Function Calls (cont.);
A First Look at Recursion

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

Recall: Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

y = foo(y, x)
print(x, y)

foo(x, x)
print(x, y)

print(foo(x, y))
print(x, y)
```

foo

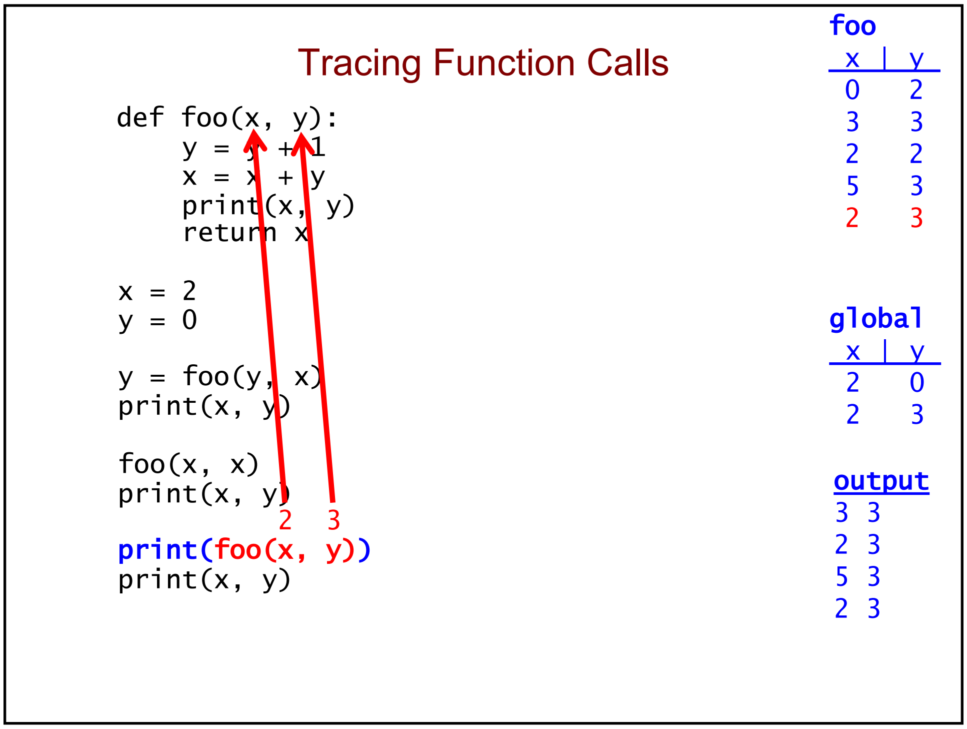
x	y
0	2
3	3
2	2
5	3

global

x	y
2	0
2	3

output

```
3 3
2 3
5 3
2 3
```



Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

y = foo(y, x)
print(x, y)

foo(x, x)
print(x, y)

print(foo(x, y))
print(x, y)
```

foo

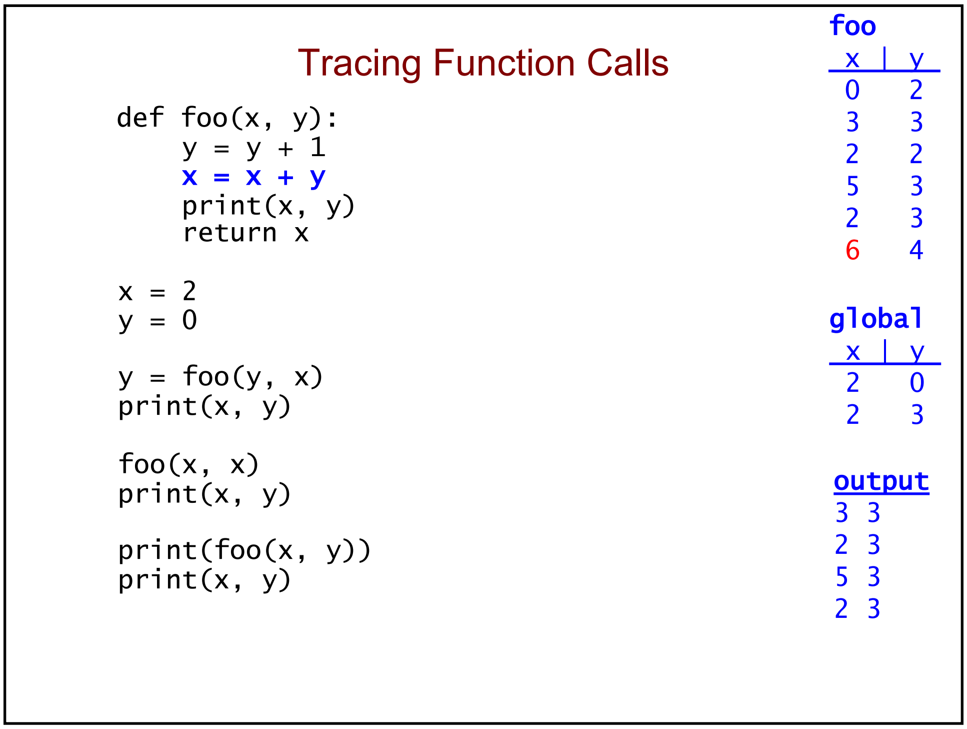
x	y
0	2
3	3
2	2
5	3
2	3

global

x	y
2	0
2	3

output

```
3 3
2 3
5 3
2 3
```



Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

y = foo(y, x)
print(x, y)

foo(x, x)
print(x, y)

print(foo(x, y))
print(x, y)
```

foo

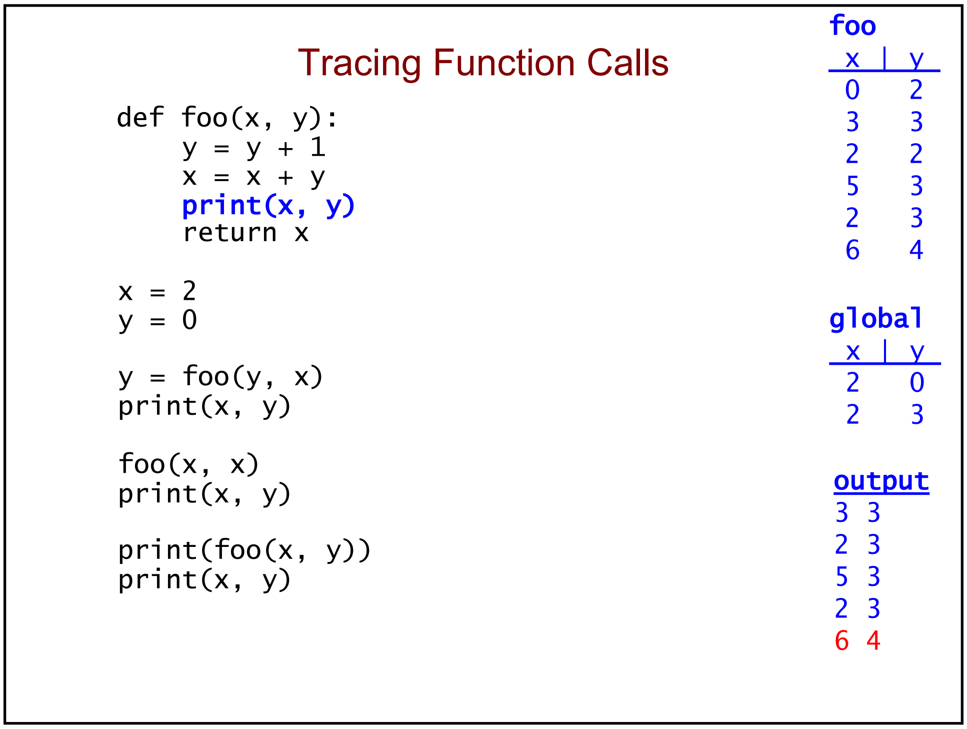
x	y
0	2
3	3
2	2
5	3
2	3
6	4

global

x	y
2	0
2	3

output

```
3 3
2 3
5 3
2 3
```



Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

y = foo(y, x)
print(x, y)

foo(x, x)
print(x, y)

print(foo(x, y))
print(x, y)
```

foo

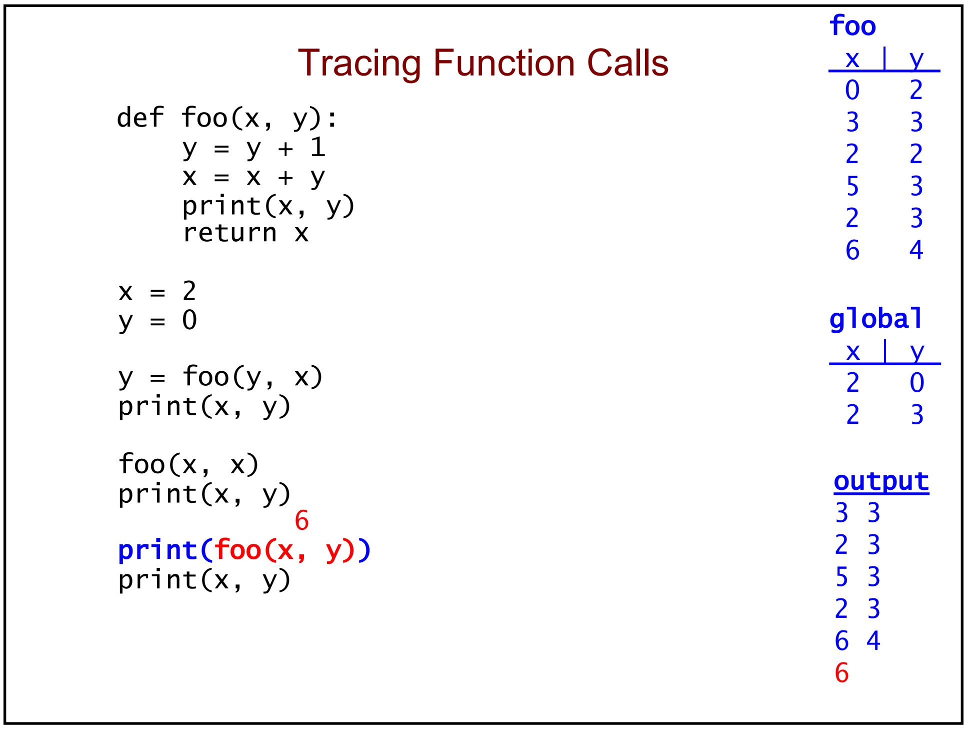
x	y
0	2
3	3
2	2
5	3
2	3
6	4

global

x	y
2	0
2	3

output

```
3 3
2 3
5 3
2 3
6 4
```



Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

y = foo(y, x)
print(x, y)

foo(x, x)
print(x, y)
           6
print(foo(x, y))
print(x, y)
```

foo

x	y
0	2
3	3
2	2
5	3
2	3
6	4

global

x	y
2	0
2	3

output

```
3 3
2 3
5 3
2 3
6 4
6
```

Tracing Function Calls

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

y = foo(y, x)
print(x, y)

foo(x, x)
print(x, y)

print(foo(x, y))
print(x, y)
```

foo

x	y
0	2
3	3
2	2
5	3
2	3
6	4

global

x	y
2	0
2	3

output

```
3 3
2 3
5 3
2 3
6 4
6
2 3
```

What does the rest do?

```
def foo(x, y):
    y = y + 1
    x = x + y
    print(x, y)
    return x

x = 2
y = 0

y = foo(y, x)
print(x, y)

foo(x, x)
print(x, y)

print(foo(x, y))
print(x, y)
```

See the extra video
in the folder for this lecture
for a step-by-step
trace of this problem!

foo

x	y
0	2
3	3
2	2
5	3
2	3
6	4

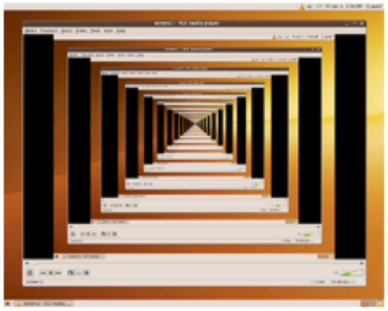
global

x	y
2	0
2	3

output

```
3 3
2 3
5 3
2 3
6 4
6
2 3
```

A First Look at Recursion



Functions Calling Themselves: *Recursion!*

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

- 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:         } base case
        return 1
    else:
        return n * fac(n – 1)   } recursive case
```

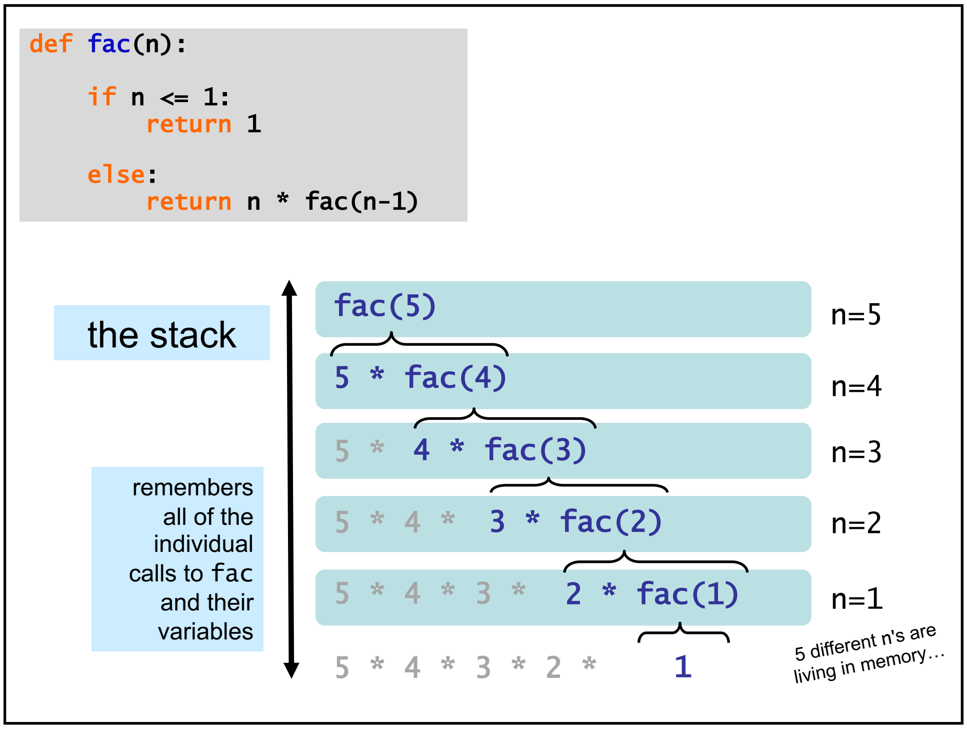
- One recursive call leads to another...

