Lecture 6: Recursion

Marvin Zhang
06/28/2016

Announcements

Announcements

- Hog is due Thursday! Submit Wednesday for 1 EC point

Announcements

- Hog is due Thursday! Submit Wednesday for 1 EC point
- HW2 is due Wednesday! Submit Wednesday for credit

Announcements

- Hog is due Thursday! Submit Wednesday for 1 EC point
- HW2 is due Wednesday! Submit Wednesday for credit
- Tutors have begun small tutoring sessions!

Announcements

- Hog is due Thursday! Submit Wednesday for 1 EC point
- HW2 is due Wednesday! Submit Wednesday for credit
- Tutors have begun small tutoring sessions!
    - Check Piazza for details

Announcements

- Hog is due Thursday! Submit Wednesday for 1 EC point
- HW2 is due Wednesday! Submit Wednesday for credit
- Tutors have begun small tutoring sessions!
    - Check Piazza for details
- Starting this week, lab assistants are running checkoffs in lab sections!

Announcements

- Hog is due Thursday! Submit Wednesday for 1 EC point
- HW2 is due Wednesday! Submit Wednesday for credit
- Tutors have begun small tutoring sessions!
  - Check Piazza for details
- Starting this week, lab assistants are running checkoffs in lab sections!
  - Talk to a lab assistant for a few minutes about your lab or homework assignment

Announcements

- Hog is due Thursday! Submit Wednesday for 1 EC point
- HW2 is due Wednesday! Submit Wednesday for credit
- Tutors have begun small tutoring sessions!
  - Check Piazza for details
- Starting this week, lab assistants are running checkoffs in lab sections!
  - Talk to a lab assistant for a few minutes about your lab or homework assignment
  - http://cs61a.org/articles/about.html#checkoffs

Announcements

- Hog is due Thursday! Submit Wednesday for 1 EC point
- HW2 is due Wednesday! Submit Wednesday for credit
- Tutors have begun small tutoring sessions!
  - Check Piazza for details
- Starting this week, lab assistants are running checkoffs in lab sections!
  - Talk to a lab assistant for a few minutes about your lab or homework assignment
  - http://cs61a.org/articles/about.html#checkoffs
- Quiz 2 is *this Thursday*

Announcements

- Hog is due Thursday! Submit Wednesday for 1 EC point
- HW2 is due Wednesday! Submit Wednesday for credit
- Tutors have begun small tutoring sessions!
  - Check Piazza for details
- Starting this week, lab assistants are running checkoffs in lab sections!
  - Talk to a lab assistant for a few minutes about your lab or homework assignment
  - http://cs61a.org/articles/about.html#checkoffs
- Quiz 2 is *this Thursday*
  - Topics covered may include environment diagrams and higher–order functions

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Functions), the goals are:

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Functions), the goals are:
  - To understand the idea of *functional abstraction*

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Functions), the goals are:
  - To understand the idea of *functional abstraction*
  - To study this idea through:

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Functions), the goals are:
  - To understand the idea of *functional abstraction*
  - To study this idea through:
    - higher-order functions

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Functions), the goals are:
  - To understand the idea of *functional abstraction*
  - To study this idea through:
    - higher-order functions
    - recursion (today and tomorrow!)

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Functions), the goals are:
  - To understand the idea of *functional abstraction*
  - To study this idea through:
    - higher-order functions
    - recursion (today and tomorrow!)
    - orders of growth

Recursion

Recursion

- A function is *recursive* if the body of that function contains a call to itself

