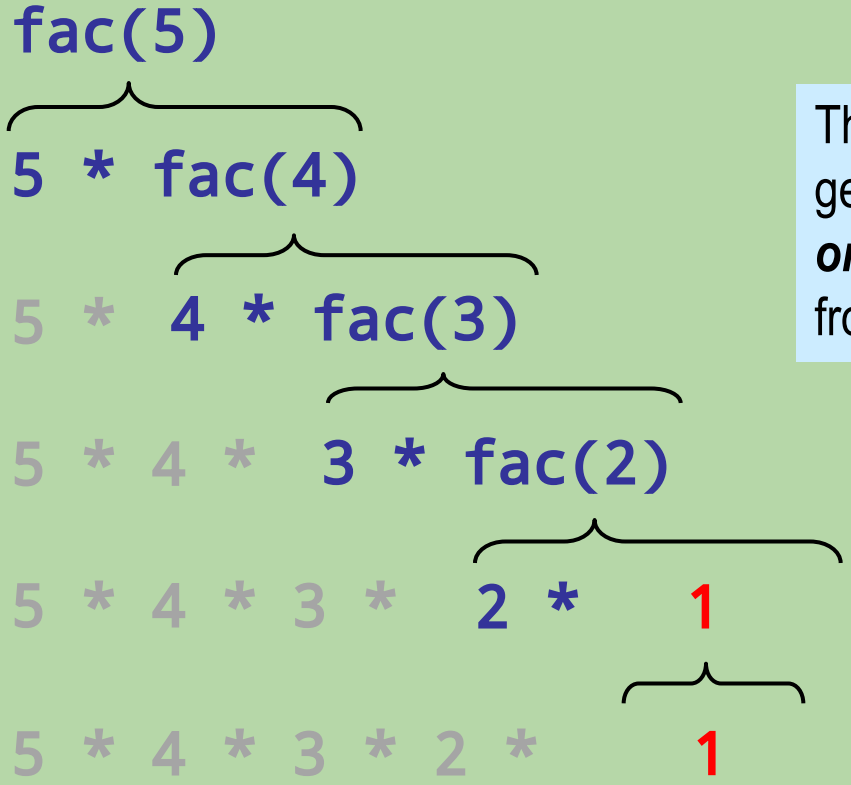
Recursion Without a Base Case – infinite loop!

http://blog.stevemould.com/the-droste-effect-image-recursion/

```
def fac(n):

    if n <= 1:
        return 1

    else:
        return n * fac(n-1)
```

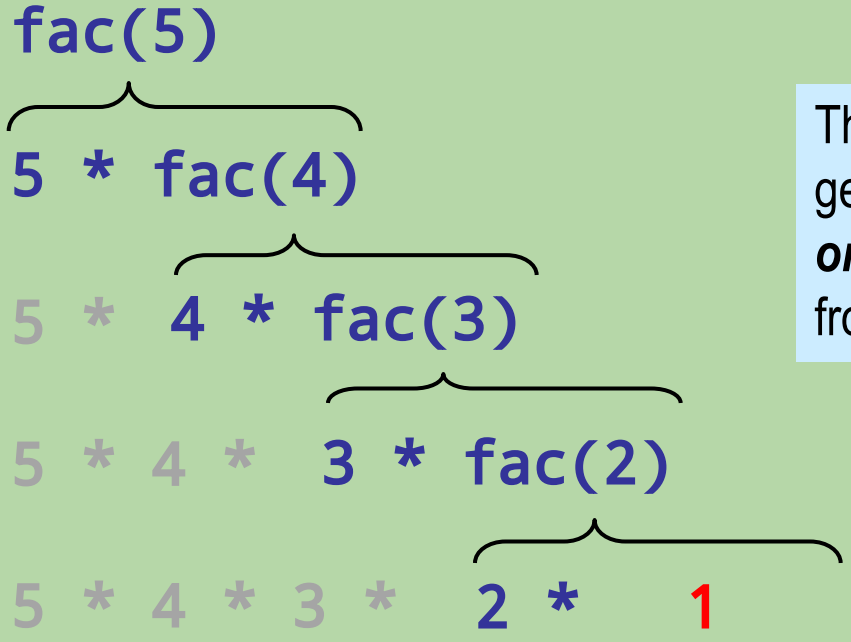


```
def fac(n):

    if n <= 1:
        return 1

    else:
        return n * fac(n-1)
```