```
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!

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



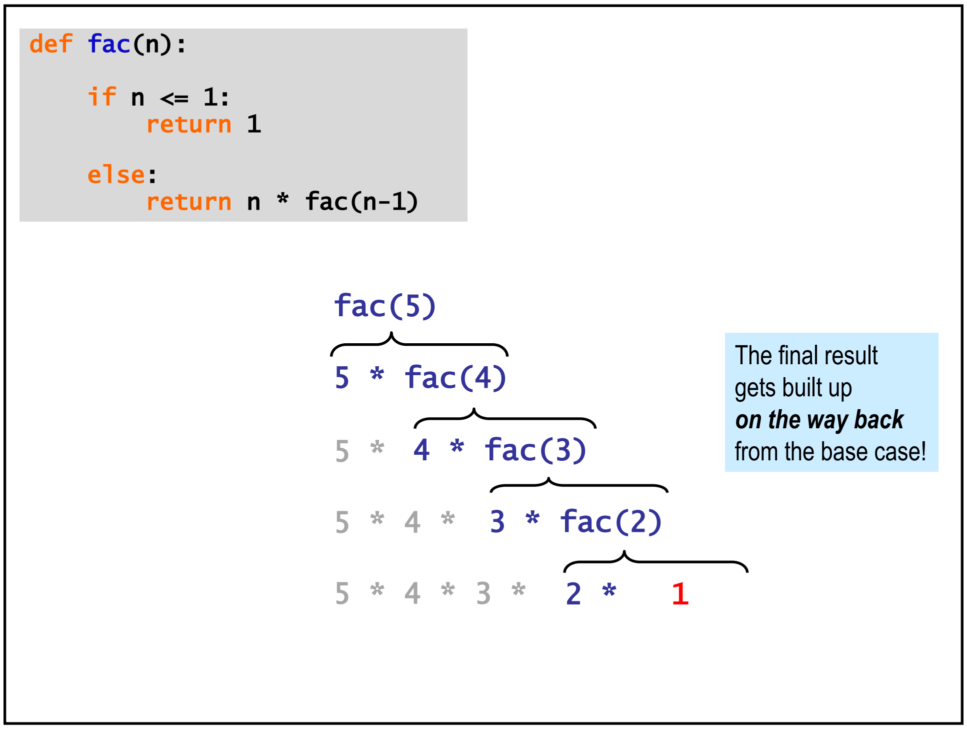
```
def fac(n):

    if n <= 1:
        return 1

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

the stack

remembers all of the individual calls to `fac` and their variables

```
fac(5)                          n=5
5 * fac(4)                      n=4
5 *  4 * fac(3)                 n=3
5 * 4 *  3 * fac(2)             n=2
5 * 4 * 3 *  2 * fac(1)         n=1
5 * 4 * 3 * 2 *     1
```

5 different n's are living in memory…

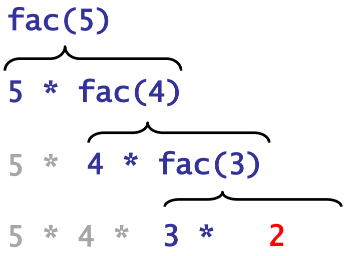


```
def fac(n):

    if n <= 1:
        return 1

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

```
fac(5)
5 * fac(4)
5 *  4 * fac(3)
5 * 4 *  3 * fac(2)
5 * 4 * 3 *  2 *    1
```

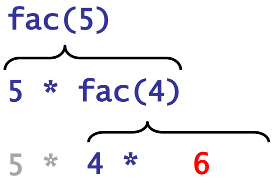
The final result gets built up ***on the way back*** from the base case!

```
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)
```

$$\overbrace{5 * 24}^{\texttt{fac(5)}}$$

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)

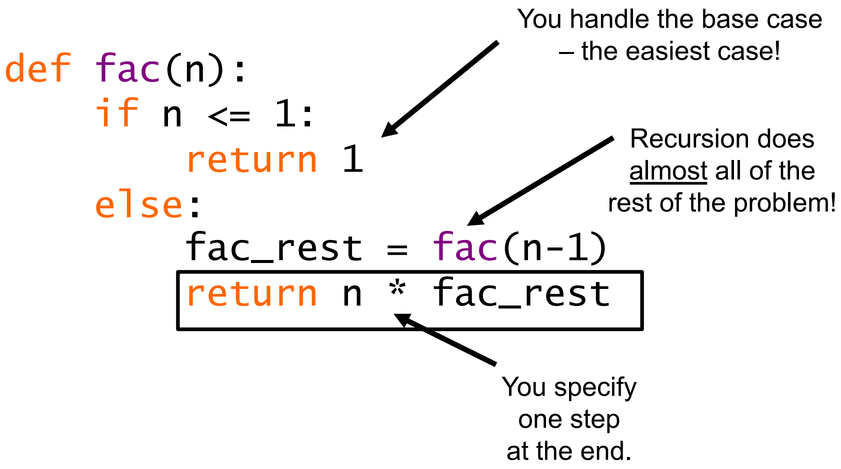
$$\boxed{\text{result: } 120}$$

Alternative Version of `fac(n)`

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

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

- Many people find this easier to read/write/understand.
- 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!***

Let Recursion Do the Work For You!



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

You handle the base case – the easiest case!

Recursion does <u>almost</u> all of the rest of the problem!

You specify one step at the end.

Recursively Processing a List or String

- Sequences are recursive!
  - 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                      # 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
```

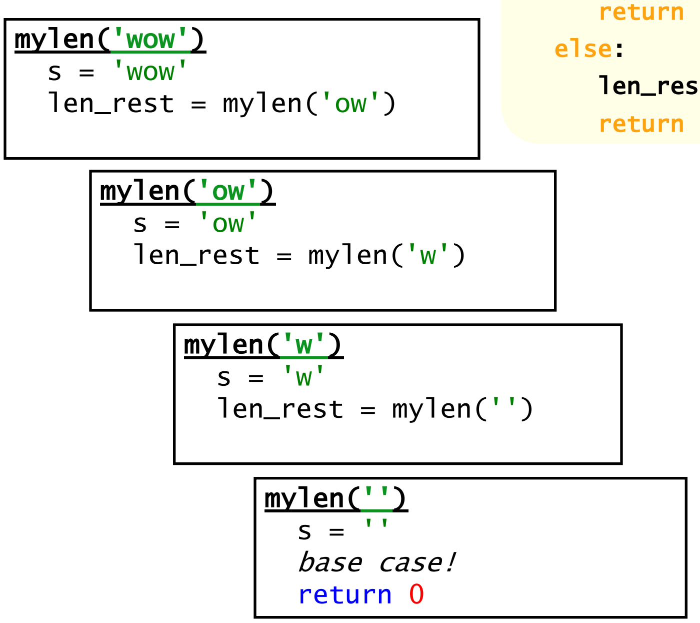
- Ask yourself:

(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
```



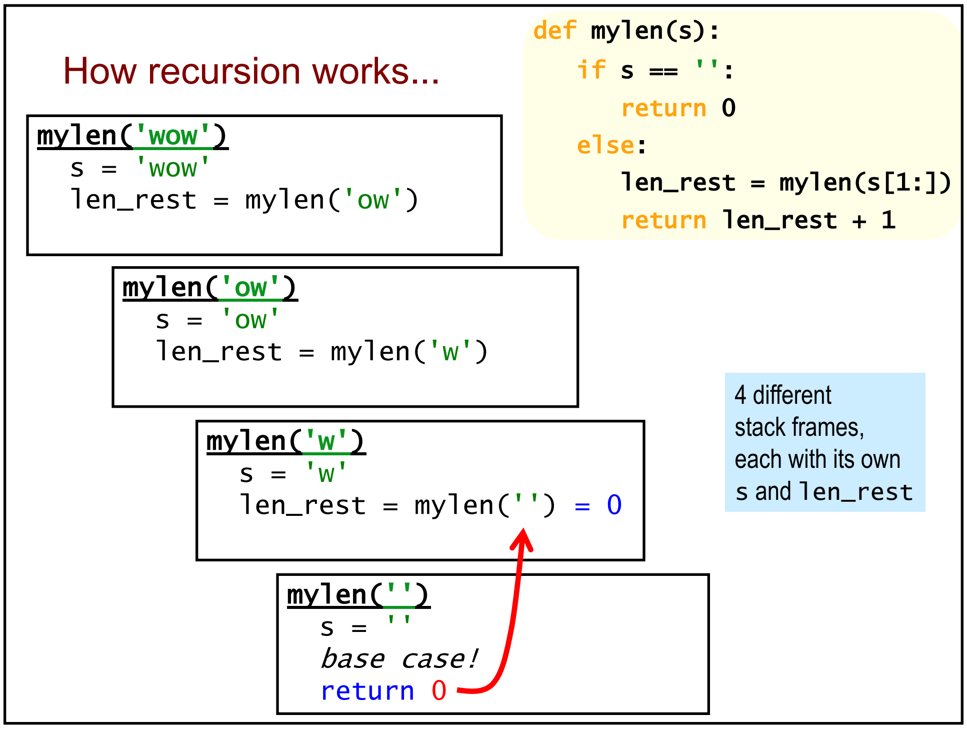
```
mylen('wow')
  s = 'wow'
  len_rest = mylen('ow')
```

```
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
```

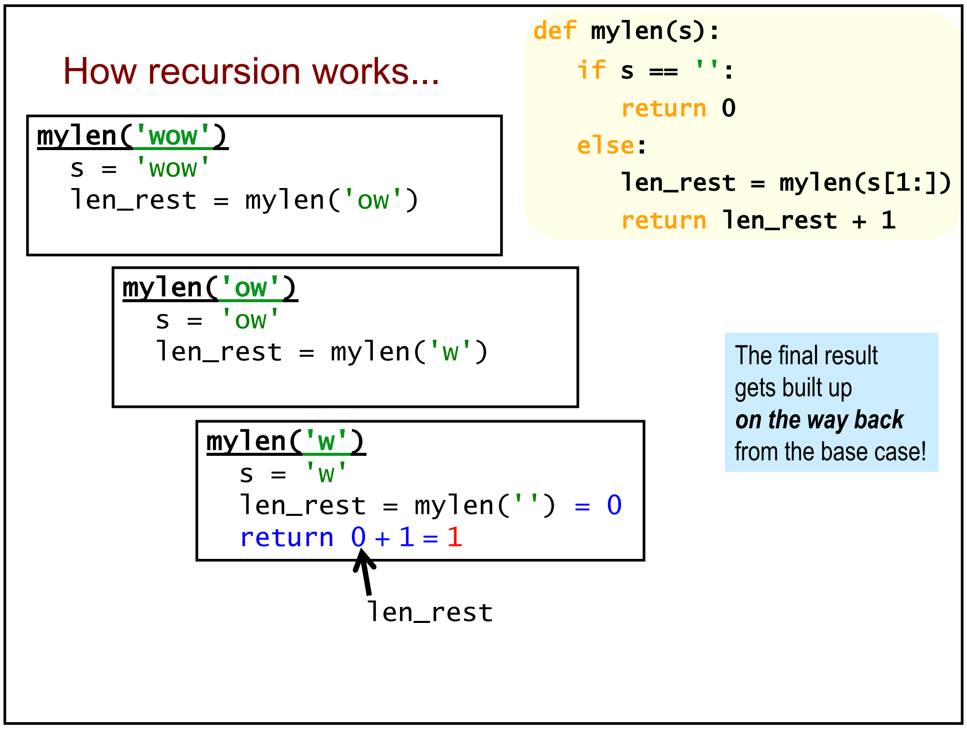
```
mylen('wow')
   s = 'wow'
   len_rest = mylen('ow')
```

```
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')
```

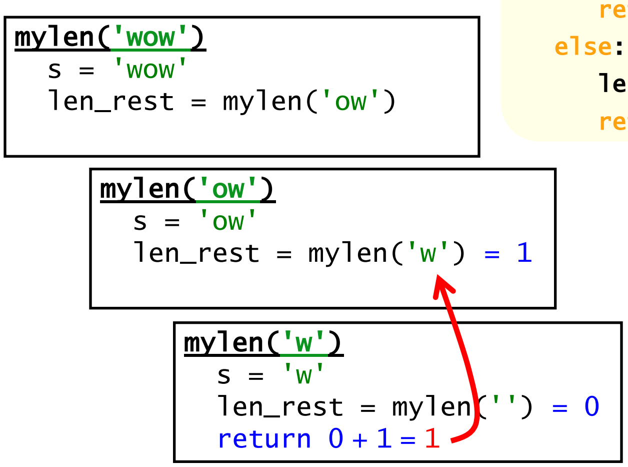
```
mylen('ow')
   s = 'ow'
   len_rest = mylen('w')
```

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

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
```

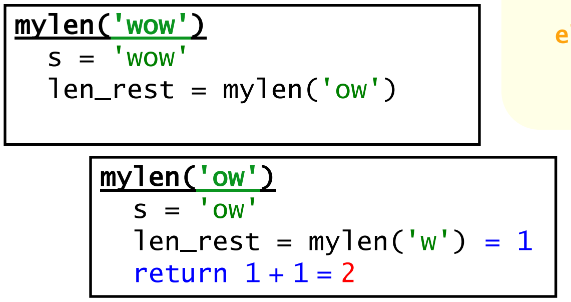


```
mylen('wow')
   s = 'wow'
   len_rest = mylen('ow')

      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!