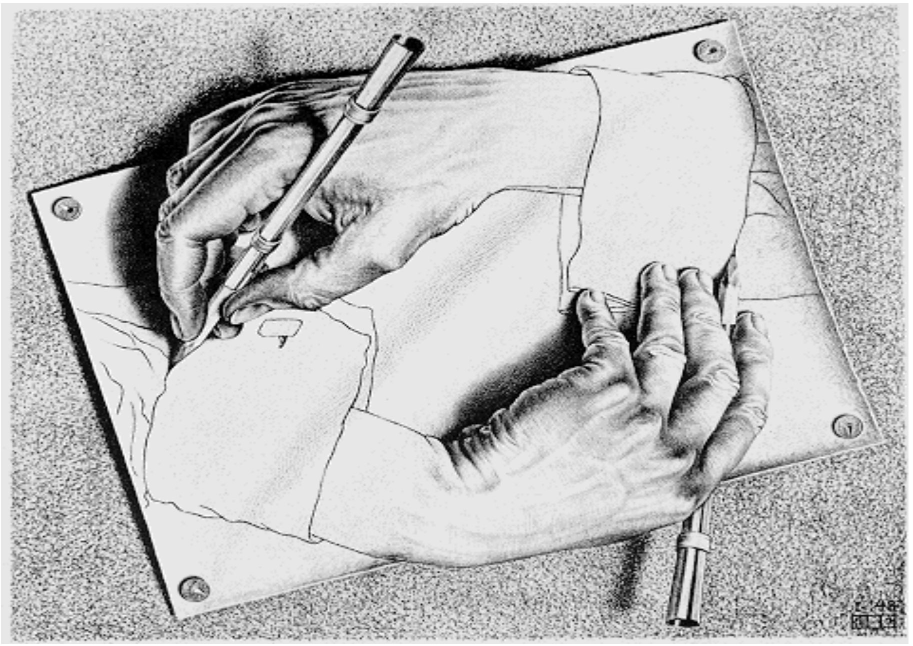
Recursion

- A function is *recursive* if the body of that function contains a call to itself
  - This implies that executing the body of a recursive function may require applying that function

Recursion

- A function is *recursive* if the body of that function contains a call to itself
  - This implies that executing the body of a recursive function may require applying that function

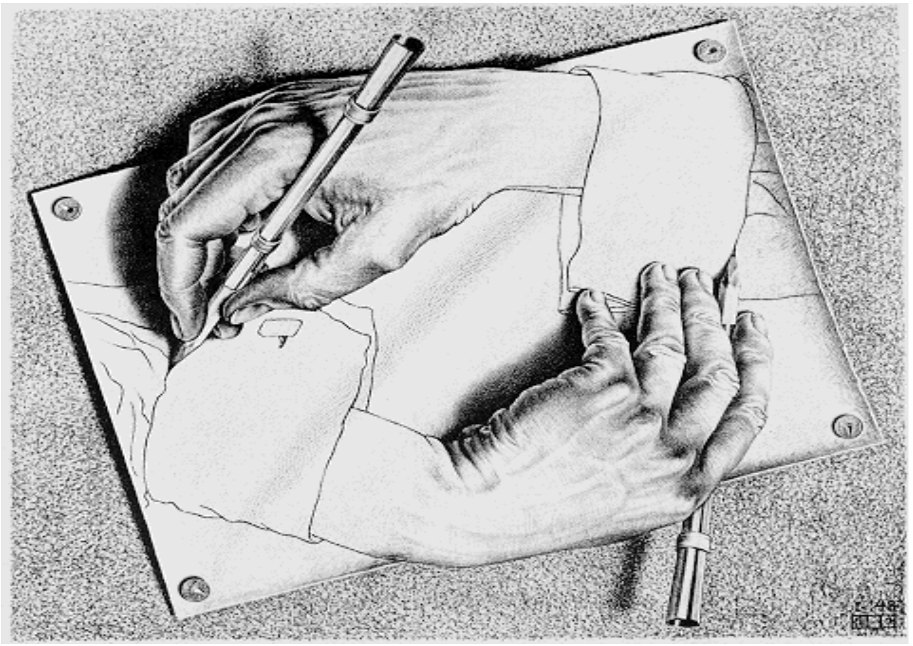
- How is this possible? We’ll see some examples next.



