Lecture 13: Recursion

CS 1109 Summer 2024

Functions recap

- Functions used to break code into small chunks
  - Helps with legibility and testing
- Variables created in functions have local scope
  - As opposed to global scope variables

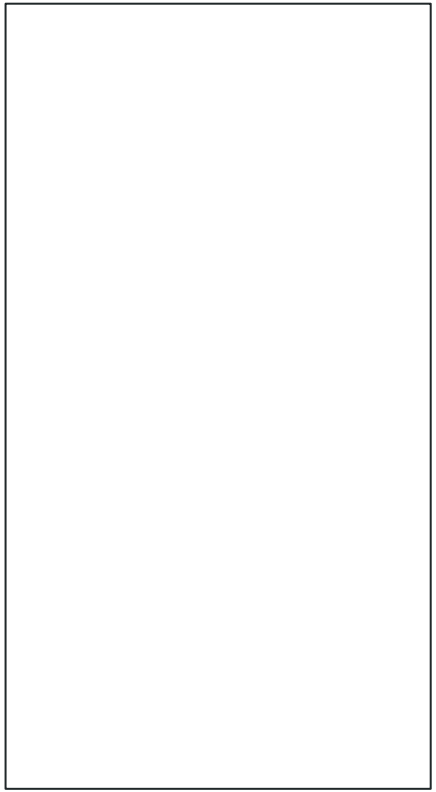
How to visualize function calls

- Every function call takes more memory
  - New arguments and variables must be stored
- Visualize function calls like a **stack** in memory

How to visualize function calls

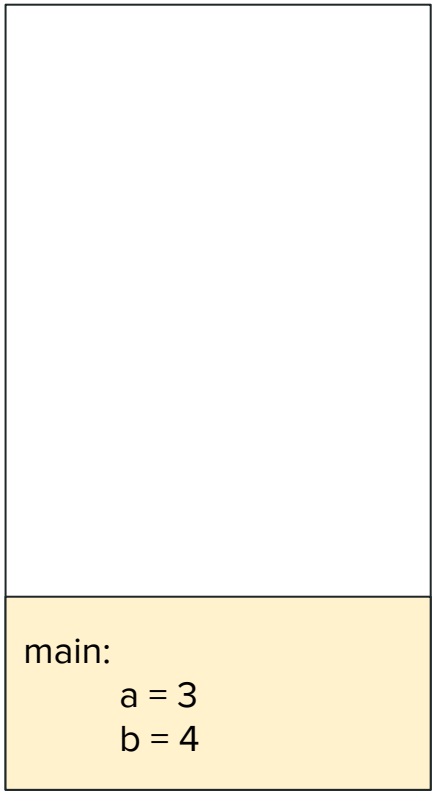
```
def add_numbers(x, y):
    z = x + y
    return z

def main():
    a = 3
    b = 4
    c = add_numbers(a, b)
main()
```

Memory



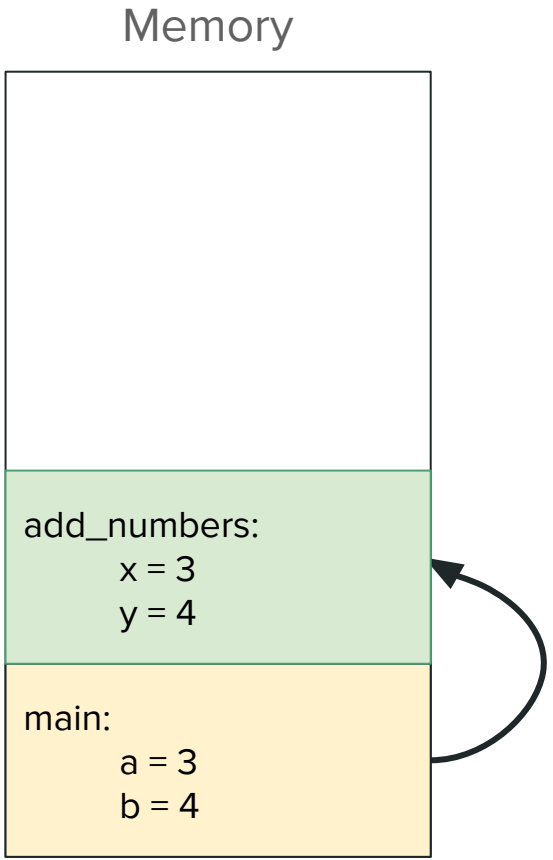
How to visualize function calls

```
def add_numbers(x, y):
    z = x + y
    return z

def main():
    a = 3
    b = 4
    c = add_numbers(a, b)
main()
```



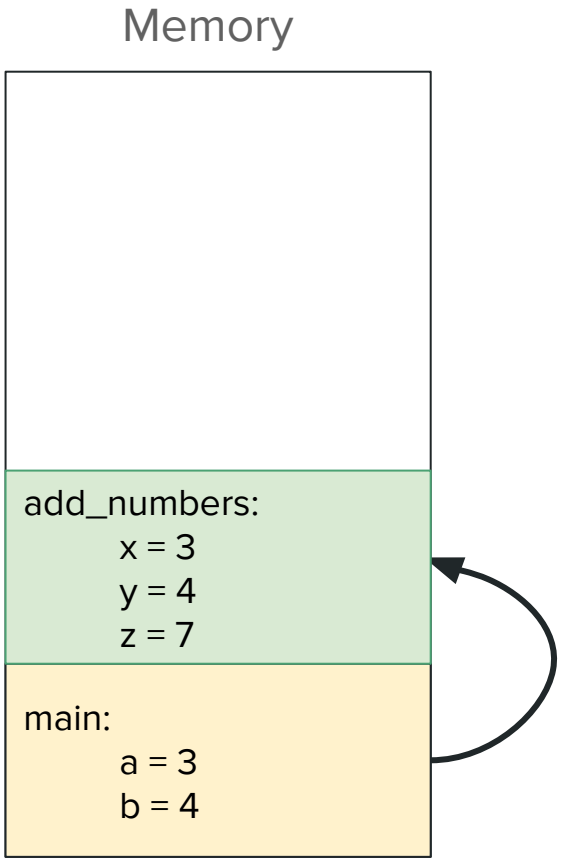
How to visualize function calls

```
def add_numbers(x, y):
    z = x + y
    return z

def main():
    a = 3
    b = 4
    c = add_numbers(a, b)
main()
```