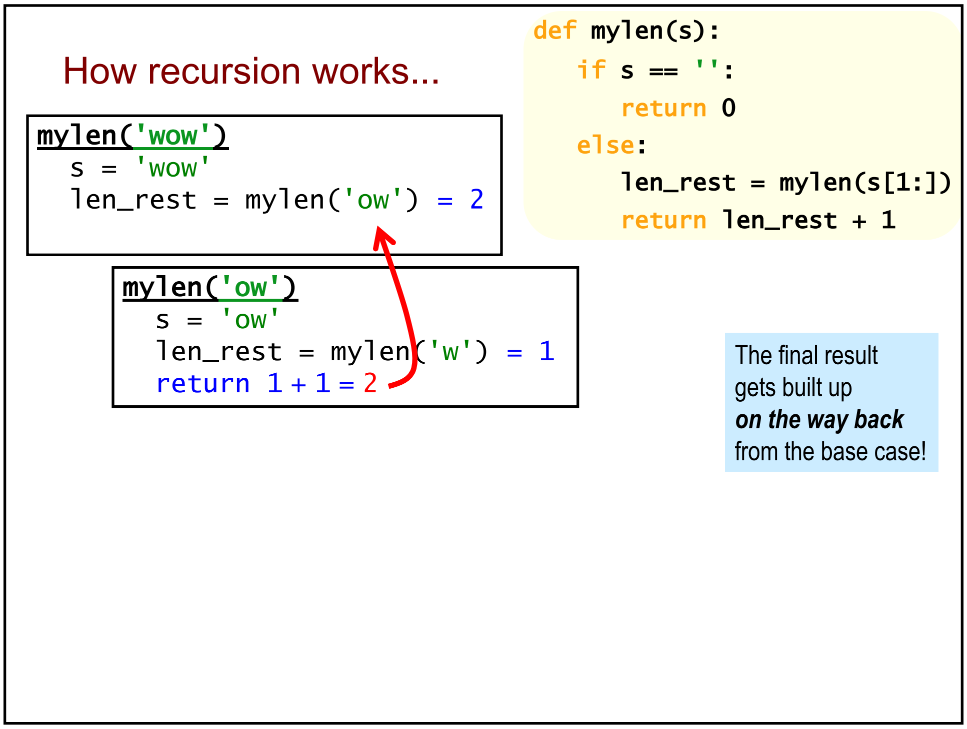
How recursion works...

```
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
         return 1 + 1 = 2
```

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



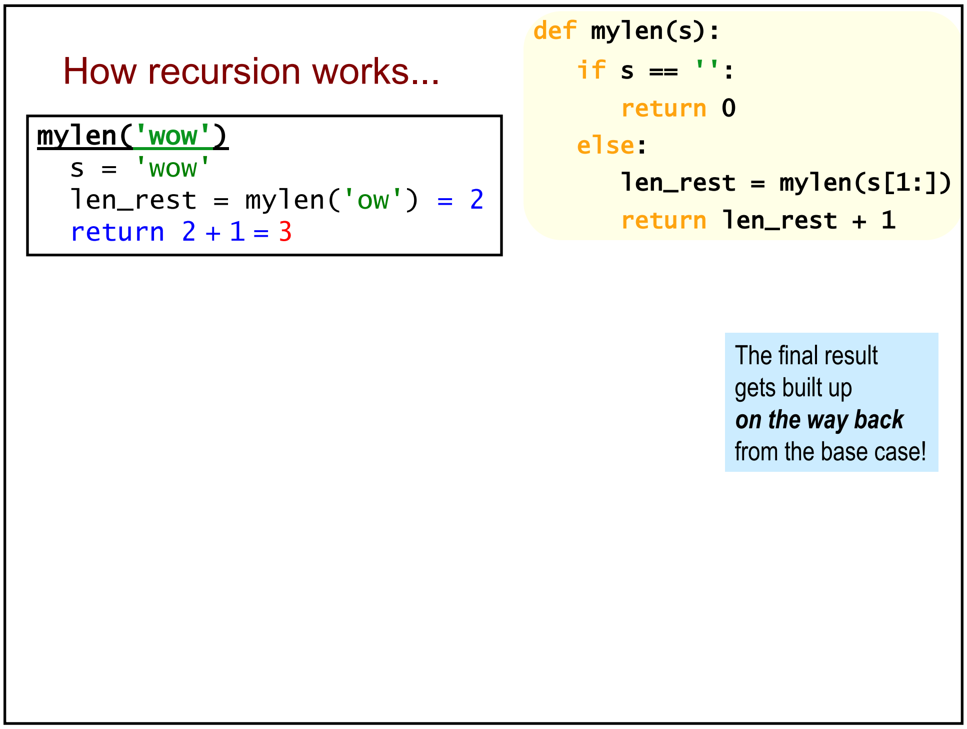
How recursion works...

```
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!

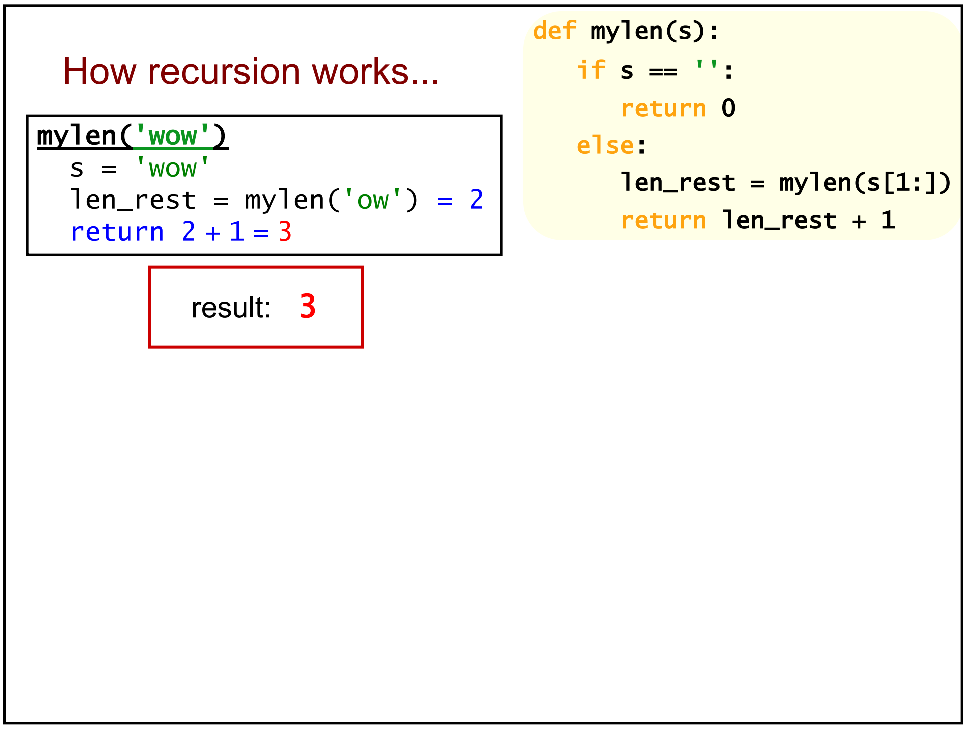


How recursion works...

```
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!

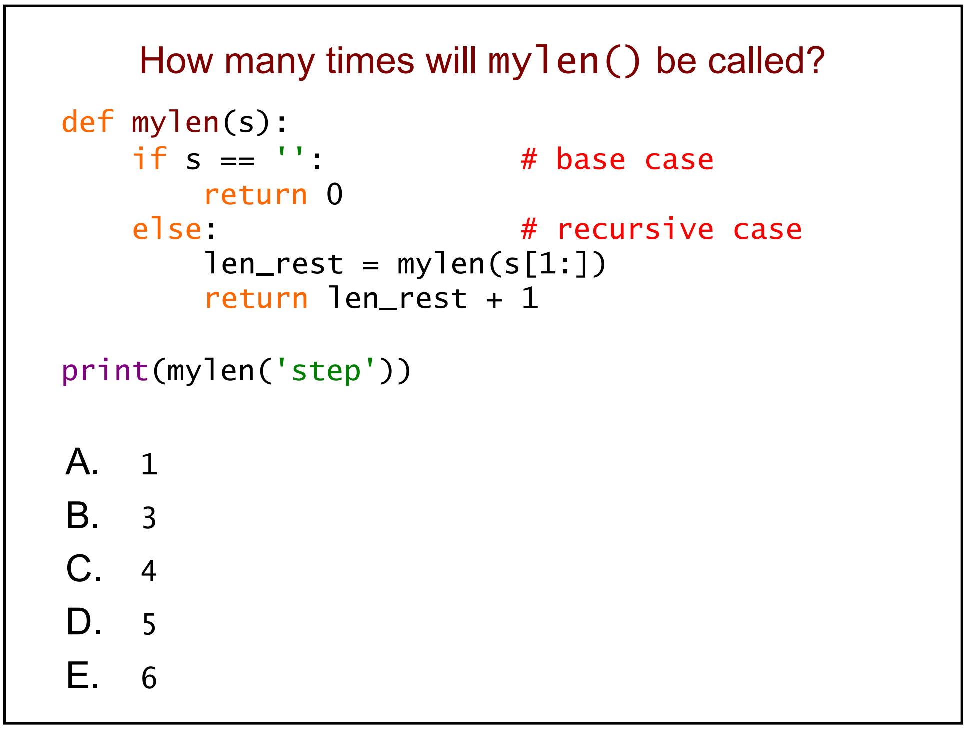


How recursion works...

```
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

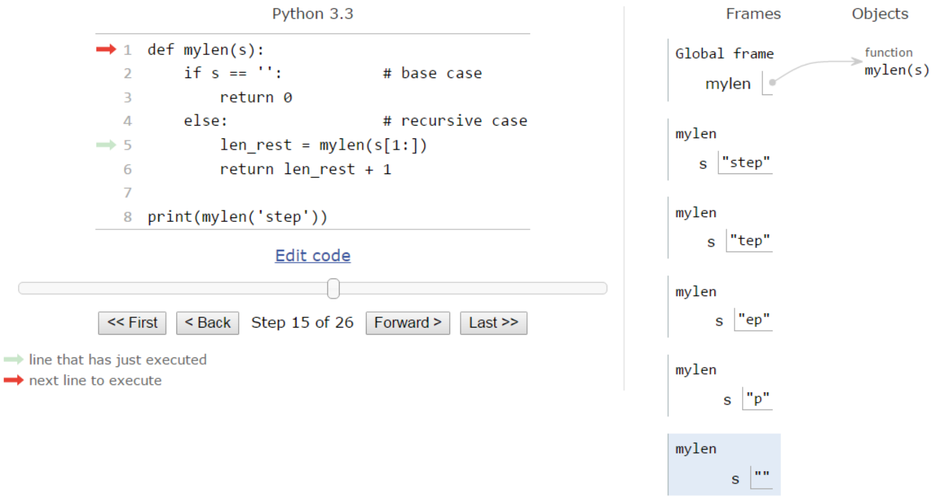
B. 3

C. 4

D. 5

E. 6

Visualizing Recursion (pythontutor.com/visualize.html)



Python 3.3

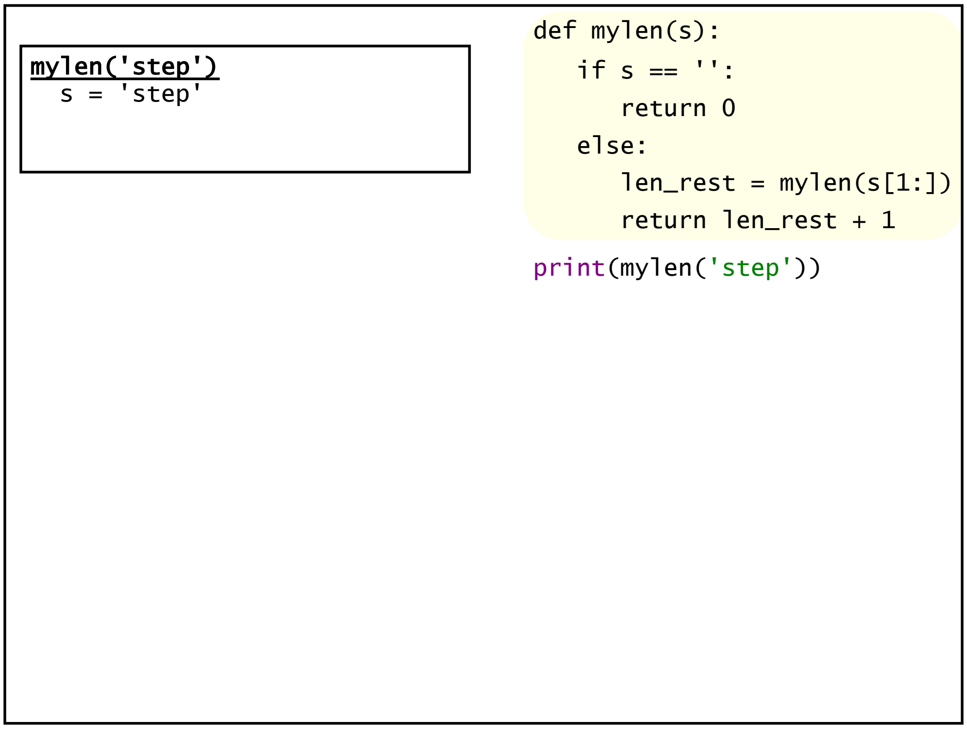
```
def mylen(s):
    if s == '':           # base case
        return 0
    else:                 # recursive case
        len_rest = mylen(s[1:])
        return len_rest + 1