Recursion

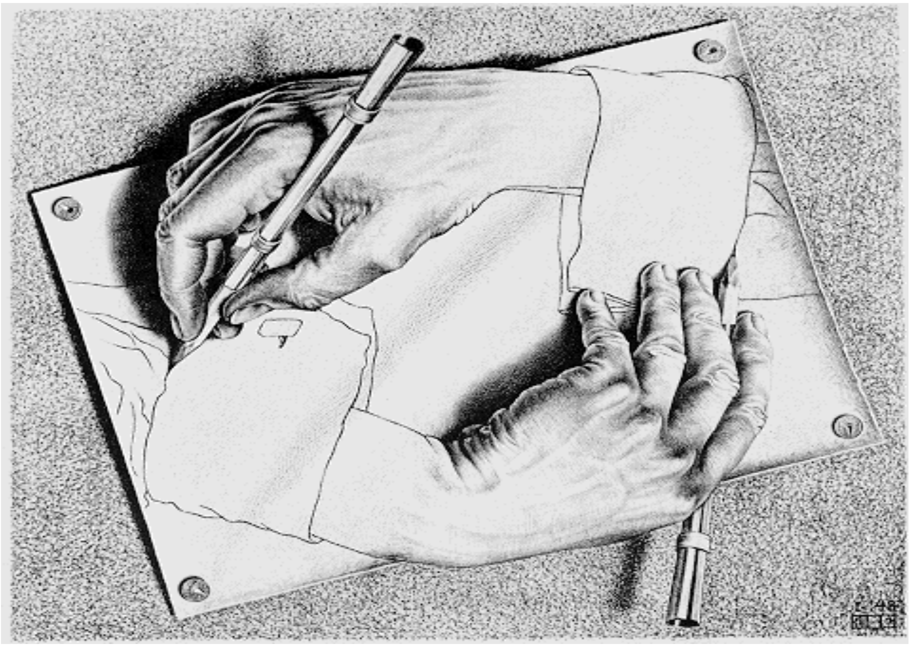
Recursion

- Why would we want to do this?



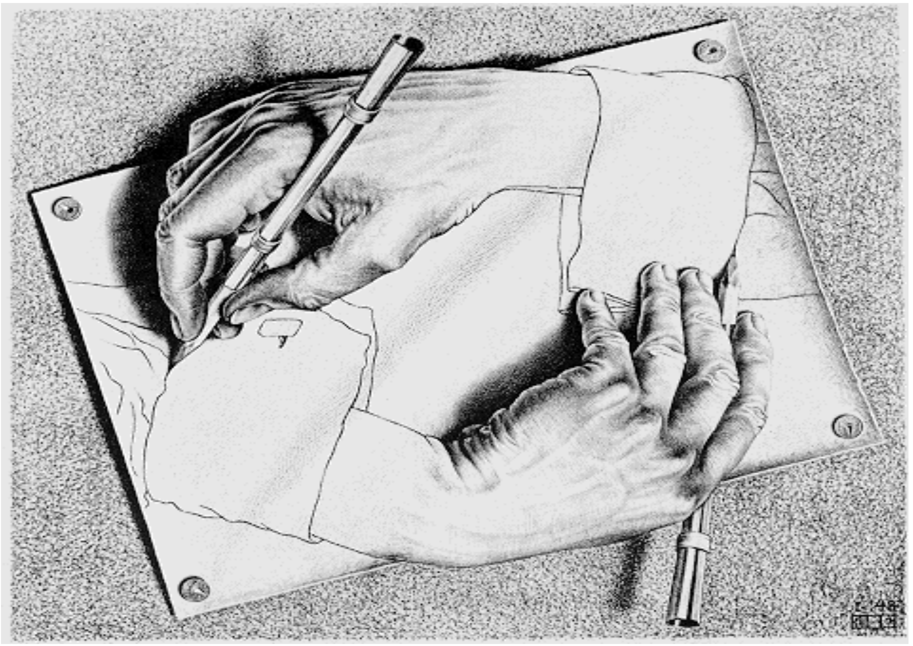
Recursion

- Why would we want to do this?
  - A common problem solving technique is to *break down the problem* into smaller problems that are easier to solve



Recursion

- Why would we want to do this?
  - A common problem solving technique is to *break down the problem* into smaller problems that are easier to solve
  - This is exactly what recursion does!