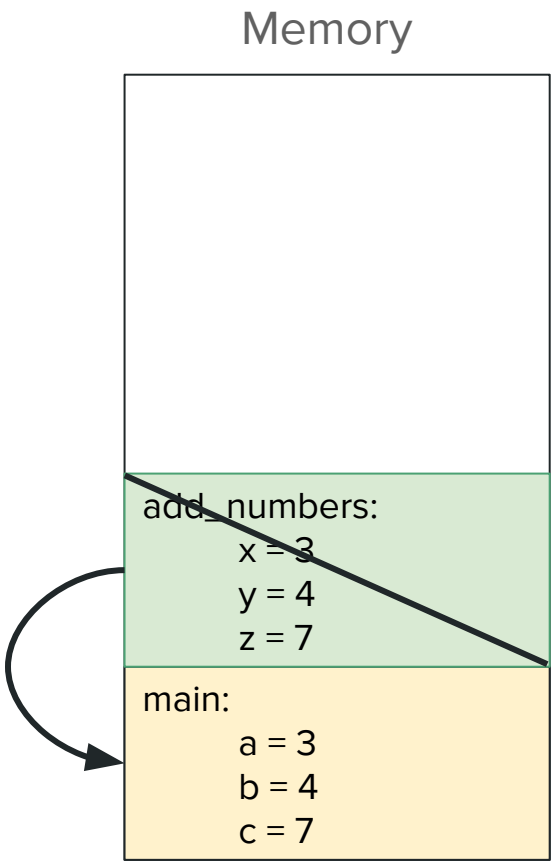
How to visualize function calls

```
def add_numbers(x, y):
    z = x + y
    return z

def main():
    a = 3
    b = 4
    c = add_numbers(a, b)
main()
```



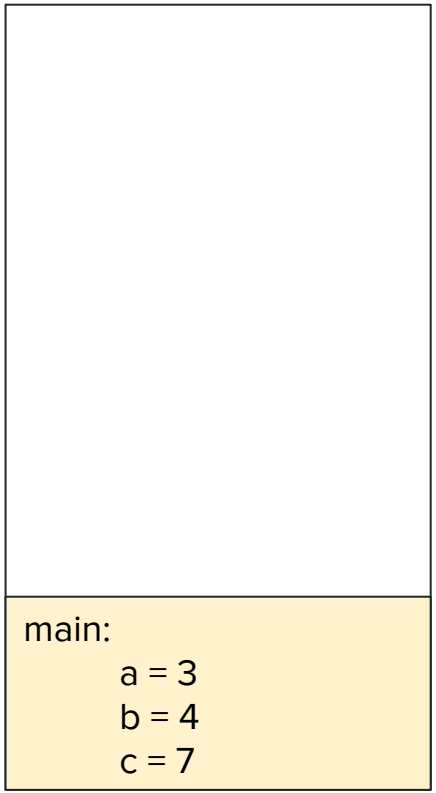
How to visualize function calls

```
def add_numbers(x, y):
    z = x + y
    return z

def main():
    a = 3
    b = 4
    c = add_numbers(a, b)
main()
```



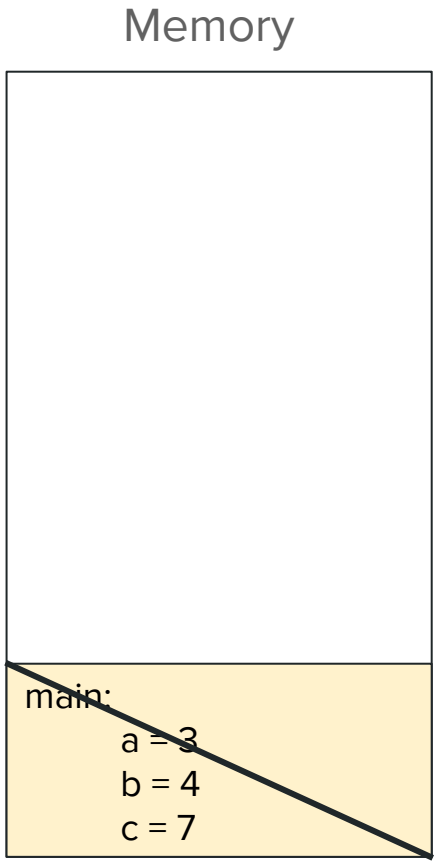
How to visualize function calls

```
def add_numbers(x, y):
    z = x + y
    return z

def main():
    a = 3
    b = 4
    c = add_numbers(a, b)
main()
```



How to visualize function calls

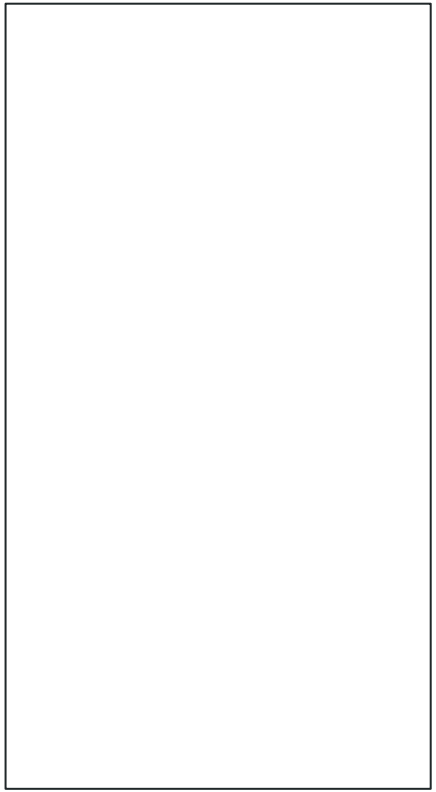
```
def add_numbers(x, y):
    z = x + y
    return z

def main():
    a = 3
    b = 4
    c = add_numbers(a, b)
main()
```



How to visualize function calls

```
def add_numbers(x, y):
    z = x + y
    return z

def main():
    a = 3
    b = 4
    c = add_numbers(a, b)
main()
```