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

Edit code

<< First | < Back | Step 15 of 26 | Forward > | Last >>

line that has just executed

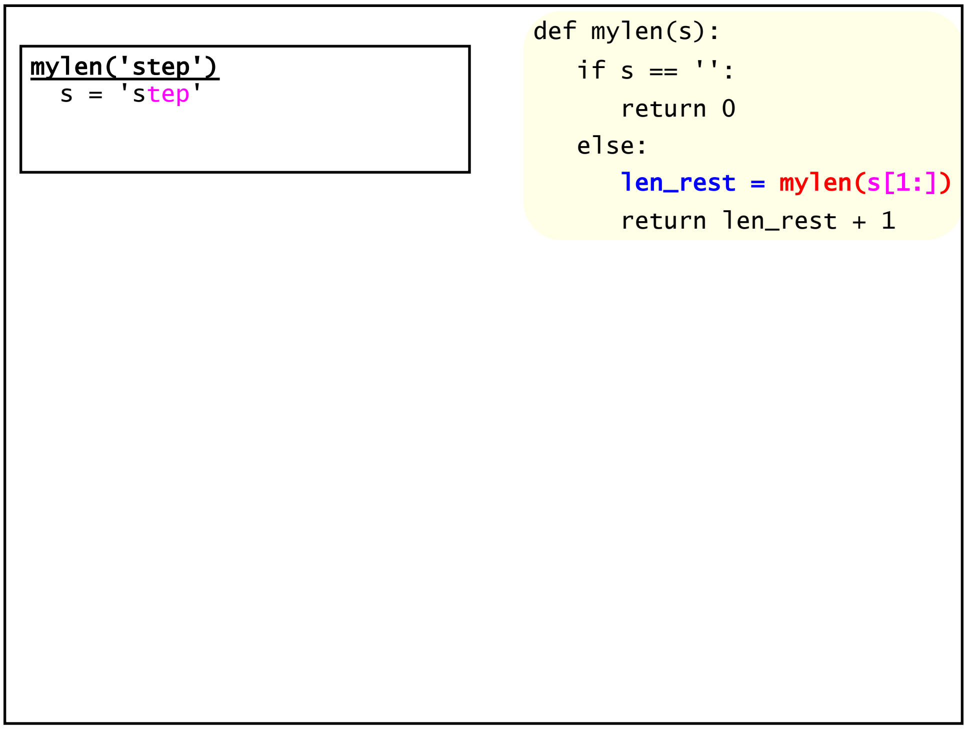
next line to execute



mylen('step')

```
  s = 'step'
```

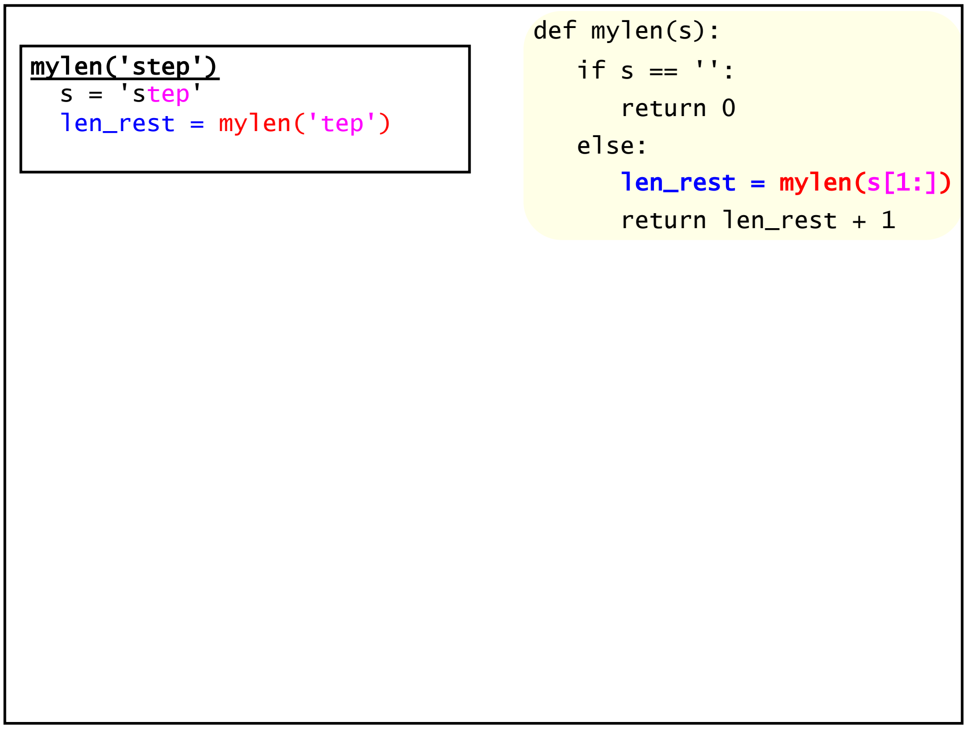
```
def mylen(s):
   if s == '':
      return 0
   else:
      len_rest = mylen(s[1:])
      return len_rest + 1

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



mylen('step')

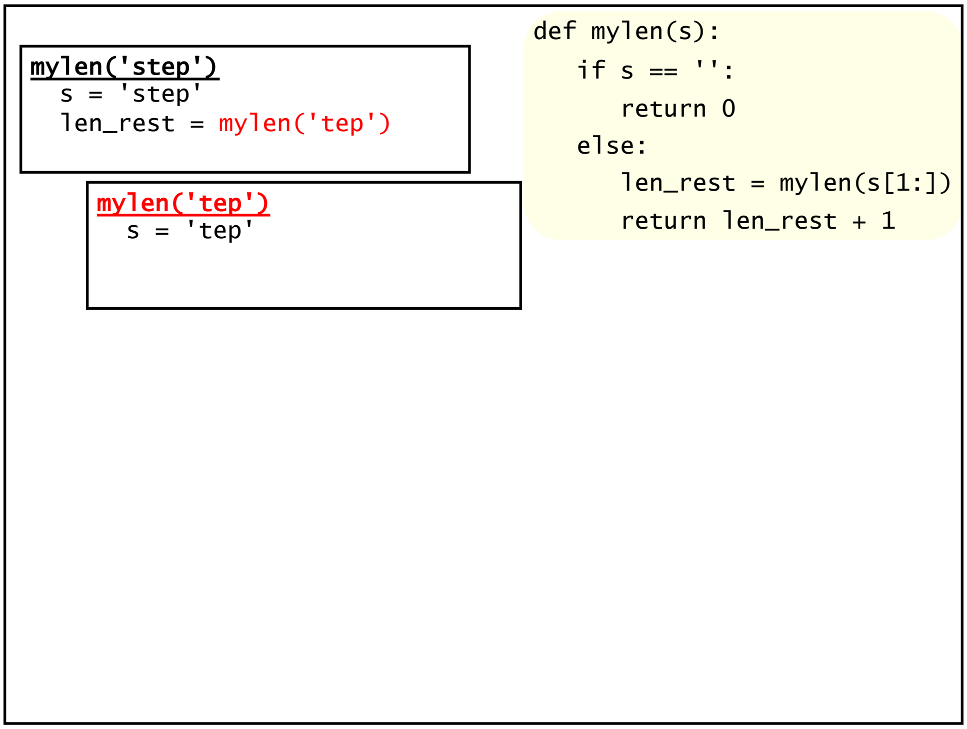
```
  s = 'step'
```

```
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')
```

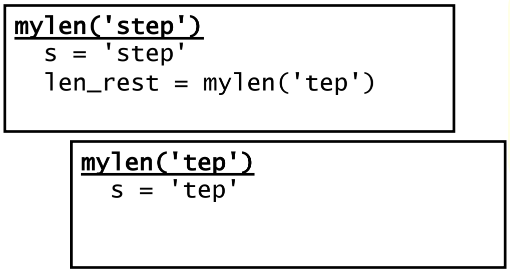
```
def mylen(s):
   if s == '':
      return 0
   else:
      len_rest = mylen(s[1:])
      return len_rest + 1
```



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

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

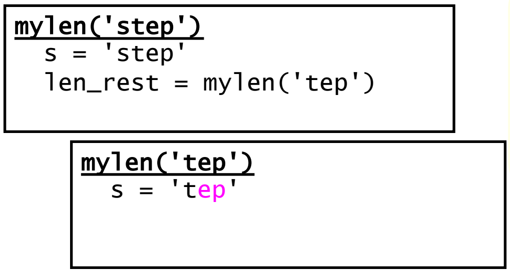
```
def mylen(s):
   if s == '':
      return 0
   else:
      len_rest = mylen(s[1:])
      return len_rest + 1
```



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

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

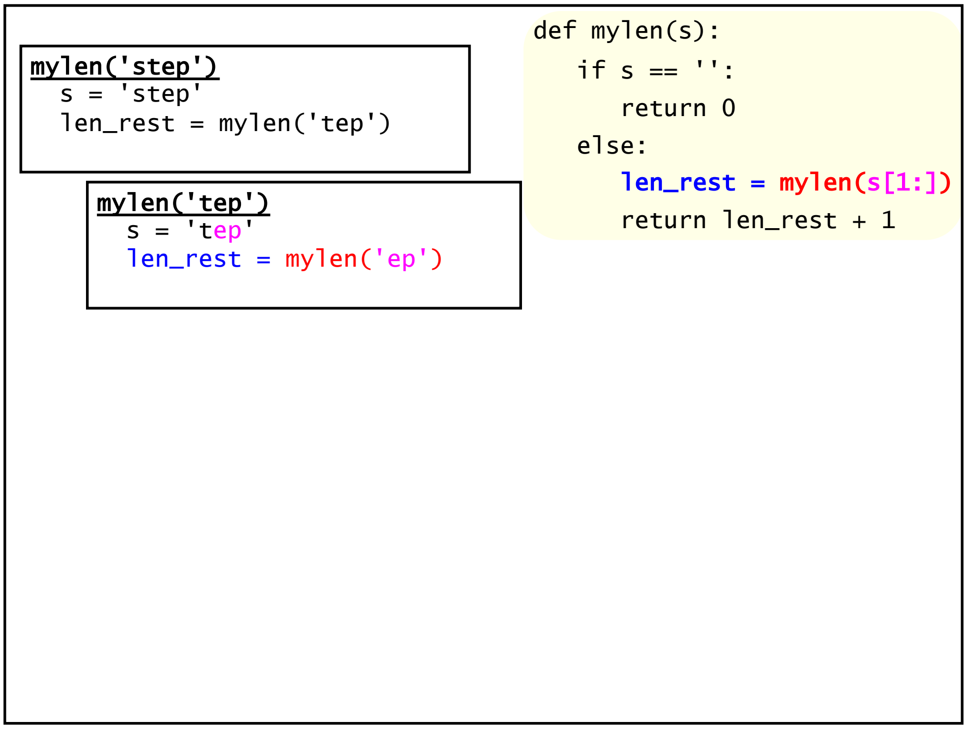
```
def mylen(s):
   if s == '':
      return 0
   else:
      len_rest = mylen(s[1:])
      return len_rest + 1
```



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

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

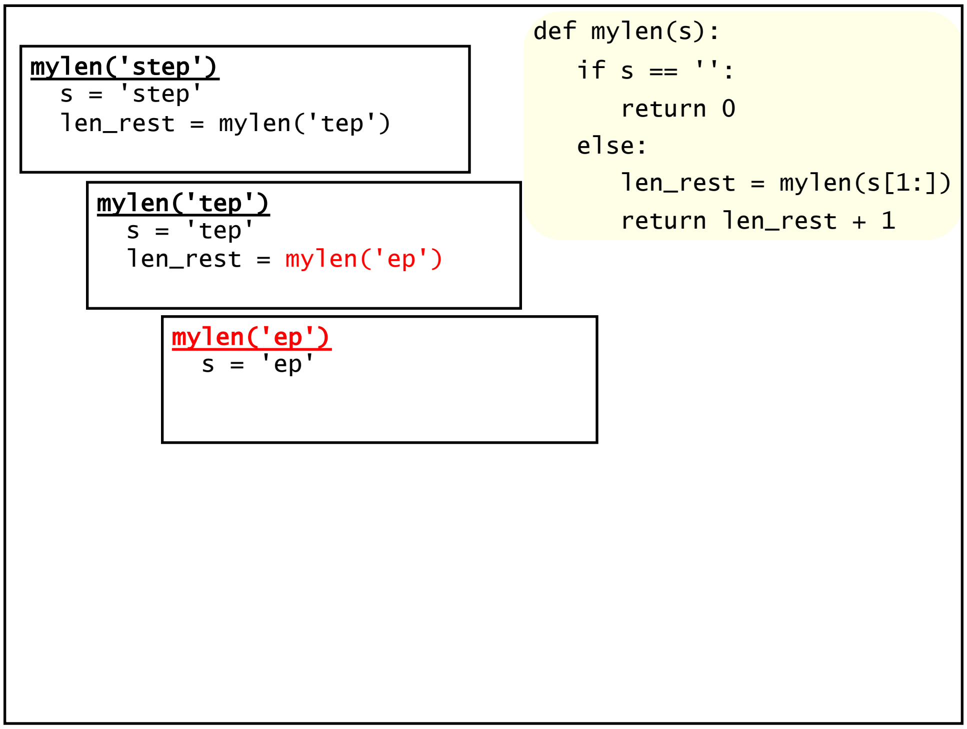
```
def mylen(s):
   if s == '':
      return 0
   else:
      len_rest = mylen(s[1:])
      return len_rest + 1
```



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

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

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

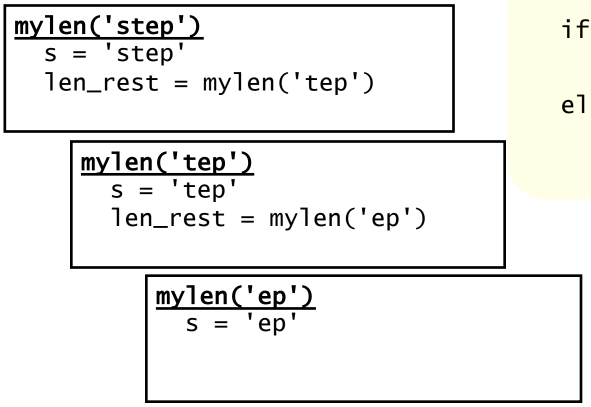