Recursion

- Why would we want to do this?
  - A common problem solving technique is to *break down the problem* into smaller problems that are easier to solve
  - This is exactly what recursion does!

Recursion

- Why would we want to do this?
    - A common problem solving technique is to *break down the problem* into smaller problems that are easier to solve
    - This is exactly what recursion does!
    - For example, how would you write a function that, given a string, returns the reversed version of the string?

Recursion (demo)

- Why would we want to do this?
  - A common problem solving technique is to *break down the problem* into smaller problems that are easier to solve
  - This is exactly what recursion does!
  - For example, how would you write a function that, given a string, returns the reversed version of the string?

Anatomy of a Recursive Function

```python
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Anatomy of a Recursive Function

- The def statement header is similar to other functions

```
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Anatomy of a Recursive Function

- The def statement header is similar to other functions
- Conditional statements check for base cases

```
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Anatomy of a Recursive Function

- The **def** statement header is similar to other functions
- Conditional statements check for base cases
- Base cases are evaluated without recursive calls

```python
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Anatomy of a Recursive Function

- The **def** statement header is similar to other functions
- Conditional statements check for base cases
- Base cases are evaluated without recursive calls
- Recursive cases are evaluated with recursive calls

```
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

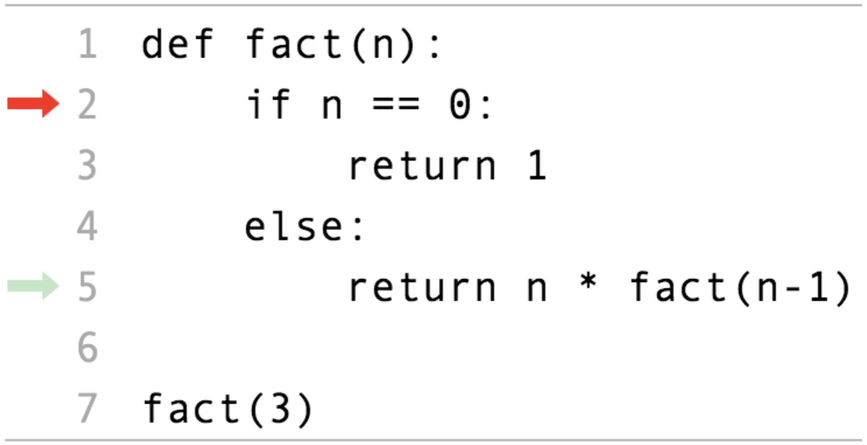
Verifying Correctness

The easy way, and the right way

Recursion in Environment Diagrams

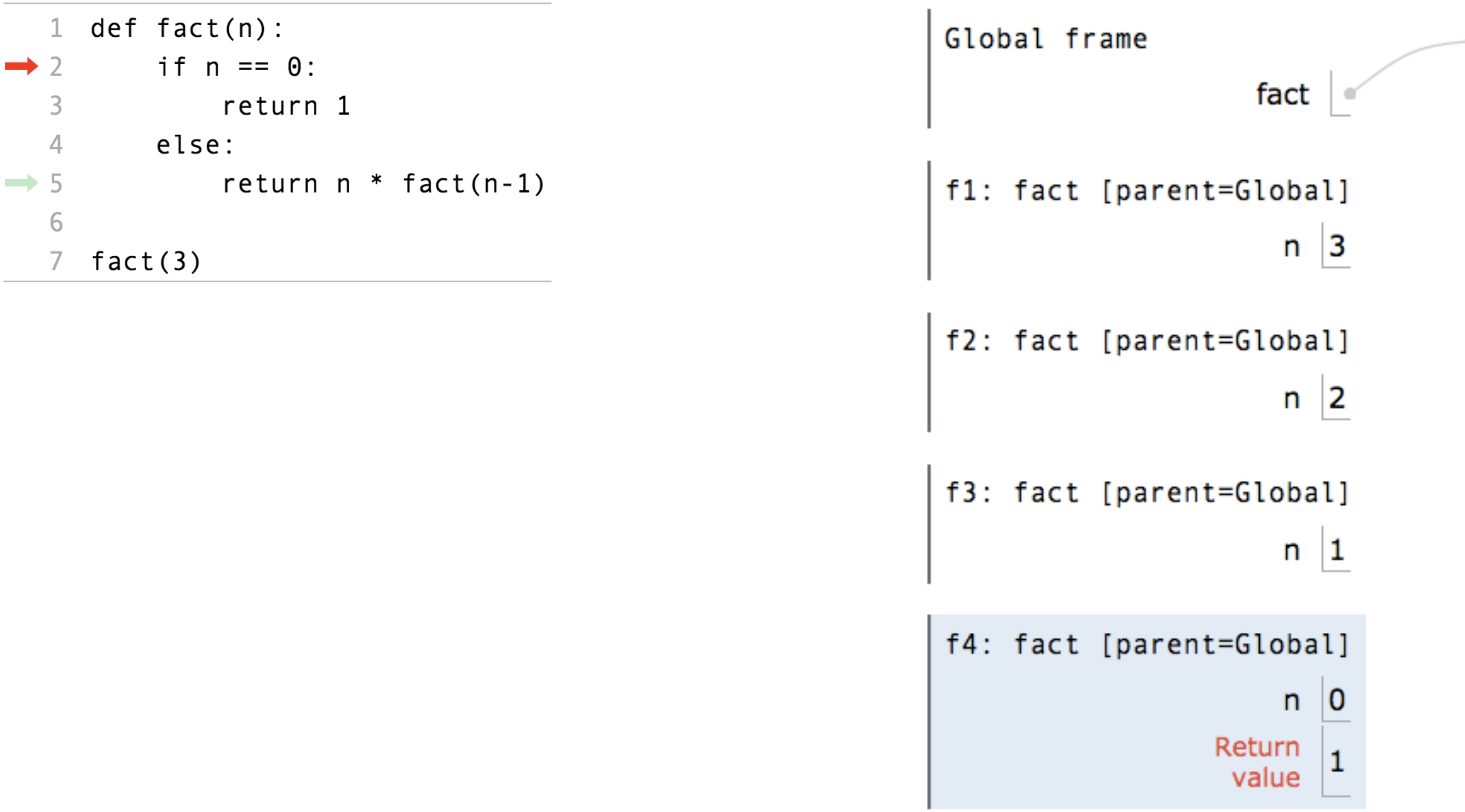
Recursion in Environment Diagrams (demo)

Recursion in Environment Diagrams (demo)



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

fact(3)
```