Memory

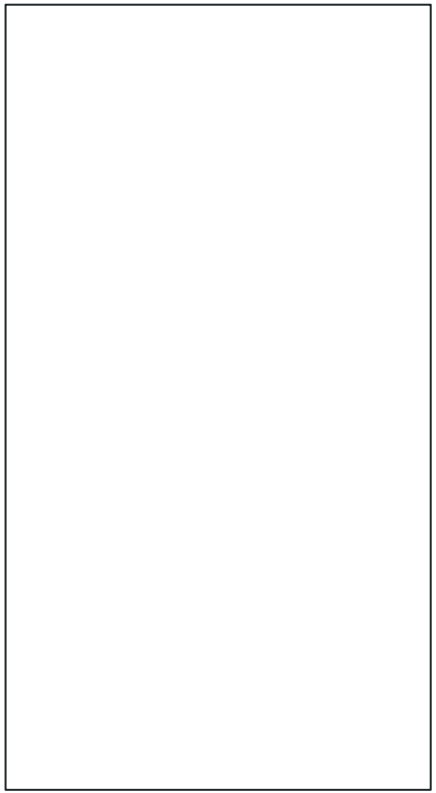


What about nested function calls?

What about nested function calls?

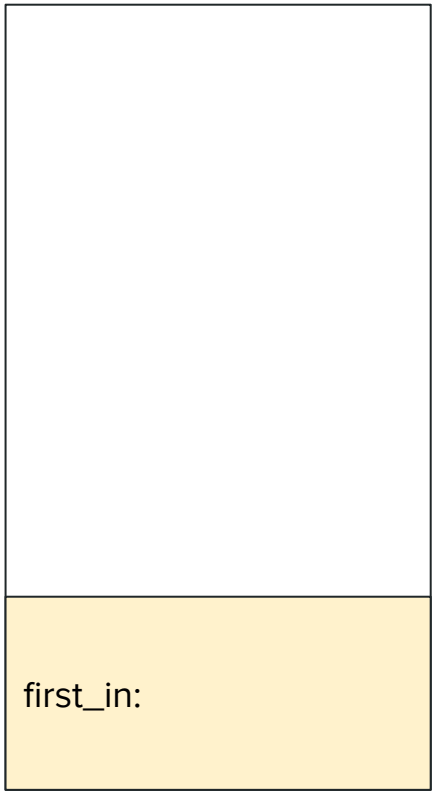
```
def last_in():
    z = 2
    return z

def next_in():
    y = last_in()
    return y

def first_in():
    x = next_in()
first_in()
```

Memory



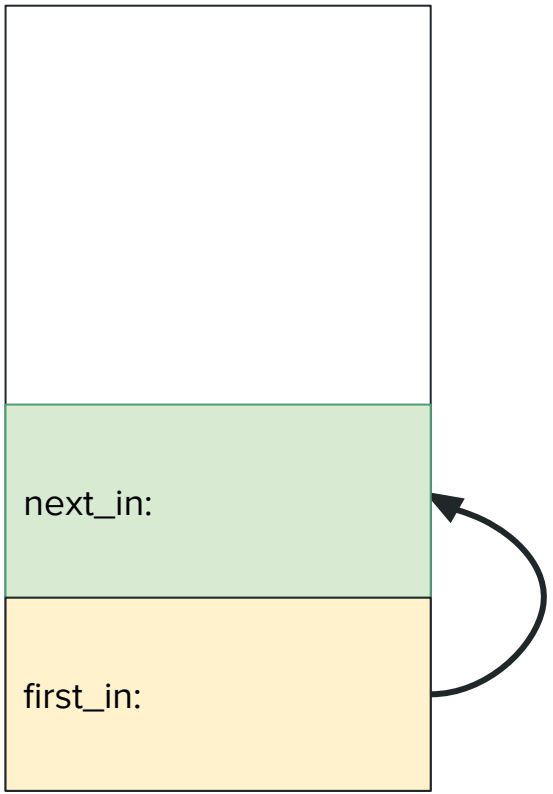
What about nested function calls?

```
def last_in():
    z = 2
    return z

def next_in():
    y = last_in()
    return y

def first_in():
    x = next_in()
first_in()
```



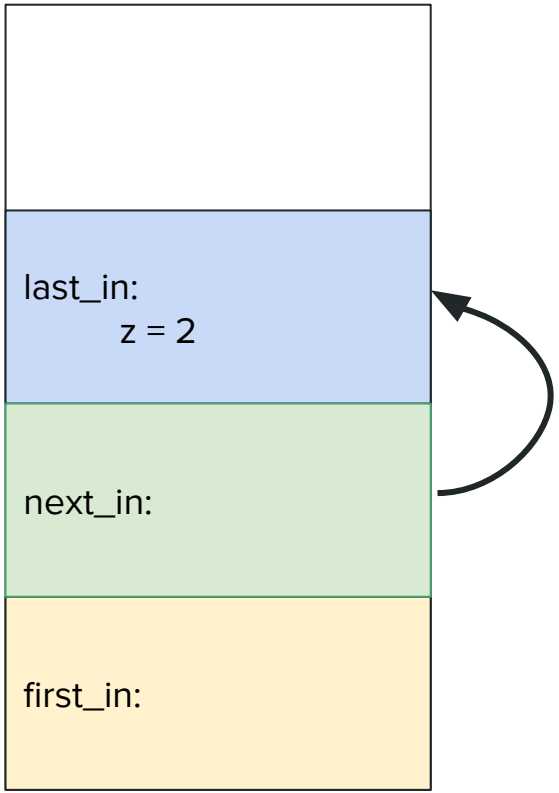
What about nested function calls?

```
def last_in():
    z = 2
    return z

def next_in():
    y = last_in()
    return y

def first_in():
    x = next_in()
first_in()
```