```
mylen('step')
  s = 'step'
  len_rest = mylen('tep')
```

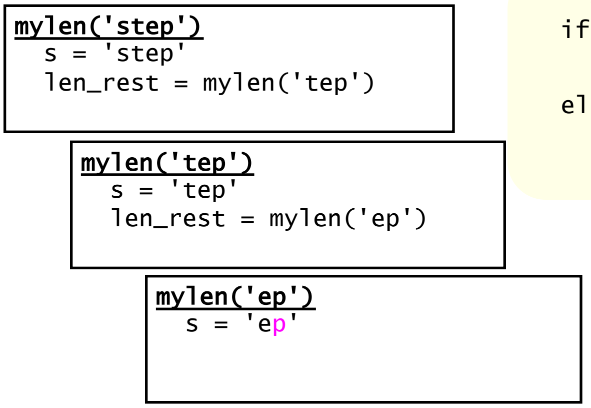
```
mylen('tep')
  s = 'tep'
  len_rest = mylen('ep')
```

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

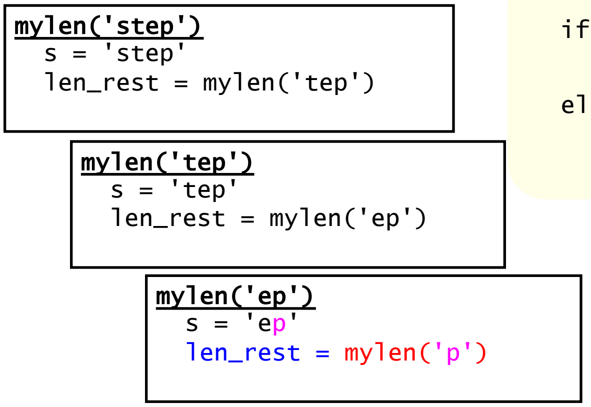
```
def mylen(s):
   if s == '':
      return 0
   else:
      len_rest = mylen(s[1:])
      return len_rest + 1
```



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



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

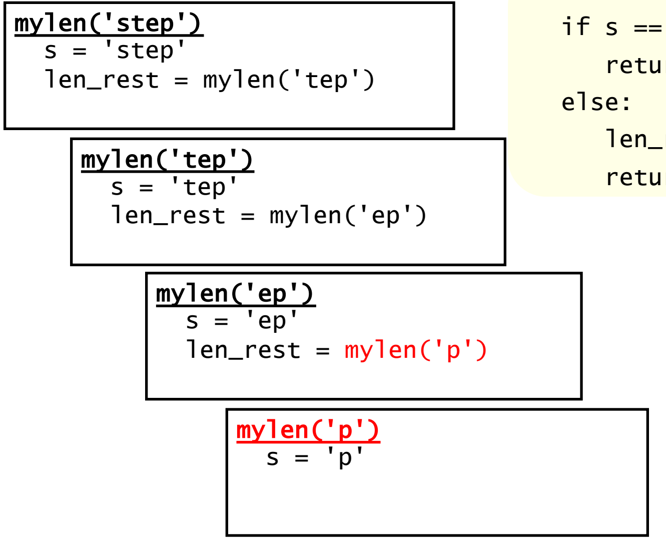


```
mylen('step')
  s = 'step'
  len_rest = mylen('tep')
```

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

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

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



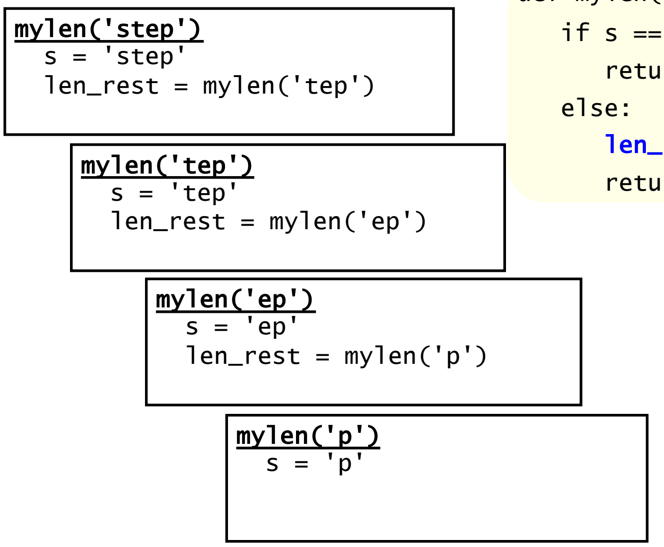
```
mylen('step')
  s = 'step'
  len_rest = mylen('tep')
```

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

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

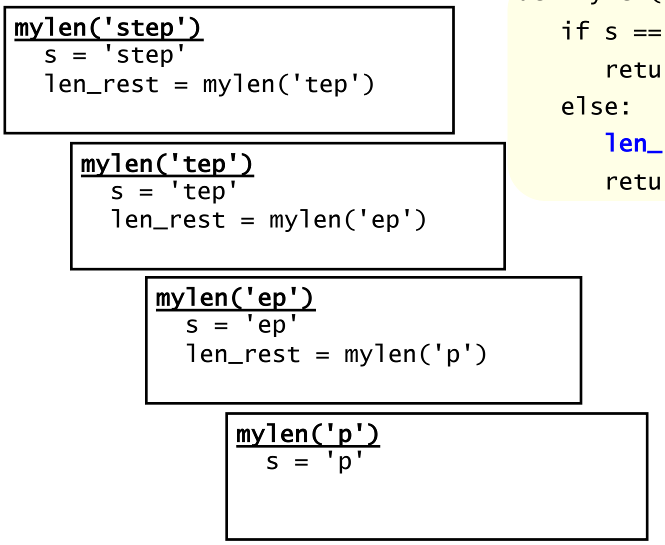
```
mylen('p')
  s = 'p'
```

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



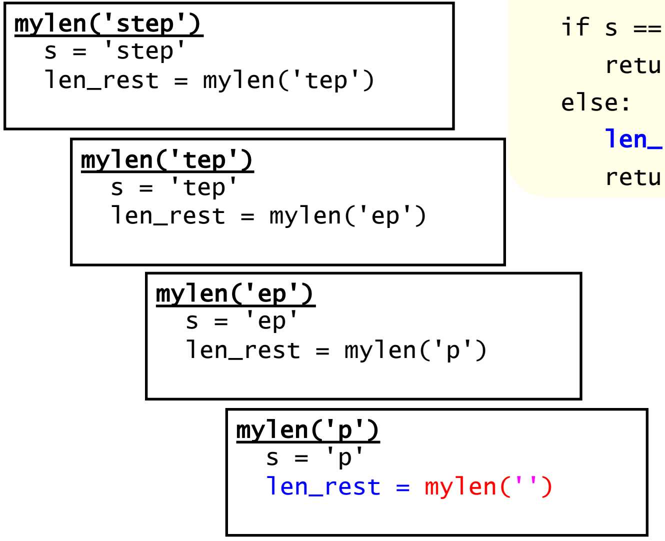
```
mylen('step')
  s = 'step'
  len_rest = mylen('tep')

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

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

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

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



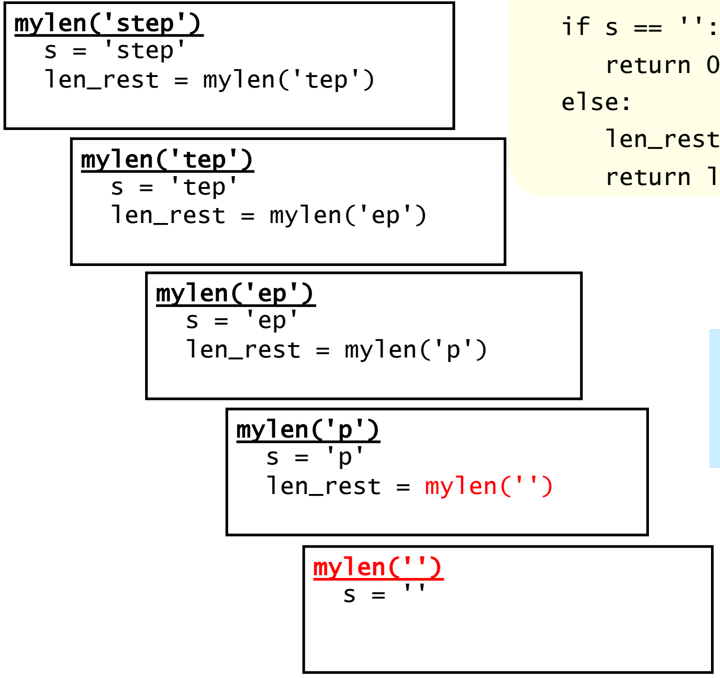
```
mylen('step')
  s = 'step'
  len_rest = mylen('tep')

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

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

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

```
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('')
```

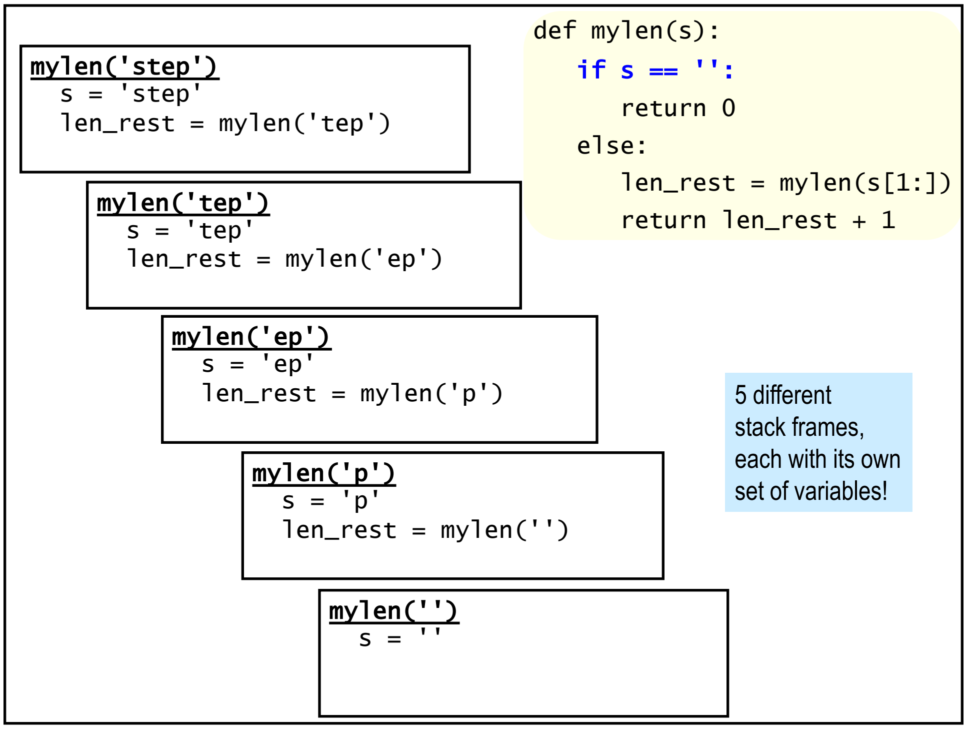
```
def mylen(s):
   if s == '':
      return 0
   else:
      len_rest = mylen(s[1:])
      return len_rest + 1
```



```
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 = ''
```

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

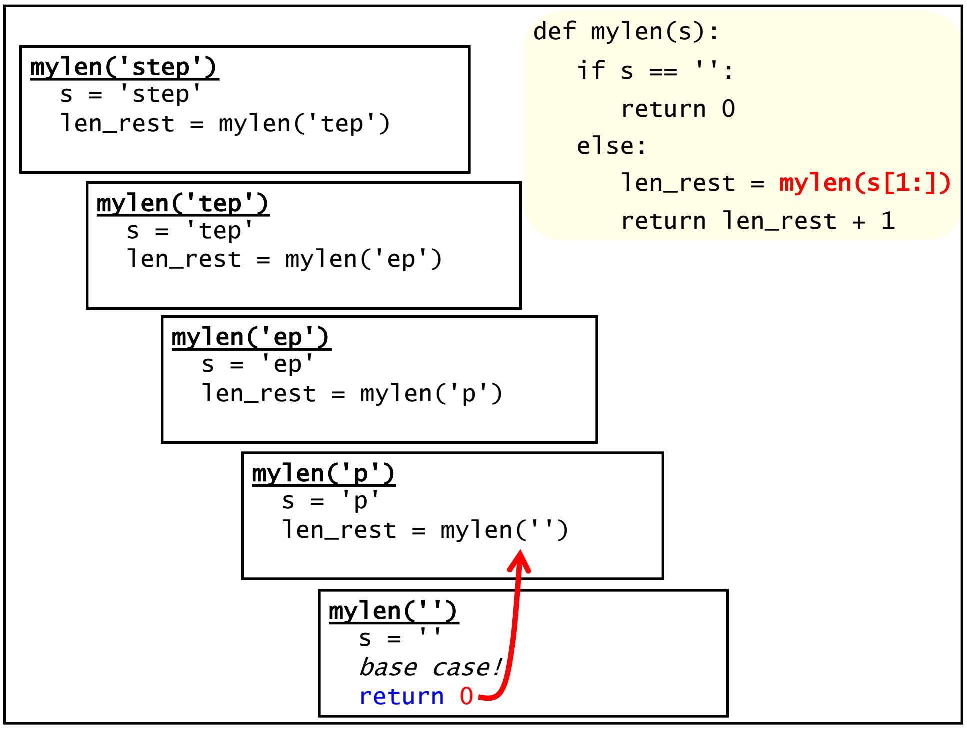
5 different stack frames, each with its own set of variables!