The final result gets built up ***on the way back*** from the base case!

```
def fac(n):

    if n <= 1:
        return 1

    else:
        return n * fac(n-1)
```

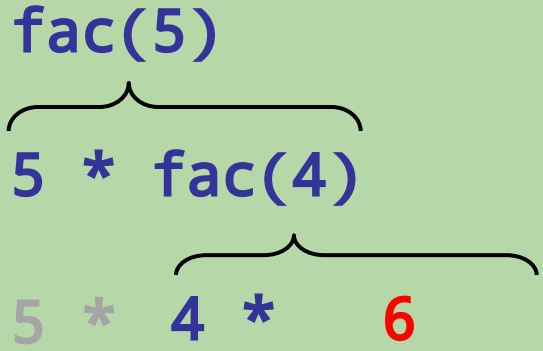


The final result gets built up ***on the way back*** from the base case!

```
def fac(n):

    if n <= 1:
        return 1

    else:
        return n * fac(n-1)
```

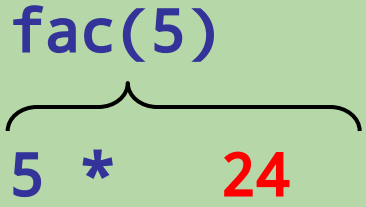
The final result gets built up ***on the way back*** from the base case!

```
def fac(n):

    if n <= 1:
        return 1

    else:
        return n * fac(n-1)
```



The final result gets built up ***on the way back*** from the base case!

```
def fac(n):

    if n <= 1:
        return 1

    else:
        return n * fac(n-1)
```



The final result gets built up ***on the way back*** from the base case!

```
def fac(n):

    if n <= 1:
        return 1

    else:
        return n * fac(n-1)
```

fac(5)

result: 120

Alternative Version of fac(n)

```
def fac(n):
    if n <= 1:
        return 1

    else:
        rest = fac(n - 1)
        return n * rest
```

- Storing the result of the recursive call will occasionally make the problem easier to solve.
- It also makes your recursive functions easier to trace and debug.
- ***We highly recommend that you take this approach when debugging!***

Let Recursion Do the Work For You!

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

You handle the base case – the easiest case!

Recursion does almost all of the rest of the problem!

You specify one step at the end.

Recursively Processing a List or String

- You can think about (and process) sequences recursively!
  - a string is a character followed by a string...
  - a list is an element followed by a list...

- Let `s` be the sequence (string or list) that we're processing.

- Do one step!
  - use `s[0]` to access the initial element
  - do something with it

- Delegate the rest!
  - use `s[1:]` to get the rest of the sequence.
  - make a recursive call to process it!

Recursively Finding the Length of a String

```
def mylen(s):
    """ returns the number of characters in s
        input s: an arbitrary string
    """
    if s == '':                 # base case


    else:                       # recursive case
```

- Ask yourself:

(base case) When can I determine the length of `s` *without* looking at a smaller string?

(recursive substructure) How could I use the length of ***anything smaller*** than `s` to determine the length of `s`?

Recursively Finding the Length of a String

```
def mylen(s):
    """ returns the number of characters in s
        input s: an arbitrary string
    """
    if s == '':                 # base case
        return 0

    else:                       # recursive case
        len_rest = mylen(s[1:])
        return len_rest + 1
```

- Ask yourself:

(base case) When can I determine the length of `s` *without* looking at a smaller string?

(recursive substructure) How could I use the length of ***anything smaller*** than `s` to determine the length of `s`?