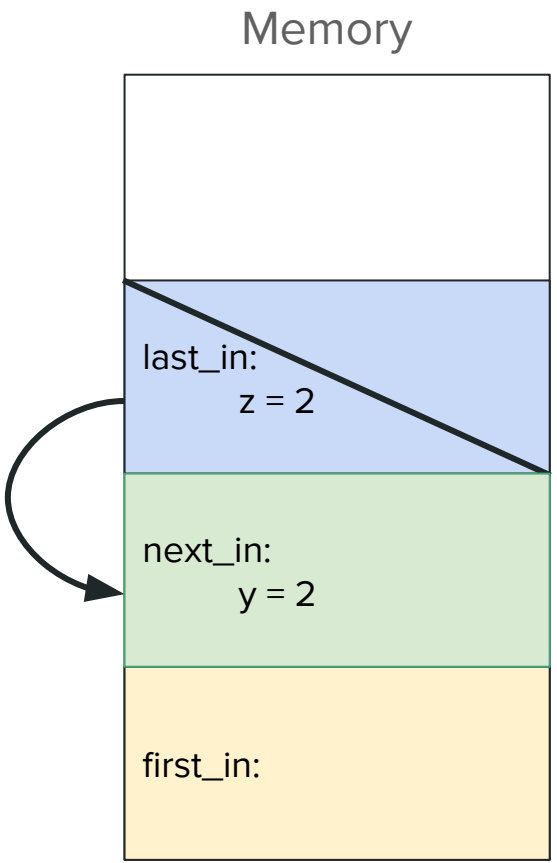
What about nested function calls?

```
def last_in():
    z = 2
    return z

def next_in():
    y = last_in()
    return y

def first_in():
    x = next_in()
first_in()
```



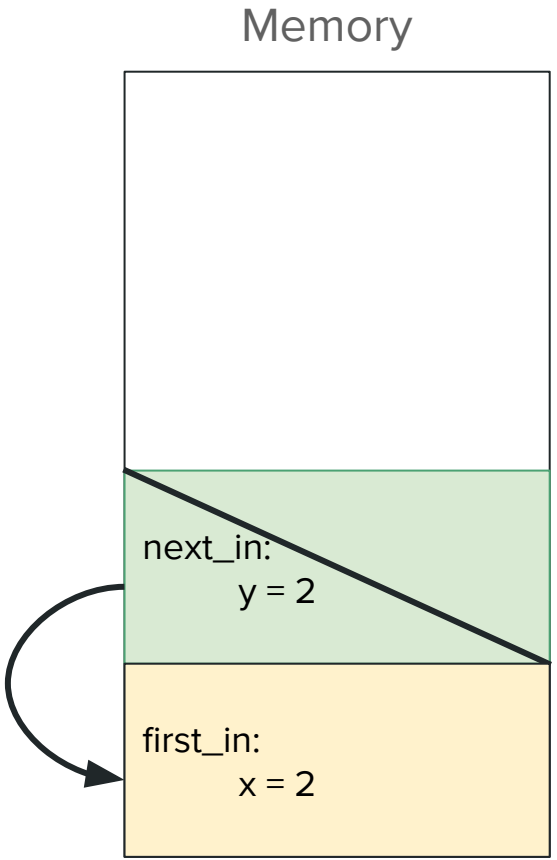
What about nested function calls?

```
def last_in():
    z = 2
    return z

def next_in():
    y = last_in()
    return y

def first_in():
    x = next_in()
first_in()
```



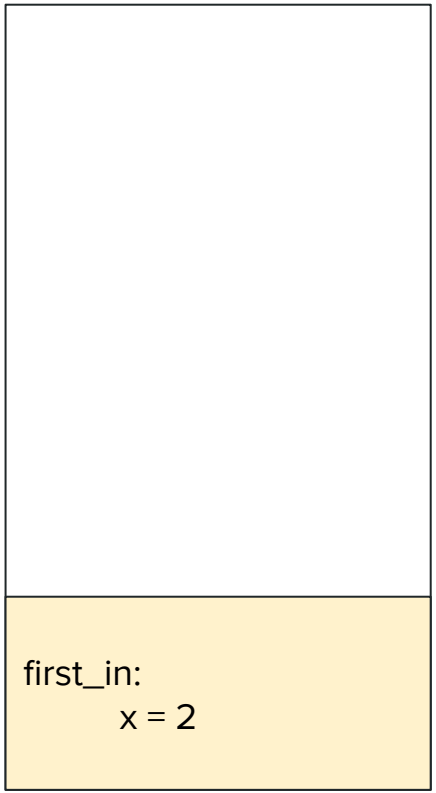
What about nested function calls?

```
def last_in():
    z = 2
    return z

def next_in():
    y = last_in()
    return y

def first_in():
    x = next_in()
first_in()
```



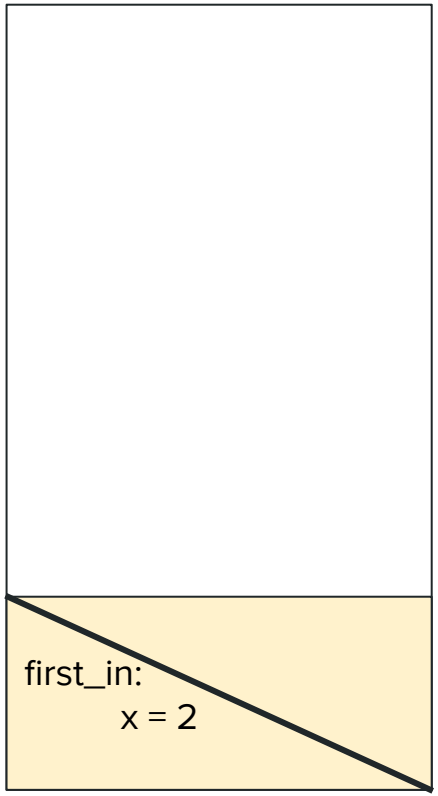
What about nested function calls?

```
def last_in():
    z = 2
    return z

def next_in():
    y = last_in()
    return y

def first_in():
    x = next_in()
first_in()
```



What about nested function calls?