```
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 = ''
```

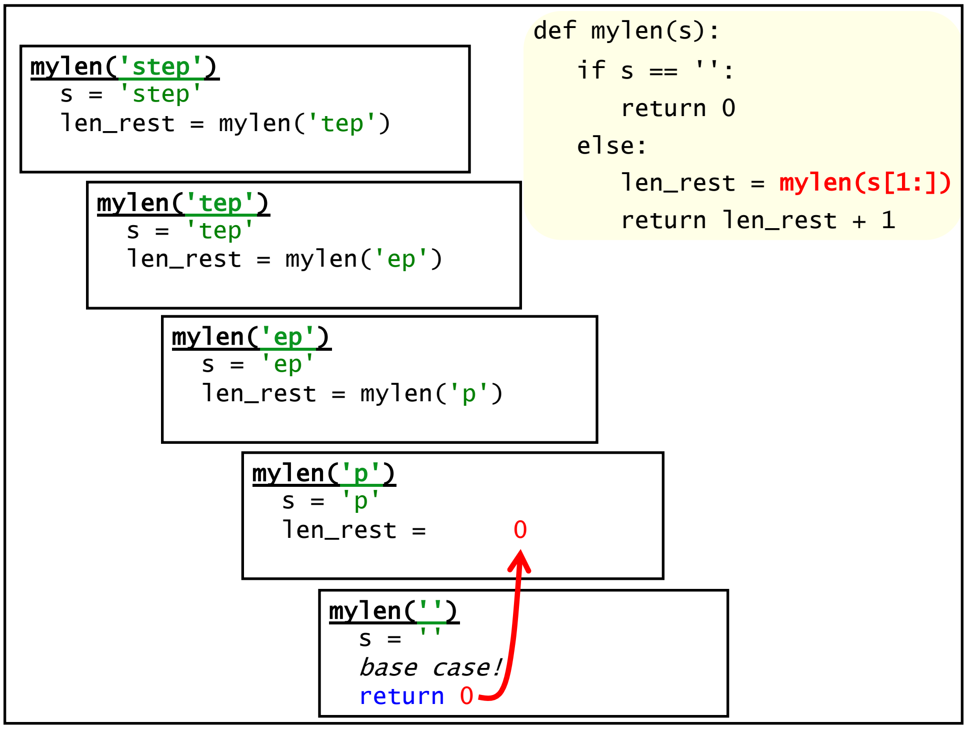
```
def mylen(s):
    if s == '':
        return 0
    else:
        len_rest = mylen(s[1:])
        return len_rest + 1
```

5 different stack frames, each with its own set of variables!



```
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
```

```
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')
```

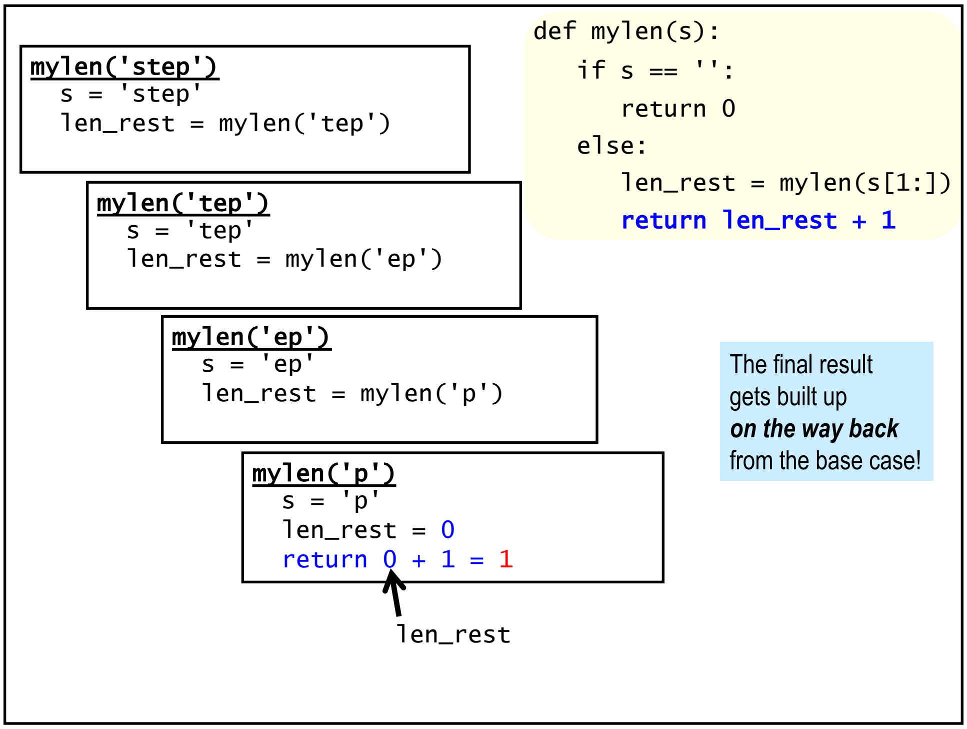
```
mylen('tep')
   s = 'tep'
   len_rest = mylen('ep')
```

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

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

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

```
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')
```

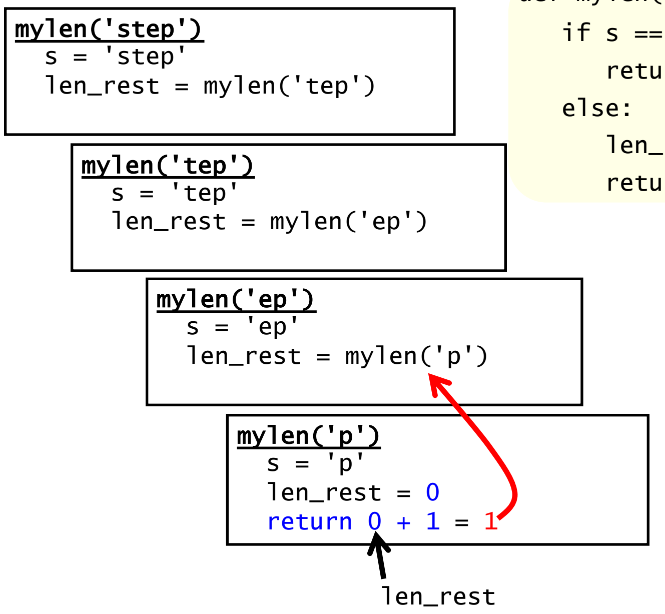
```
mylen('tep')
   s = 'tep'
   len_rest = mylen('ep')
```

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

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

```
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')
```

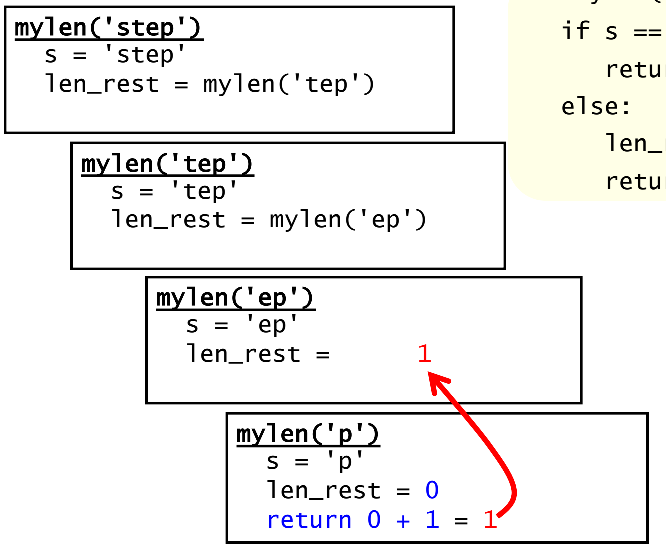
```
mylen('tep')
  s = 'tep'
  len_rest = mylen('ep')
```

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

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

```
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')
```

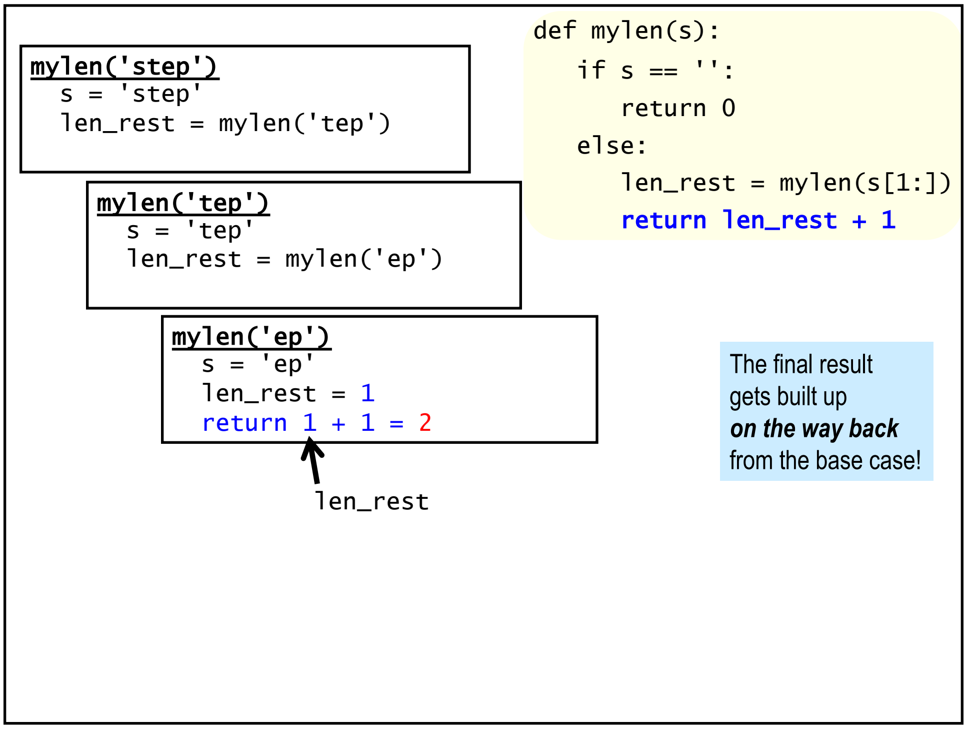
```
mylen('tep')
  s = 'tep'
  len_rest = mylen('ep')
```

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

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

```
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')
```

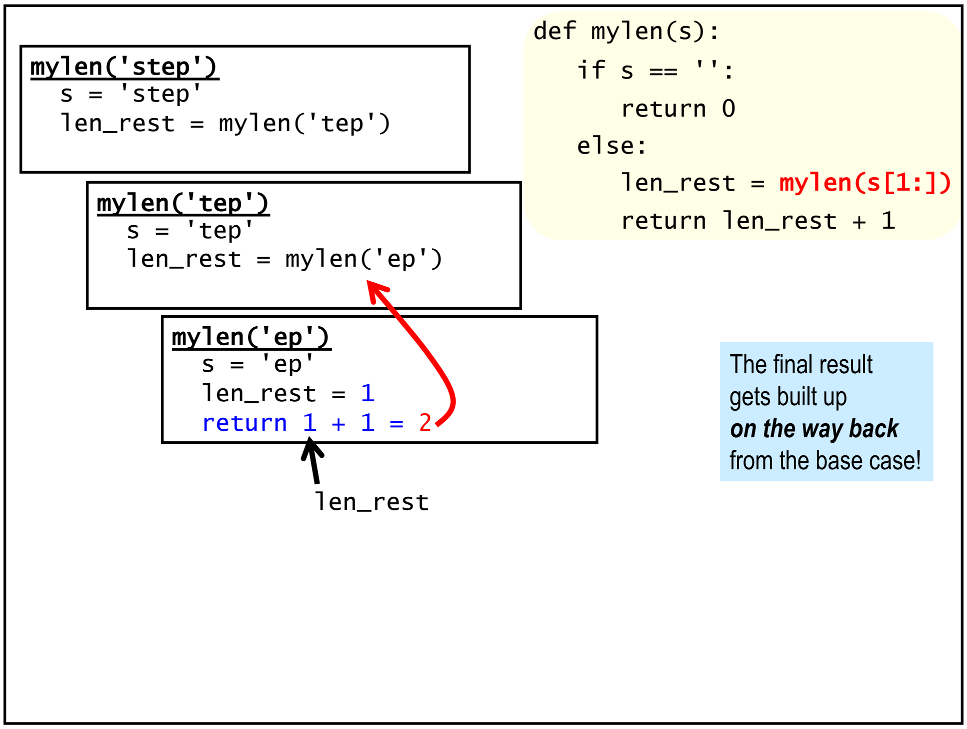
```
mylen('tep')
  s = 'tep'
  len_rest = mylen('ep')
```

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

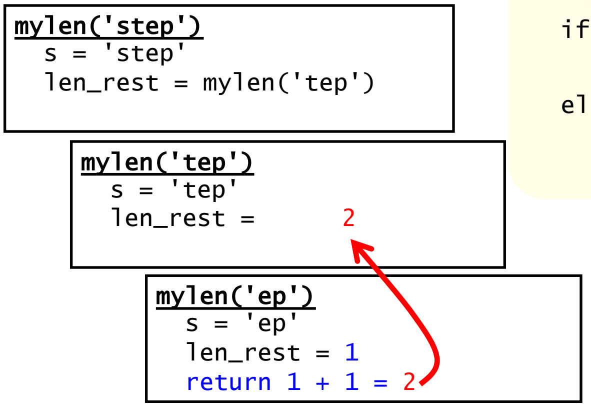
```
mylen('tep')
  s = 'tep'
  len_rest = mylen('ep')
```

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