How recursion works...

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

```
mylen('wow')
    s = 'wow'
    len_rest = mylen('ow')
```

```
mylen('ow')
    s = 'ow'
    len_rest = mylen('w')
```

```
mylen('w')
    s = 'w'
    len_rest = mylen('')
```

```
mylen('')
    s = ''
    base case!
    return 0
```

4 different stack frames, each with its own `s` and `len_rest`

How recursion works...

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

```
mylen('wow')
    s = 'wow'
    len_rest = mylen('ow')
```

```
mylen('ow')
    s = 'ow'
    len_rest = mylen('w')
```

```
mylen('w')
    s = 'w'
    len_rest = mylen('') = 0
```

```
mylen('')
    s = ''
    base case!
    return 0
```

4 different stack frames, each with its own `s` and `len_rest`

How recursion works...

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

```
mylen('wow')
    s = 'wow'
    len_rest = mylen('ow')
```

```
mylen('ow')
    s = 'ow'
    len_rest = mylen('w')
```

```
mylen('w')
    s = 'w'
    len_rest = mylen('') = 0
    return 0 + 1 = 1
```

len_rest

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

```
mylen('wow')
    s = 'wow'
    len_rest = mylen('ow')
```

```
mylen('ow')
    s = 'ow'
    len_rest = mylen('w') = 1
```

```
mylen('w')
    s = 'w'
    len_rest = mylen('') = 0
    return 0 + 1 = 1
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
mylen('wow')
   s = 'wow'
   len_rest = mylen('ow')
```

```
mylen('ow')
   s = 'ow'
   len_rest = mylen('w') = 1
   return 1 + 1 = 2
```

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
mylen('wow')
   s = 'wow'
   len_rest = mylen('ow') = 2
```

```
mylen('ow')
   s = 'ow'
   len_rest = mylen('w') = 1
   return 1 + 1 = 2
```

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
mylen('wow')
   s = 'wow'
   len_rest = mylen('ow') = 2
   return 2 + 1 = 3
```

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
mylen('wow')
   s = 'wow'
   len_rest = mylen('ow') = 2
   return 2 + 1 = 3
```

result: **3**

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

How many times will `mylen()` be called?

```
def mylen(s):
    if s == '':                 # base case
        return 0
    else:                       # recursive case
        len_rest = mylen(s[1:])
        return len_rest + 1

print(mylen('step'))
```

A. 1

B. 3

C. 4

D. 5

E. 6

How many times will `mylen()` be called?

```
def mylen(s):
    if s == '':                 # base case
        return 0
    else:                       # recursive case
        len_rest = mylen(s[1:])
        return len_rest + 1

print(mylen('step'))
```

A. 1

B. 3

C. 4

D. **5**

E. 6

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

```
mylen('step')
  s = 'step'
  len_rest = mylen('tep')
```

```
mylen('tep')
  s = 'tep'
  len_rest = mylen('ep')
```

```
mylen('ep')
  s = 'ep'
  len_rest = mylen('p')
```

```
mylen('p')
  s = 'p'
  len_rest = mylen('')
```

```
mylen('')
  s = ''
  base case!
  return 0
```

5 different stack frames, each with its own `s` and `len_rest`

```
mylen('step')
  s = 'step'
  len_rest = mylen('tep')
```

```
mylen('tep')
  s = 'tep'
  len_rest = mylen('ep')
```

```
mylen('ep')
  s = 'ep'
  len_rest = mylen('p')
```

```
mylen('p')
  s = 'p'
  len_rest = mylen('') = 0
```

```
mylen('')
  s = ''
  base case!
  return 0
```

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

```
mylen('step')
  s = 'step'
  len_rest = mylen('tep')
```

```
mylen('tep')
  s = 'tep'
  len_rest = mylen('ep')
```

```
mylen('ep')
  s = 'ep'
  len_rest = mylen('p') = 1
```

```
mylen('p')
  s = 'p'
  len_rest = mylen('') = 0
  return 0 + 1 = 1
```

len_rest

The final result gets built up ***on the way back*** from the base case!