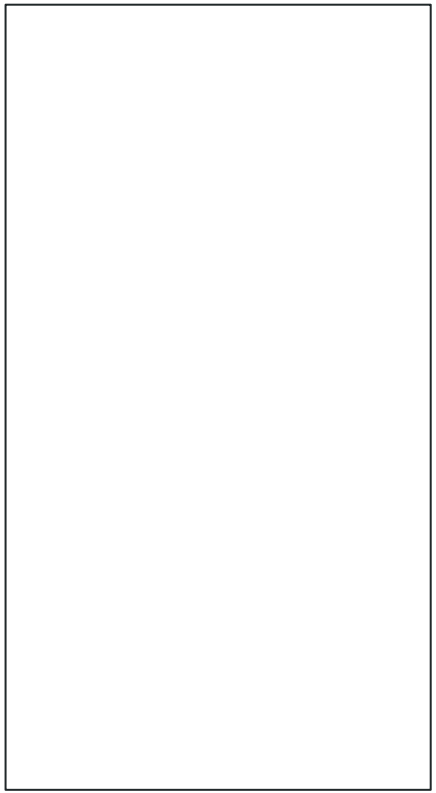
```
def last_in():
    z = 2
    return z

def next_in():
    y = last_in()
    return y

def first_in():
    x = next_in()
first_in()
```



What about nested function calls?

```
def last_in():
    z = 2
    return z

def next_in():
    y = last_in()
    return y

def first_in():
    x = next_in()
first_in()
```

Memory



Call Stack

- Function calls “reside” in memory called the “call stack”
- Stack is LIFO (last in, first out)
  - Think of stacking plates as you clean them; grab top plate when you use one
- Too many nested functions can trigger out-of-memory errors
  - Called “stack overflow”

Call Stack

- Every error you've seen has a stack trace
- Shows which function error was in and all parent (caller) functions

```
def last_in():
    some_num = int("a")
    return some_num

def next_in():
    y = last_in()
    return y

def first_in():
    x = next_in()
first_in()
```

```
Traceback (most recent call last):
  File "/usr/lib/python3.8/idlelib/run.py", line 559, in runcode
    exec(code, self.locals)
  File "/home/kevinnegy/test.py", line 11, in <module>
    first_in()
  File "/home/kevinnegy/test.py", line 10, in first_in
    x = next_in()
  File "/home/kevinnegy/test.py", line 6, in next_in
    y = last_in()
  File "/home/kevinnegy/test.py", line 2, in last_in
    int("a")
ValueError: invalid literal for int() with base 10: 'a'
```

Recursion

- What if a function calls itself?
- Possible because each instance (not related to classes) of function exists separately in memory
- Recursion - when a function calls itself, normally to solve a problem
- Good for divide-and-conquer problems
    - Split the problem into small steps
    - Each function solves one step

Recursion - Example

- Print a countdown from 20 and then print “Liftoff!”

```
def countdown(time):
    while time >= 0:
        print(time)
        time -= 1
    print("Liftoff!")
countdown(20)
```

Recursion - Example

- Print a countdown from 20 and then print “Liftoff!”
- Can we use recursion to solve problem?

```
def countdown(time):
    while time >= 0:
        print(time)
        time -= 1
    print("Liftoff!")
countdown(20)
```

Recursion - Example

- Print a countdown from 20 and then print “Liftoff!”
- First: slice problem up into steps
  - Each step will be handled by one function call
  - Natural step here is one time tick down and a print

```
def countdown(time):
    while time >= 0:
        print(time)
        time -= 1
    print("Liftoff!")
countdown(20)
```

Recursion - Example

- Print a countdown from 20 and then print “Liftoff!”
- First: slice problem up into steps
  - Each step will be handled by one function call
  - Natural step here is one time tick down and a print
- Second: when will recursion end? (i.e. when do we stop?)
  - When `time < 0`

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
```

Recursion - Example

- Print a countdown from 20 and then print “Liftoff!”
- First: slice problem up into steps
  - Each step will be handled by one function call
  - Natural step here is one time tick down and a print
- Second: when will recursion end? (i.e. when do we stop?)
  - When `time < 0`
- Next: write down the normal step

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
```

Recursion - Example

- Print a countdown from 20 and then print “Liftoff!”
- First: slice problem up into steps
  - Each step will be handled by one function call
  - Natural step here is one time tick down and a print
- Second: when will recursion end? (i.e. when do we stop?)
  - When `time < 0`
- Next: write down the normal step
- Last: write down recursive function call

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
```

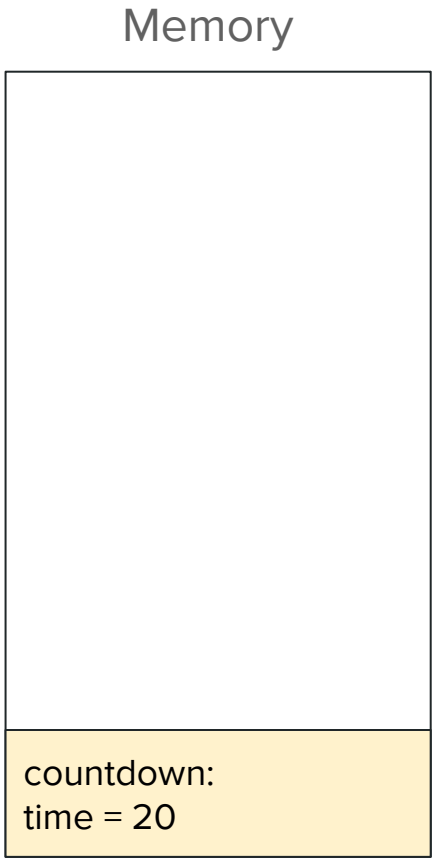
Recursion - Example

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```