```
len_rest
```

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

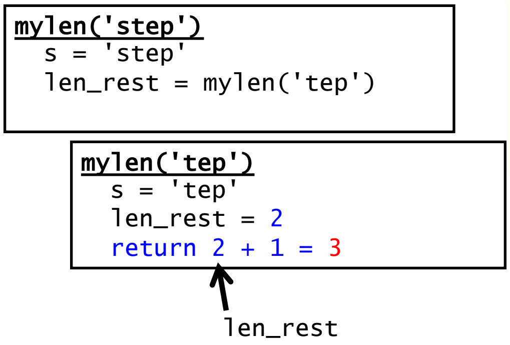
The final result gets built up ***on the way back*** from the base case!



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

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

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

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

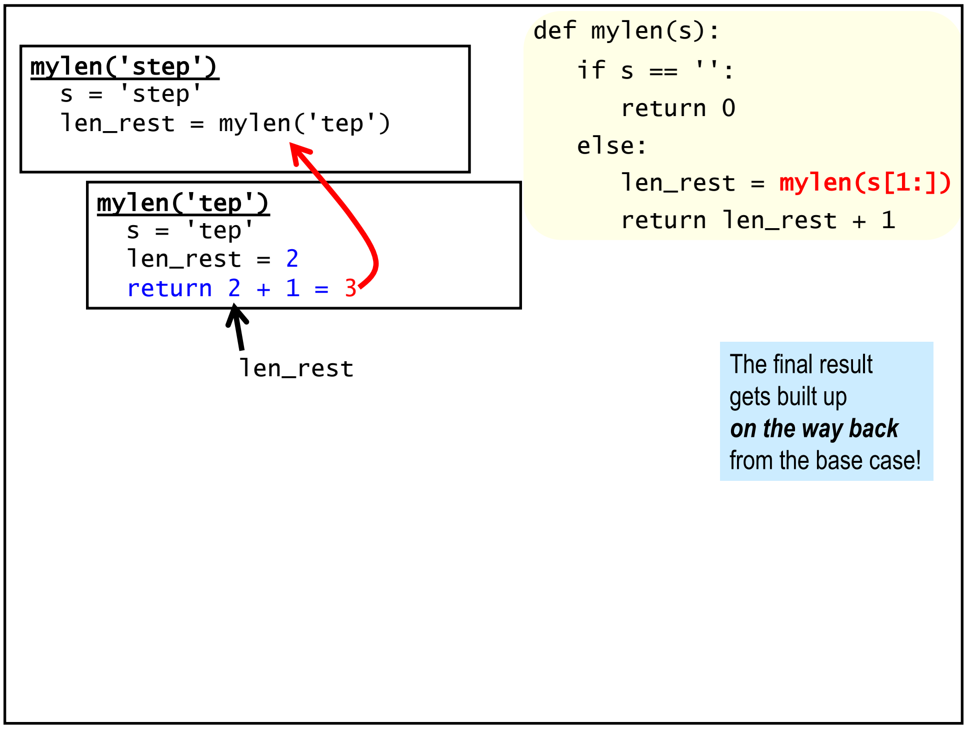
The final result gets built up ***on the way back*** from the base case!



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

mylen('tep')
  s = 'tep'
  len_rest = 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!



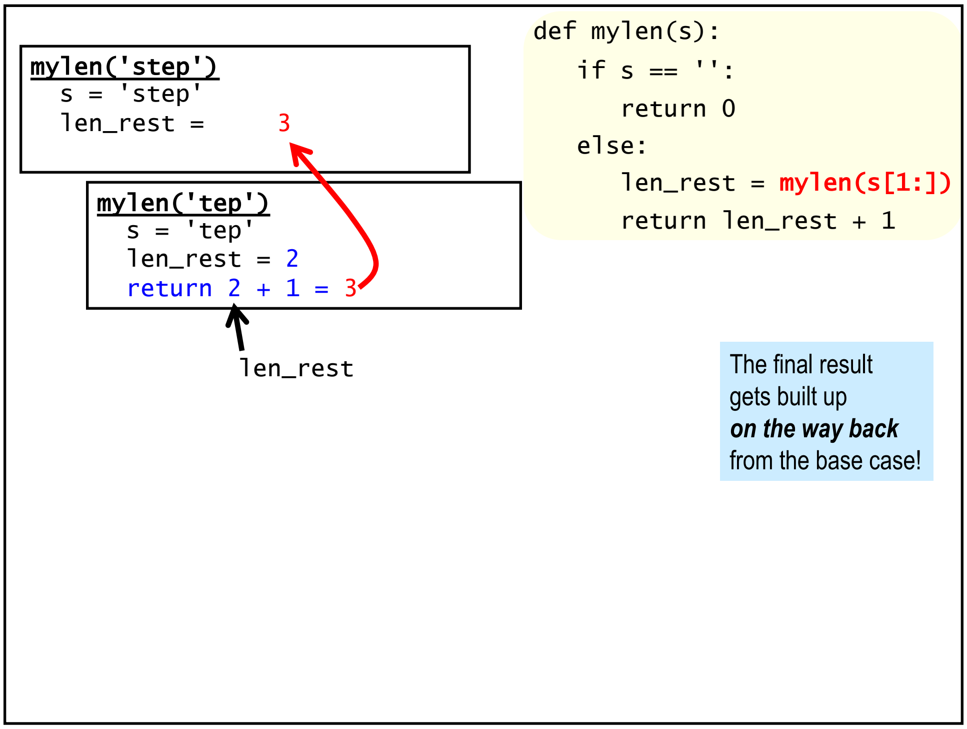
```
mylen('step')
  s = 'step'
  len_rest = mylen('tep')
```

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



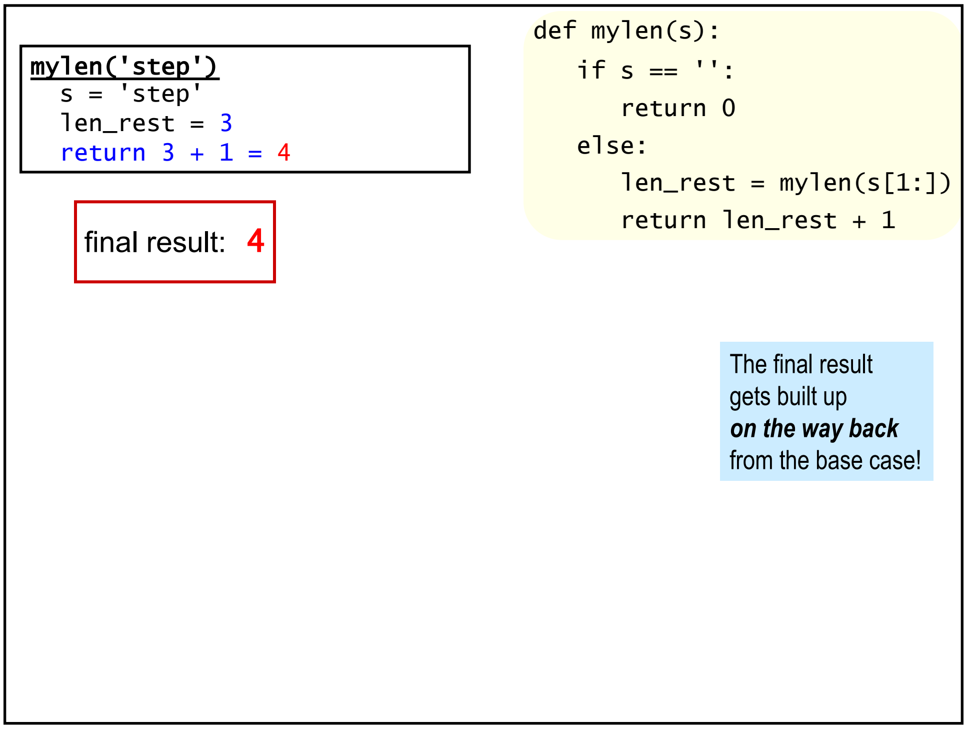
```
mylen('step')
  s = 'step'
  len_rest =      3
```

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

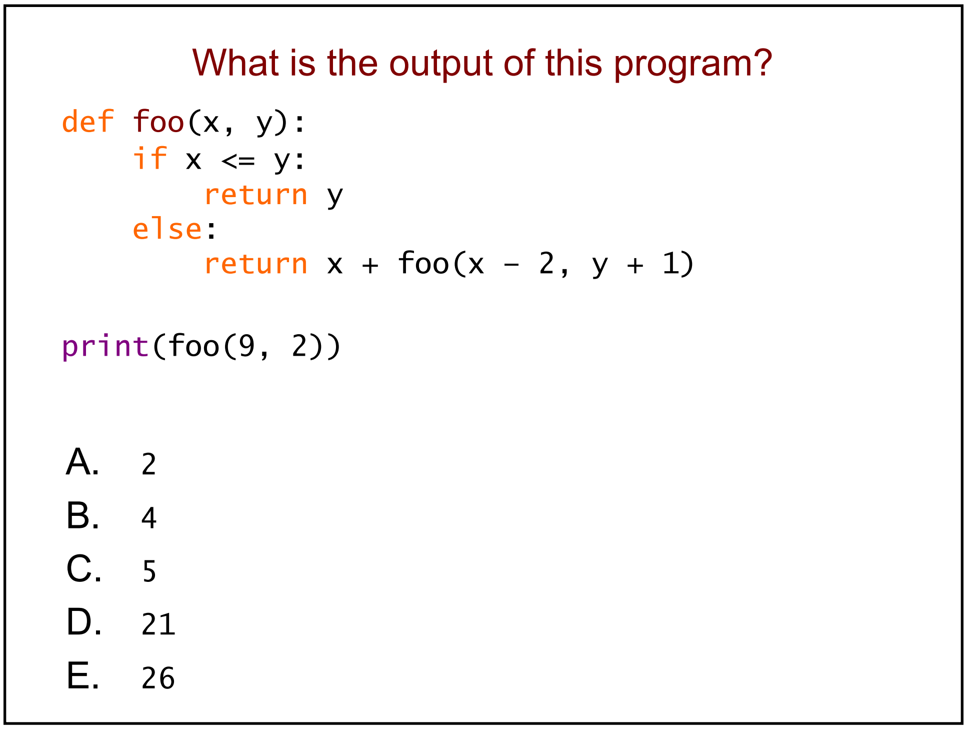


```
mylen('step')
  s = 'step'
  len_rest = 3
  return 3 + 1 = 4
```

final 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))
```

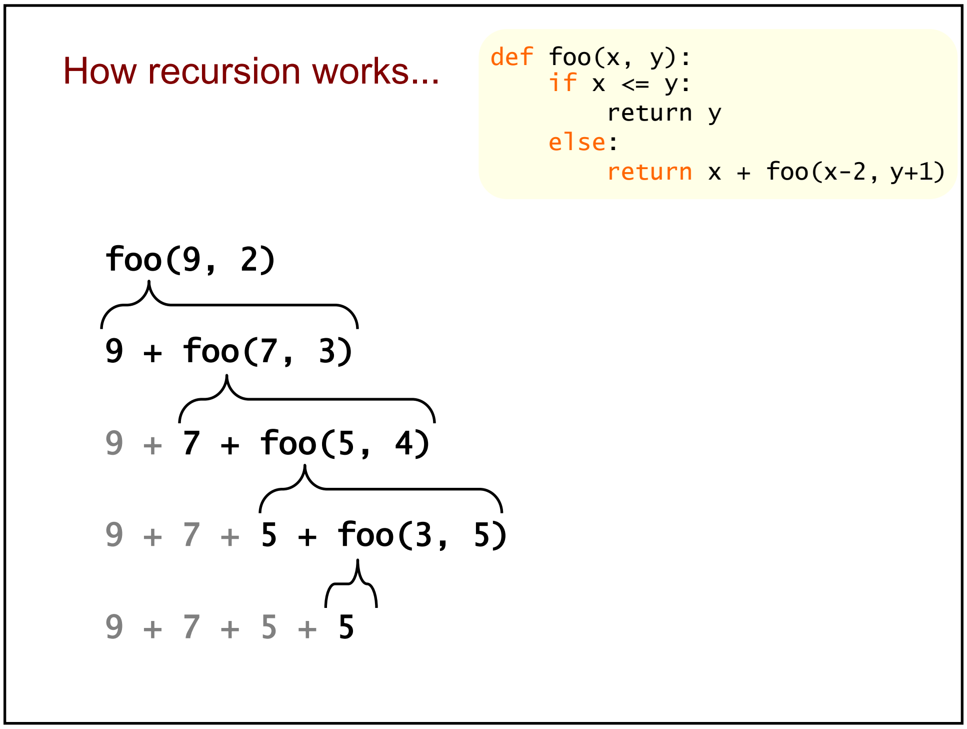
A. 2

B. 4

C. 5

D. 21

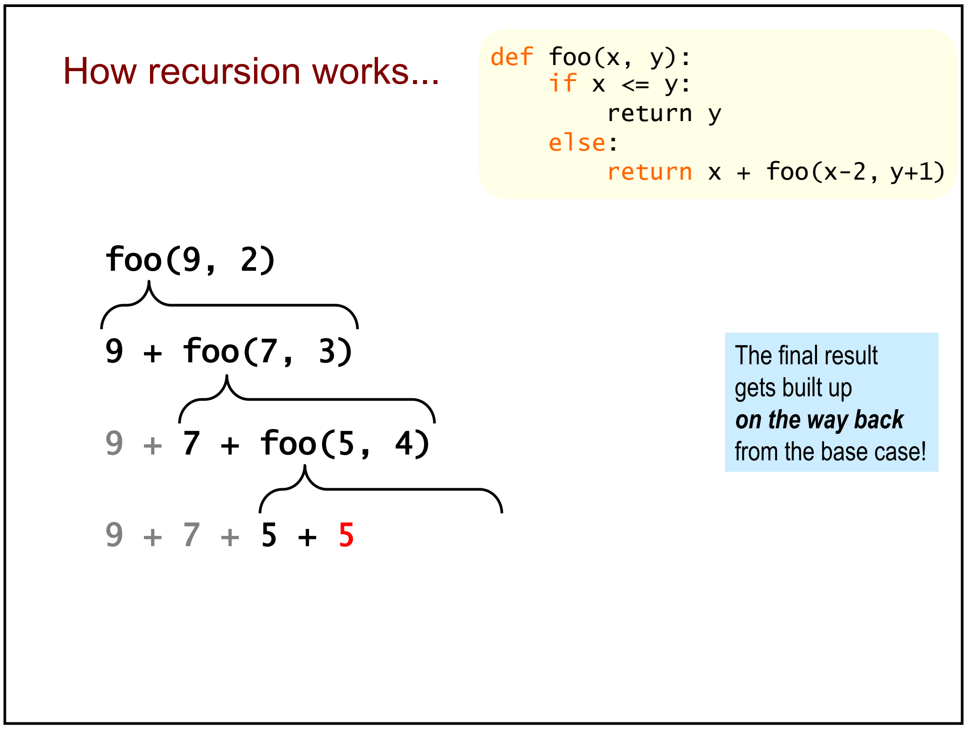
E. 26



How recursion works...