```
mylen('step')
  s = 'step'
  len_rest = mylen('tep')
```

```
mylen('tep')
  s = 'tep'
  len_rest = mylen('ep') = 2
```

```
mylen('ep')
  s = 'ep'
  len_rest = mylen('p') = 1
  return 1 + 1 = 2
```

len_rest

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

The final result gets built up ***on the way back*** from the base case!

```
mylen('step')
  s = 'step'
  len_rest = mylen('tep') = 3
```

```
mylen('tep')
  s = 'tep'
  len_rest = mylen('ep') = 2
  return 2 + 1 = 3
```

len_rest

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

The final result gets built up ***on the way back*** from the base case!

```
mylen('step')
   s = 'step'
   len_rest = mylen('tep') = 3
   return 3 + 1 = 4
```

result: **4**

```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

The final result gets built up ***on the way back*** from the base case!

What is the output of this program?

```
def foo(x, y):
    if x <= y:
        return y
    else:
        return x + foo(x - 2, y + 1)

print(foo(9, 2))
```

A. 2

B. 4

C. 5

D. 21

E. 26

What is the output of this program?

```
def foo(x, y):
    if x <= y:
        return y
    else:
        return x + foo(x - 2, y + 1)

print(foo(9, 2))
```

A. 2

B. 4

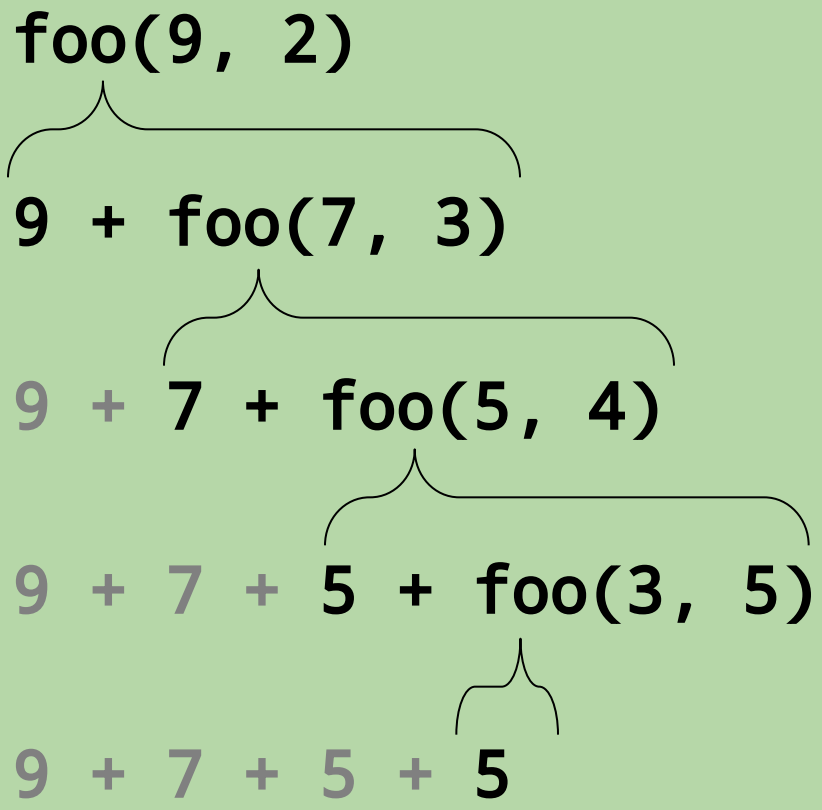
C. 5

D. 21

E. 26

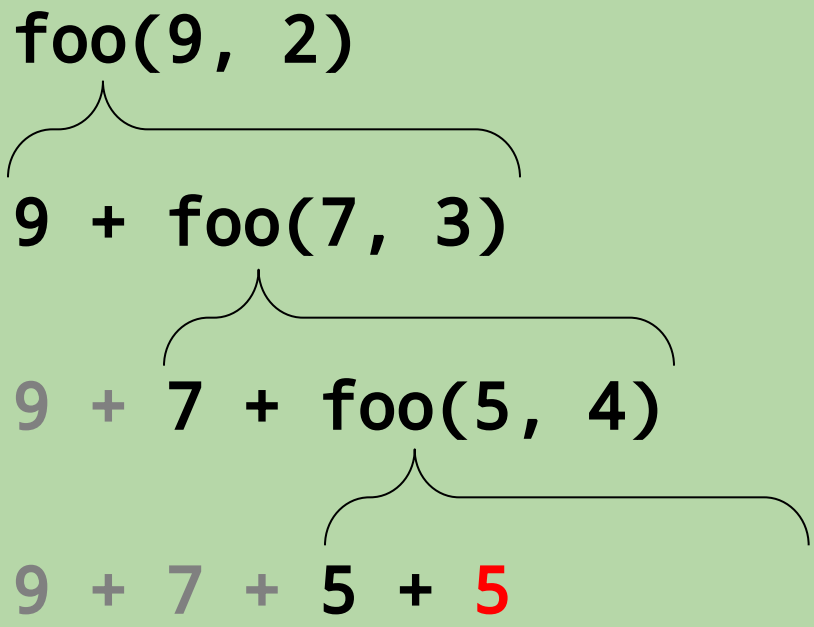
How recursion works...

```
def foo(x, y):
    if x <= y:
        return y
    else:
        return x + foo(x-2, y+1)
```



How recursion works...

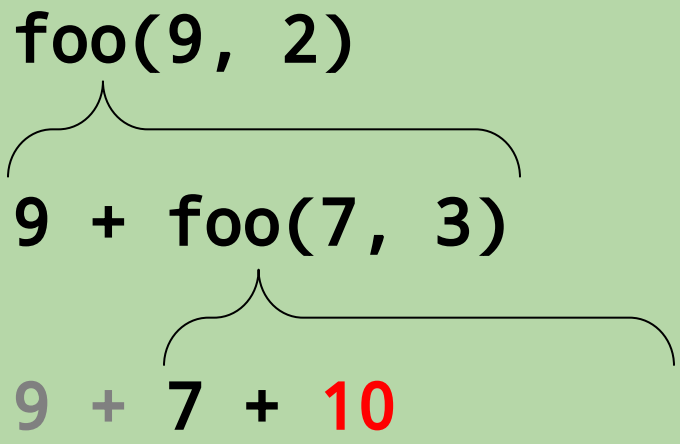
```
def foo(x, y):
    if x <= y:
        return y
    else:
        return x + foo(x-2, y+1)
```



The final result gets built up ***on the way back*** from the base case!

How recursion works...

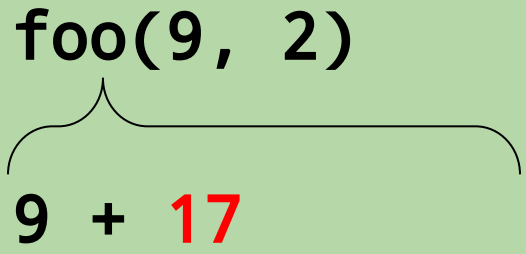
```
def foo(x, y):
    if x <= y:
        return y
    else:
        return x + foo(x-2, y+1)
```



The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def foo(x, y):
    if x <= y:
        return y
    else:
        return x + foo(x-2, y+1)
```



The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def foo(x, y):
    if x <= y:
        return y
    else:
        return x + foo(x-2, y+1)
```

foo(9, 2)

result: **26**

Designing a Recursive Function

1. Use Test Driven Design and then

2. Start by programming the base case(s) and testing.
   - *What instance(s) of this problem can I solve directly (without looking at anything smaller)?*