```
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Liftoff!
```

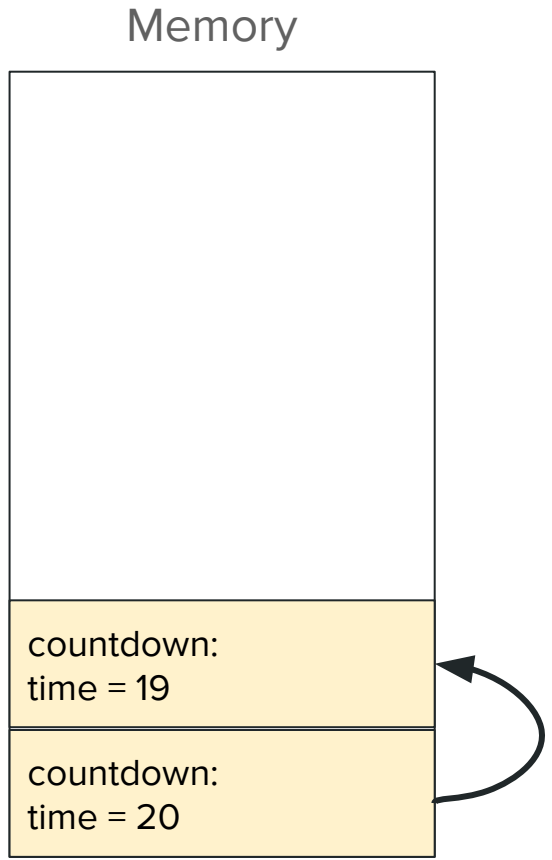
Recursion - Example

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```



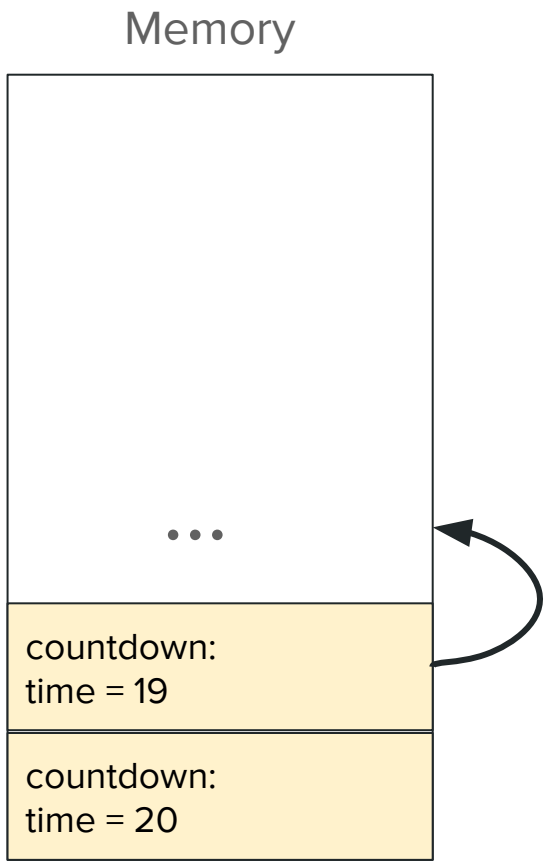
Recursion - Example

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```



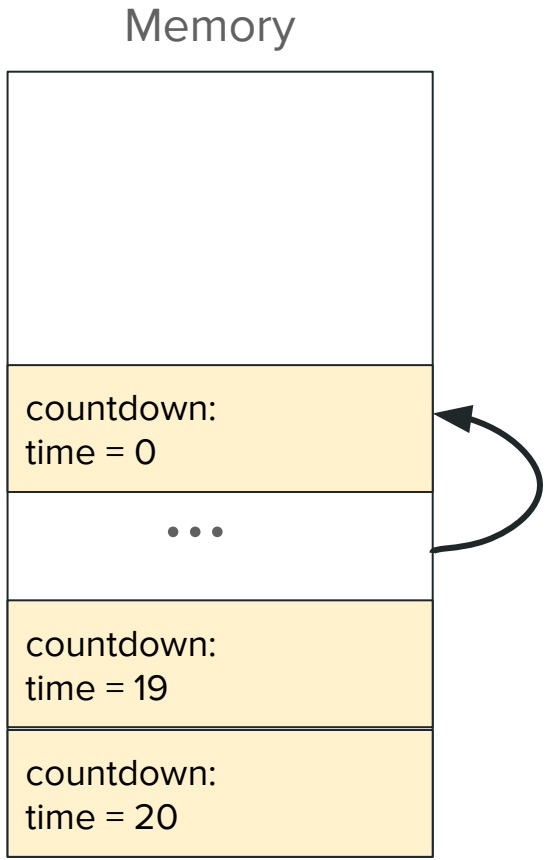
Recursion - Example

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```



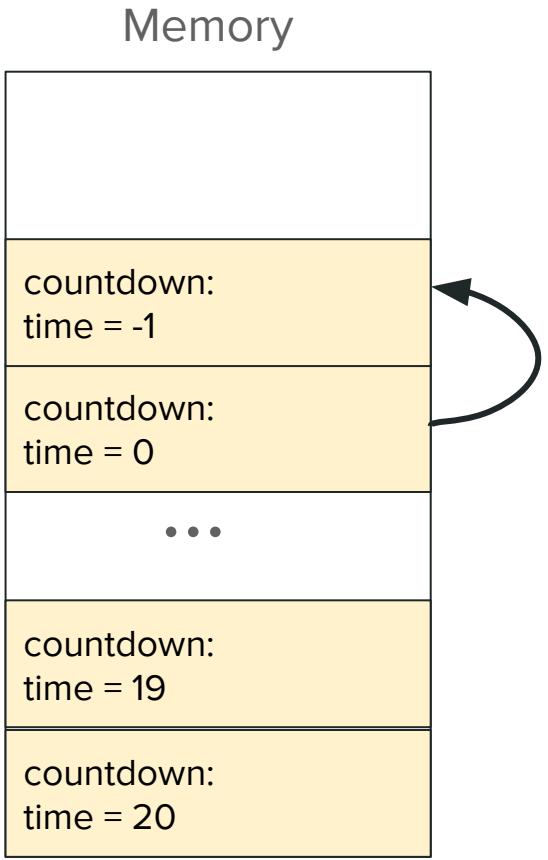
Recursion - Example

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```



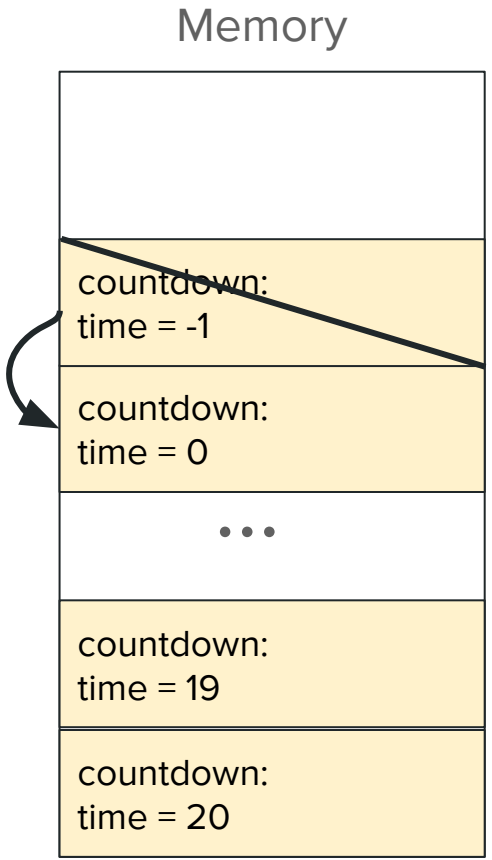