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

```
foo(9, 2)

9 + foo(7, 3)

9 + 7 + foo(5, 4)

9 + 7 + 5 + foo(3, 5)

9 + 7 + 5 + 5
```

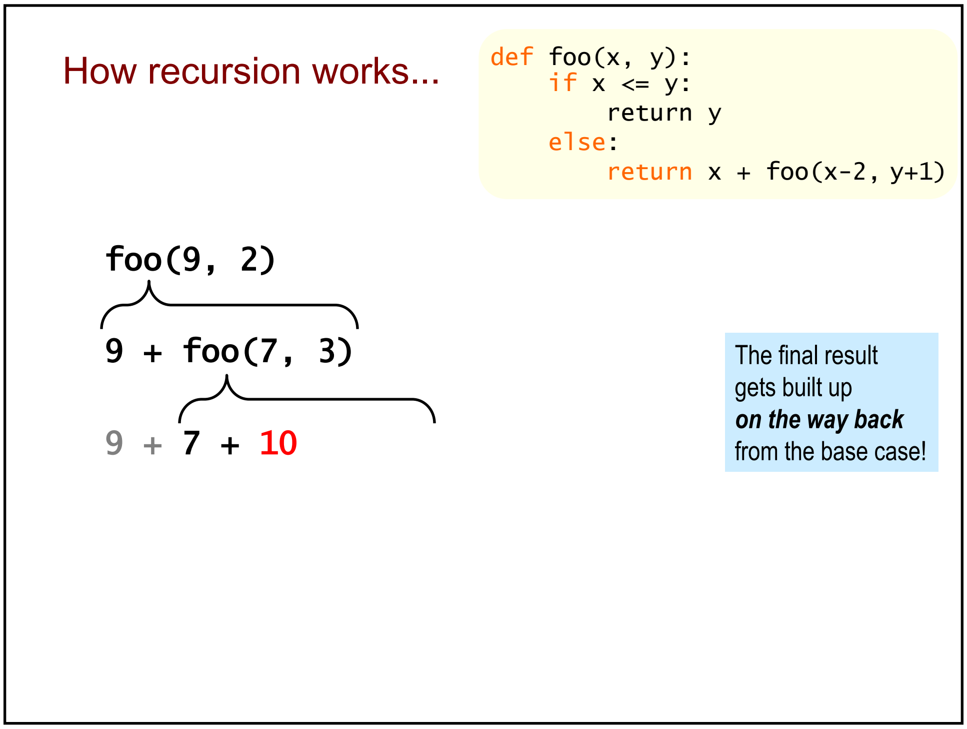


How recursion works...

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

```
foo(9, 2)

9 + foo(7, 3)

9 + 7 + foo(5, 4)

9 + 7 + 5 + 5
```

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

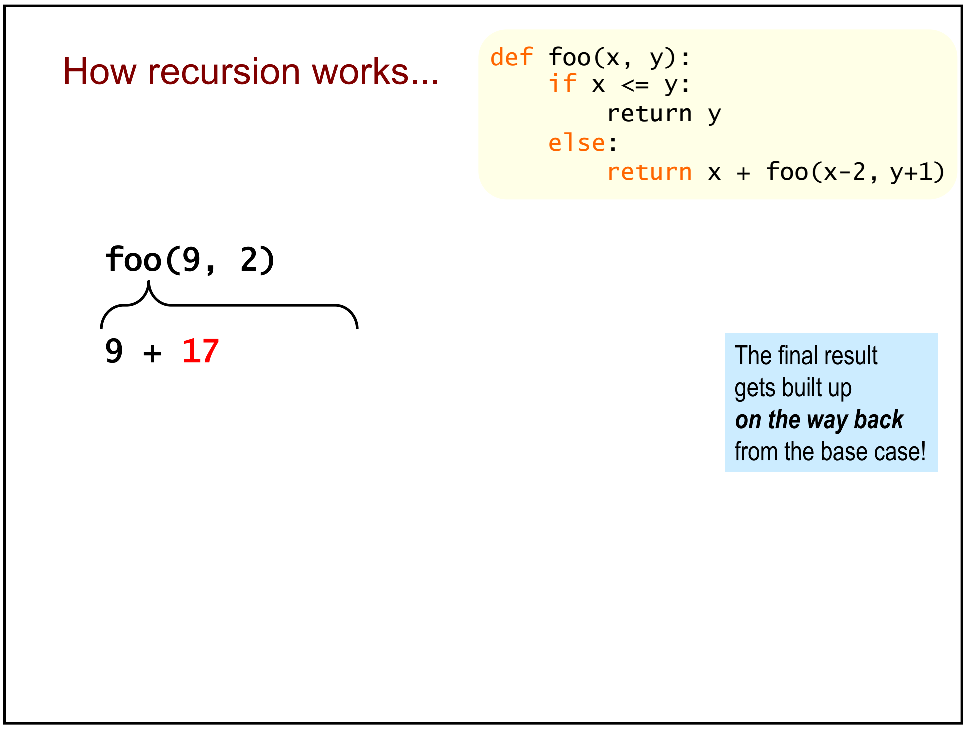


How recursion works...

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

```
foo(9, 2)

9 + foo(7, 3)

9 + 7 + 10
```

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

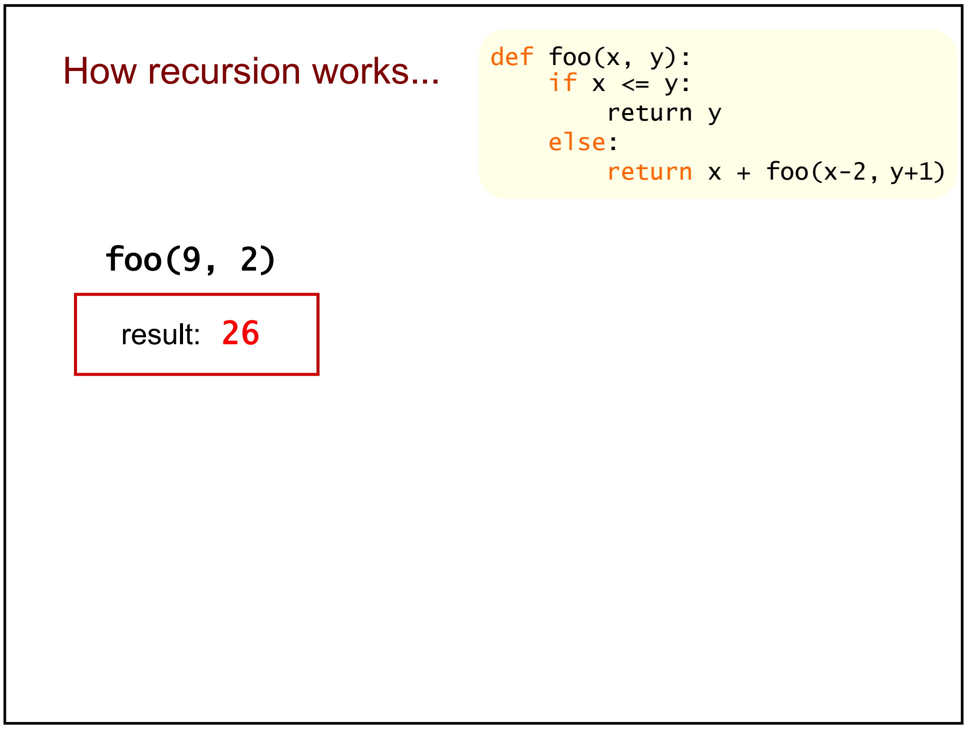


How recursion works...

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

```
foo(9, 2)

9 + 17
```

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

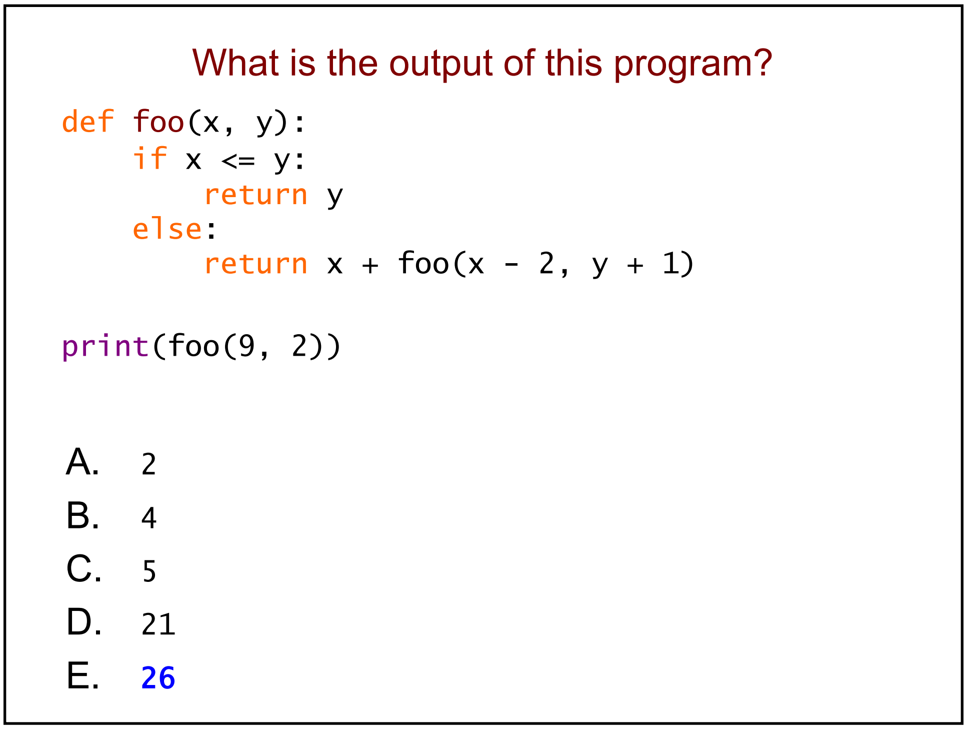


How recursion works...

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

foo(9, 2)

result: **26**



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**

A Recursive Warning!

Hofstadter’s Law:
It always takes longer than you think it will take,
even when you take into account Hofstadter’s Law.

Wrap up PS 1, part I ASAP!
Get started on part II, which is due on Sunday.

Come for help as needed!
See the course website for the office-hour calendar.
Take advantage of Piazza!

Make sure you are getting my BB announcements as emails!

Recall: 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?

Two Approaches to the Same Problem

- Our original version of `power()` uses this definition of b^p:

  $b^p = b * b^{p-1}$ when $p > 0$
  $b^0 = 1$

  - for example:

    $2^{10} = 2 * 2^9$

    $2^9 = 2 * 2^8$

    …

- Each recursive call only reduces the exponent by 1.
- How many times will `power()` be called when computing 2^{1000}?
  1001

Two Approaches to the Same Problem (cont.)

- There's an alternative way to reduce this problem.
- When the exponent is ***even***, we can do this:

  $b^p = (b^{p/2}) * (b^{p/2})$

  - for example:
    $2^{10} = 2^5 * 2^5$

- When the exponent is ***odd***, we can do this:

  $b^p = b * (b^{p/2}) * (b^{p/2})$ (using integer division: `p//2`)

  - for example:
    $2^5 = 2 * 2^2 * 2^2$

- Each recursive call cuts the exponent in half!
- How can we determine if `p` is odd?
  check the value of `p % 2`

A More Efficient Power!

$b^p = (b^{p/2}) * (b^{p/2})$ when p is even and greater than 0

$b^p = b * (b^{p/2}) * (b^{p/2})$ when p is odd and greater than 0

```
def power2(b, p):
    """ docstring goes here...   """
    if p == 0:                   # base case
        return 1
    else:                        # recursive case
        pow_rest = power2(b, p // 2)
        if p % 2 == 0:
            return pow_rest * pow_rest
        else:
            return b * pow_rest * pow_rest
```

A More Efficient Power! (cont.)

- How many times will `power2()` be called when computing 2^{1000}?
  11

```
power2(2, 1000)
  power2(2, 500)
    power2(2, 250)
      power2(2, 125)
        power2(2, 62)
          power2(2, 31)
            power2(2, 15)
              power2(2, 7)
                power2(2, 3)
                  power2(2, 1)
                    power2(2, 0)
```

- Much more efficient than the original `power()` when the starting exponent is large!