3. Find the recursive substructure.
   - *How could I use the solution to **any smaller version** of the problem to solve the overall problem?*

4. Do one step!

5. Delegate the rest to recursion!

A Recursive Function for Counting Vowels

```
def num_vowels(s):
    """ returns the number of vowels in s
        input s: a string of lowercase letters
    """
    # We'll design this together!
```

- Examples of how it should work:

```
>>> num_vowels('compute')
3
>>> num_vowels('now')
1
```

- The `in` operator will be helpful:

```
>>> 'fun' in 'function'
True
>>> 'i' in 'team'
False
```

Test Driven Design Steps

1. Inputs/Outputs, special cases
2. Function signature
3. Design test cases, then code function
4. Refined testing as coding proceeds

Test Driven Design Steps

1. Inputs/Outputs, Special Cases

   Returns number of vowels in a string argument

   - s empty, i.e. s=‘’
   - s with one vowel
   - s with more than one vowel
   - s with vowel at beginning or end

2. Function Signature

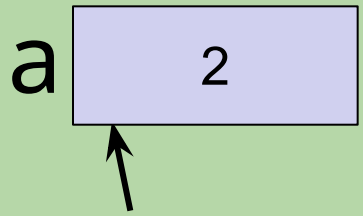
```
def num_vowels(s):
    """ returns the number of vowels in s
        input s: a string of lowercase letters
    """
```

3. Test Cases

Design Questions for num_vowels()

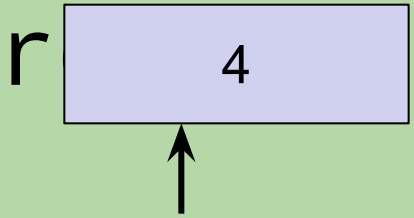
(base case) When can I determine the # of vowels in `s` *without* looking at a smaller string?

(recursive substructure) How could I use the solution to ***anything smaller*** than `s` to determine the solution to `s`?



You can only see the first letter of this string.

If I told you the # of vowels in the covered portion, how would you determine the total number of vowels?

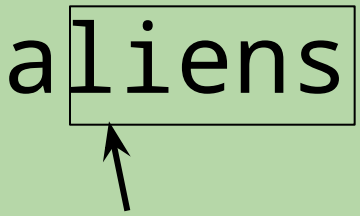


What about this string?

Design Questions for num_vowels()

(base case) When can I determine the # of vowels in `s` *without* looking at a smaller string?

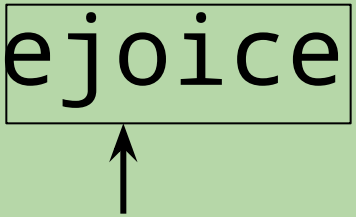
(recursive substructure) How could I use the solution to ***anything smaller*** than `s` to determine the solution to `s`?



You can only see the first letter of this string.

If I told you the # of vowels in the covered portion, how would you determine the total number of vowels?

`1 + (# in covered)`



What about this string?

`0 + (# in covered)`

The recursive call gives us `(# in covered)`!!!

How Many Lines of This Function Have a Bug?

```
def num_vowels(s):
    if s == '':
        return 0
    else:
        rest = num_vowels(s[0:])
        if s[0] in 'aeiou':
            return 1
        else:
            return 0
```

A. 0

B. 1

C. 2

D. 3

E. more than 3

How Many Lines of This Function Have a Bug?

```
def num_vowels(s):
    if s == '':
        return 0
    else:
        rest = num_vowels(s[1:])
        if s[0] in 'aeiou':
            return 1 + rest
        else:
            return 0 + rest
```