Recursion - Example

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```



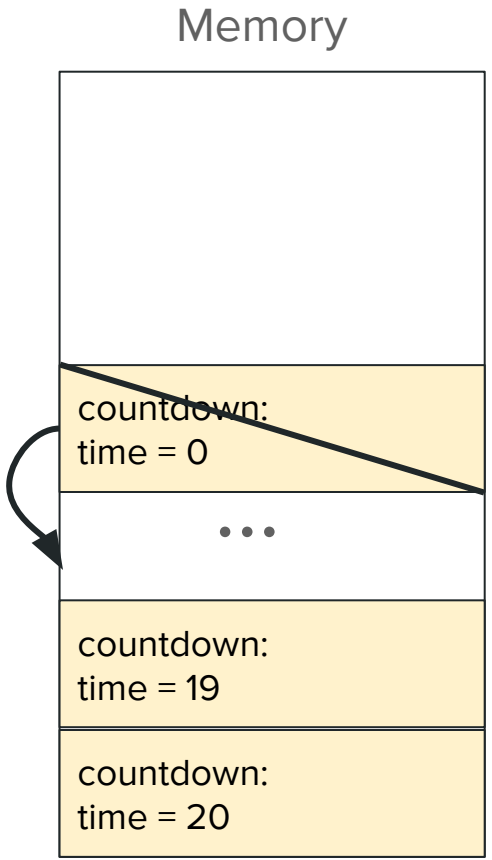
Recursion - Example

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```



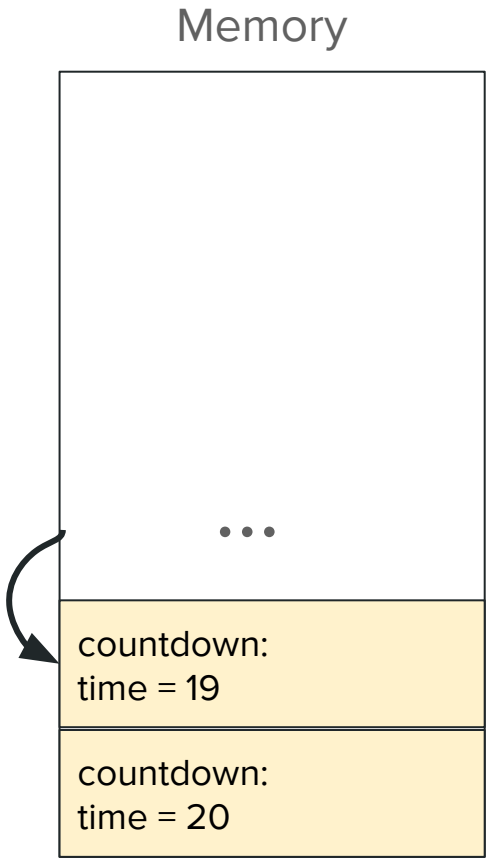
Recursion - Example

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```



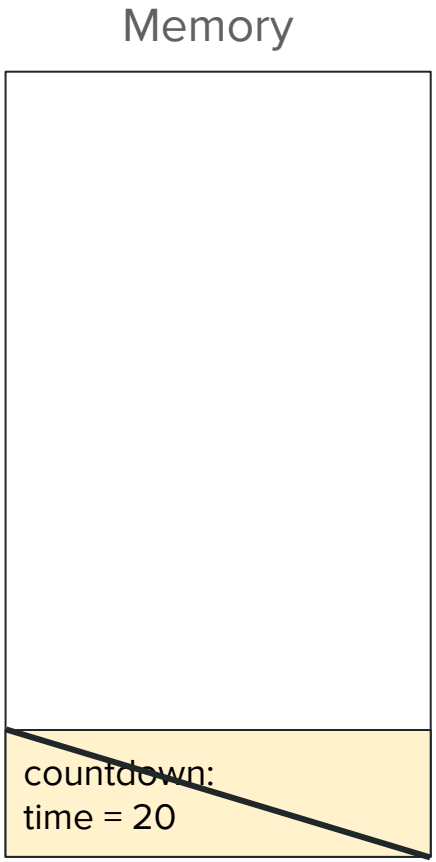
Recursion - Example

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```



Recursion - Example

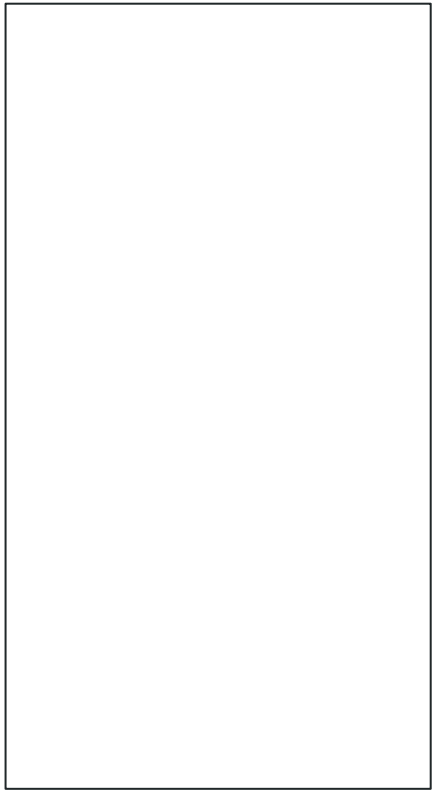
```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```



Recursion - Example

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```

Memory



Recursion - Example

```
def countdown(time):
    while time >= 0:
        print(time)
        time -= 1
    print("Liftoff!")
countdown(20)
```

Iterative

```
def countdown(time):
    if time < 0:
        print("Liftoff!")
        return
    print(time)
    countdown(time-1)
countdown(20)
```

Recursive

Recursion - In-class activity

```
def powers_of_two(exponent):
    result = 1
    for i in range(exponent):
        result *= 2
    return result
```

Iterative

Recursion - In-class activity

```
def powers_of_two(exponent):
    result = 1
    for i in range(exponent):
        result *= 2
    return result
```

Iterative

```
def powers_of_two(exponent):
    if exponent == 0:
        return 1
    return 2 * powers_of_two(exponent-1)
```

Recursive

Recursion - In-class activity

```
def powers_of_two(exponent):
    if exponent == 0:
        return 1
    return 2 * powers_of_two(exponent-1)
```

Recursive

Recursion - Example 2

- Add up all the numbers from 0 to 9

```
def add_numbers(num):
    total = 0
    for i in range(num):
        total += i
    return total
```

Recursion - Example 2

- Add up all the numbers from 0 to 9
- First - break problem into steps and define one step
  - Add `i`

```
def add_numbers(num):
    total = 0
    for i in range(num):
        total += i
    return total
```

Recursion - Example 2

- Add up all the numbers from 0 to 9
- First - break problem into steps and define one step
  - Add `current_num`
- Second - when should we stop?
  - Count backwards and stop when `current_num` is 0

```
def add_numbers(current_num):
    if current_num == 0:
```

Recursion - Example 2

- Add up all the numbers from 0 to 9
- First - break problem into steps and define one step
  - Add `current_num`
- Second - when should we stop?
  - Count backwards and stop when `current_num` is 0
  - Come back to this

```
def add_numbers(current_num):
    if current_num == 0:
```

Recursion - Example 2

- Add up all the numbers from 0 to 9
- First - break problem into steps and define one step
  - Add `current_num`
- Second - when should we stop?
  - Count backwards and stop when `current_num` is 0
  - Come back to this
- Third - write down normal step

```
def add_numbers(current_num):
    if current_num == 0:
        [redacted]
    return current_num + [redacted]
```

Recursion - Example 2

- Add up all the numbers from 0 to 9
- First - break problem into steps and define one step
  - Add `current_num`
- Second - when should we stop?
  - Count backwards and stop when `current_num` is 0
  - Come back to this
- Third - write down normal step
- Last - call function again with a smaller number

```
def add_numbers(current_num):
    if current_num == 0:
        
    return current_num + add_numbers(current_num - 1)
```

Recursion - Example 2

- Add up all the numbers from 0 to 9
- What should we do when we need to stop?

```
def add_numbers(current_num):
    if current_num == 0:
        
    return current_num + add_numbers(current_num - 1)
```

Recursion - Example 2

- Add up all the numbers from 0 to 9
- What should we do when we need to stop?
- Cannot just `return` since return value used in addition

```
def add_numbers(current_num):
    if current_num == 0:
        
    return current_num + add_numbers(current_num - 1)
```

Recursion - Example 2

- Add up all the numbers from 0 to 9
- What should we do when we need to stop?
- Cannot just `return` since return value used in addition
- We should return 0

```
def add_numbers(current_num):
    if current_num == 0:
        
    return current_num + add_numbers(current_num - 1)
```

Recursion - Example 2

- Add up all the numbers from 0 to 9
- What should we do when we need to stop?
- Cannot just `return` since return value used in addition
- We should return 0

```
def add_numbers(current_num):
    if current_num == 0:
        return 0
    return current_num + add_numbers(current_num - 1)
```

Recursion - Example 2

```
def add_numbers(num):
    total = 0
    for i in range(num):
        total += i
    return total
```

Iterative

```
def add_numbers(current_num):
    if current_num == 0:
        return 0
    return current_num + add_numbers(current_num - 1)
```

Recursive

Recursion - Notes

- You have to trust that recursive call will do the rest of work for you
  - You just handle current step
- Most loops can be expressed as a recursive function
- Sometimes recursive function is more legible than for loop
- Sometimes recursive solutions are difficult to express as for loops
- That being said, it's not always good to use recursion
  - Too many recursive calls -> stack overflow
  - Can be harder to read than for loops

```
def stack_overflow():
    stack_overflow()
stack_overflow()
```

Recursion - Example 3

- Find needle in a haystack - recursive solution!

```
def find_needle(haystack):
    for i in range(len(haystack)):
        for j in range(len(haystack[i])):
            if haystack[i][j] == '/':
                return [i, j]
```

Recursion - Example 3

- Find needle in a haystack - recursive solution!
- First - what is our normal step?
  - Check if the current indices contain the needle
- Second - when will we stop?
  - When `i` and `j` are outside of valid index ranges
  - If we find needle

```
def find_needle(haystack):
    for i in range(len(haystack)):
        for j in range(len(haystack[i])):
            if haystack[i][j] == '/':
                return [i, j]
```

Recursion - Example 3

- Find needle in a haystack - recursive solution!
- First - what is our normal step?
  - Check if the current indices contain the needle
- Second - when will we stop?
  - When `i` and `j` are outside of valid index ranges
  - If we find needle

```
def find_needle(haystack, row, col):
    if row >= len(haystack) or col >= len(haystack[0]):
        return None

    if haystack[row][col] == '/':
        return [row, col]
```

Recursion - Example 3

- Find needle in a haystack - recursive solution!
- First - what is our normal step?
  - Check if the current indices contain the needle
- Second - when will we stop?
  - When `i` and `j` are outside of valid index ranges
  - If we find needle
- Third - write normal step (normal step is looking for stopping condition)

```
def find_needle(haystack, row, col):
    if row >= len(haystack) or col >= len(haystack[0]):
        return None

    if haystack[row][col] == '/':
        return [row, col]
```

Recursion - Example 3

- Find needle in a haystack - recursive solution!
- Last - recursive step
  - Need to increment `row` and `col` correctly
  - In nested loop version, only increment `row` if all `col`'s exhausted
  - Do the same here
  - Notice the `col` reset

```
def find_needle(haystack, row, col):
    if row >= len(haystack) or col >= len(haystack[0]):
        return None

    if haystack[row][col] == '/':
        return [row, col]

    # Recursive step
    answer = find_needle(haystack, row, col + 1)
    if answer is None:
        return find_needle(haystack, row+1, 0)
    else:
        return answer
```

Recursion - Example 3

- For loop is simpler here

Recursive

```
def find_needle(haystack, row, col):
    if row >= len(haystack) or col >= len(haystack[0]):
        return None

    if haystack[row][col] == '/':
        return [row, col]

    # Recursive step
    answer = find_needle(haystack, row, col + 1)
    if answer is None:
        return find_needle(haystack, row+1, 0)
    else:
        return answer
```

Iterative

```
def find_needle(haystack):
    for i in range(len(haystack)):
        for j in range(len(haystack[i])):
            if haystack[i][j] == '/':
                return [i, j]
```