A. 0

B. 1

C. 2

D. **3**

E. more than 3

How recursion works...

```
def num_vowels(s):
    if s == '':
        return 0
    else:
        rest = num_vowels(s[1:])
        if s[0] in 'aeiou':
            return 1 + rest
        else:
            return 0 + rest
```

num_vowels('ate')

```
s = 'ate'
rest = num_vowels('te')
```

num_vowels('te')

```
s = 'te'
rest = num_vowels('e')
```

num_vowels('e')

```
s = 'e'
rest = num_vowels('')
```

num_vowels('')

```
s = ''
base case!
return 0
```

4 different stack frames, each with its own `s` and `rest`

How recursion works...

```
def num_vowels(s):
    if s == '':
        return 0
    else:
        rest = num_vowels(s[1:])
        if s[0] in 'aeiou':
            return 1 + rest
        else:
            return 0 + rest
```

num_vowels('ate')

```
s = 'ate'
rest = num_vowels('te')
```

num_vowels('te')

```
s = 'te'
rest = num_vowels('e')
```

num_vowels('e')

```
s = 'e'
rest = num_vowels('') = 0
```

num_vowels('')

```
s = ''
base case!
return 0
```

4 different stack frames, each with its own `s` and `rest`

How recursion works...

```
num_vowels('ate')
   s = 'ate'
   rest = num_vowels('te')
```

```
num_vowels('te')
   s = 'te'
   rest = num_vowels('e')
```

```
num_vowels('e')
   s = 'e'   s[0] -> 'e'
   rest = num_vowels('') = 0
   return 1 + 0 = 1
```

rest

```
def num_vowels(s):
    if s == '':
        return 0
    else:
        rest = num_vowels(s[1:])
        if s[0] in 'aeiou':
            return 1 + rest
        else:
            return 0 + rest
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
num_vowels('ate')
   s = 'ate'
   rest = num_vowels('te')
```

```
num_vowels('te')
   s = 'te'
   rest = num_vowels('e') = 1
```

```
num_vowels('e')
   s = 'e'   s[0] -> 'e'
   rest = num_vowels('') = 0
   return 1 + 0 = 1
```

```
def num_vowels(s):
    if s == '':
        return 0
    else:
        rest = num_vowels(s[1:])
        if s[0] in 'aeiou':
            return 1 + rest
        else:
            return 0 + rest
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
num_vowels('ate')
   s = 'ate'
   rest = num_vowels('te')
```

```
num_vowels('te')
   s = 'te'   s[0] -> 't'
   rest = num_vowels('e') = 1
   return 0 + 1 = 1
```

```
def num_vowels(s):
    if s == '':
        return 0
    else:
        rest = num_vowels(s[1:])
        if s[0] in 'aeiou':
            return 1 + rest
        else:
            return 0 + rest
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
num_vowels('ate')
   s = 'ate'
   rest = num_vowels('te') = 1
```

```
num_vowels('te')
   s = 'te'   s[0] -> 't'
   rest = num_vowels('e') = 1
   return 0 + 1 = 1
```

```
def num_vowels(s):
    if s == '':
        return 0
    else:
        rest = num_vowels(s[1:])
        if s[0] in 'aeiou':
            return 1 + rest
        else:
            return 0 + rest
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
num_vowels('ate')
   s = 'ate'   s[0] -> 'a'
   rest = num_vowels('te') = 1
   return 1 + 1 = 2
```

result: **2**

```
def num_vowels(s):
    if s == '':
        return 0
    else:
        rest = num_vowels(s[1:])
        if s[0] in 'aeiou':
            return 1 + rest
        else:
            return 0 + rest
```

Recursively Raising a Number to a Power

```
def power(b, p):
    """ returns b raised to the p power
        inputs: b is a number (int or float)
                p is a non-negative integer
    """
    if                          # base case


    else:
```

- Ask yourself:

(base case) When can I determine b^p *without* determining a smaller power?

(recursive substructure) How could I use ***anything smaller*** than b^p to determine b^p?

Recursively Raising a Number to a Power

```
def power(b, p):
    """ returns b raised to the p power
        inputs: b is a number (int or float)
                p is a non-negative integer
    """
    if p == 0:                  # base case
        return 1

    else:
        pow_rest = power(b, p-1)
        return b * pow_rest
```

- Ask yourself:

(base case) When can I determine b^p *without* determining a smaller power?

(recursive substructure) How could I use ***anything smaller*** than b^p to determine b^p?

How recursion works...

```
def power(b, p):
    if p == 0:
        return 1
    else:
        pow_rest = power(b, p-1)
        return b * pow_rest
```

power(3, 3)
```
b = 3, p = 3
pow_rest = power(3, 2)
```

power(3, 2)
```
b = 3, p = 2
pow_rest = power(3, 1)
```

power(3, 1)
```
b = 3, p = 1
pow_rest = power(3, 0)
```

power(3, 0)
```
b = 3, p = 0
base case!
return 1
```

4 different stack frames, each with its own b , p, and pow_rest

How recursion works...

```
def power(b, p):
    if p == 0:
        return 1
    else:
        pow_rest = power(b, p-1)
        return b * pow_rest
```

power(3, 3)
```
b = 3, p = 3
pow_rest = power(3, 2)
```

power(3, 2)
```
b = 3, p = 2
pow_rest = power(3, 1)
```

power(3, 1)
```
b = 3, p = 1
pow_rest = power(3, 0) = 1
```

power(3, 0)
```
b = 3, p = 0
base case!
return 1
```

4 different stack frames, each with its own b , p, and pow_rest

How recursion works...

```
def power(b, p):
    if p == 0:
        return 1
    else:
        pow_rest = power(b, p-1)
        return b * pow_rest
```

power(3, 3)
```
b = 3, p = 3
pow_rest = power(3, 2)
```

power(3, 2)
```
b = 3, p = 2
pow_rest = power(3, 1)
```

power(3, 1)
```
b = 3, p = 1
pow_rest = power(3, 0) = 1
return 3 * 1 = 3
```

b pow_rest

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def power(b, p):
    if p == 0:
        return 1
    else:
        pow_rest = power(b, p-1)
        return b * pow_rest
```

power(3, 3)
```
b = 3, p = 3
pow_rest = power(3, 2)
```

power(3, 2)
```
b = 3, p = 2
pow_rest = power(3, 1) = 3
```

power(3, 1)
```
b = 3, p = 1
pow_rest = power(3, 0) = 1
return 3 * 1 = 3
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def power(b, p):
    if p == 0:
        return 1
    else:
        pow_rest = power(b, p-1)
        return b * pow_rest
```

power(3, 3)

```
b = 3, p = 3
pow_rest = power(3, 2)
```

power(3, 2)

```
b = 3, p = 2
pow_rest = power(3, 1) = 3
return 3 * 3 = 9
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def power(b, p):
    if p == 0:
        return 1
    else:
        pow_rest = power(b, p-1)
        return b * pow_rest
```

power(3, 3)
b = 3, p = 3
pow_rest = power(3, 2) = 9

power(3, 2)
b = 3, p = 2
pow_rest = power(3, 1) = 3
return 3 * 3 = 9

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
power(3, 3)
   b = 3, p = 3
   pow_rest = power(3, 2) = 9
   return 3 * 9 = 27
```

result: **27**

```
def power(b, p):
    if p == 0:
        return 1
    else:
        pow_rest = power(b, p-1)
        return b * pow_rest
```

Debugging with pdb

Debugging the simple way:

```
x = my_function()

print(x)    # use print to see variable contents
```

Debugging the interactive way:

```
$ python3 -m pdb my_function.py
```

or...

```
import pdb

pdb.set_trace()   # use “breakpoints” to stop
execution
```

Debugging with pdb

pdb provides an *interactive* debugging session.

pdb commands:

- c: continue execution
- w: shows the context of the current line it is executing.
- a: print the argument list of the current function
- s: Execute the current line and stop at the first possible occasion.
- n: Continue execution until the next line in the current function is reached or it returns.

https://docs.python.org/3.2/library/pdb.html