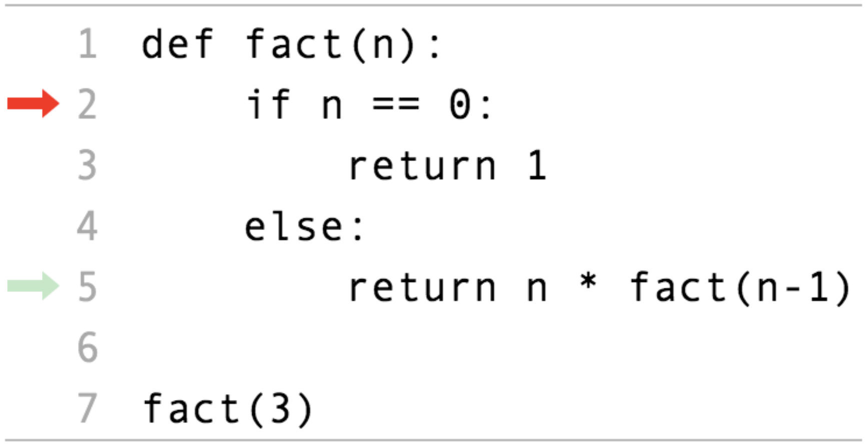
Recursion in Environment Diagrams (demo)



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

fact(3)
```

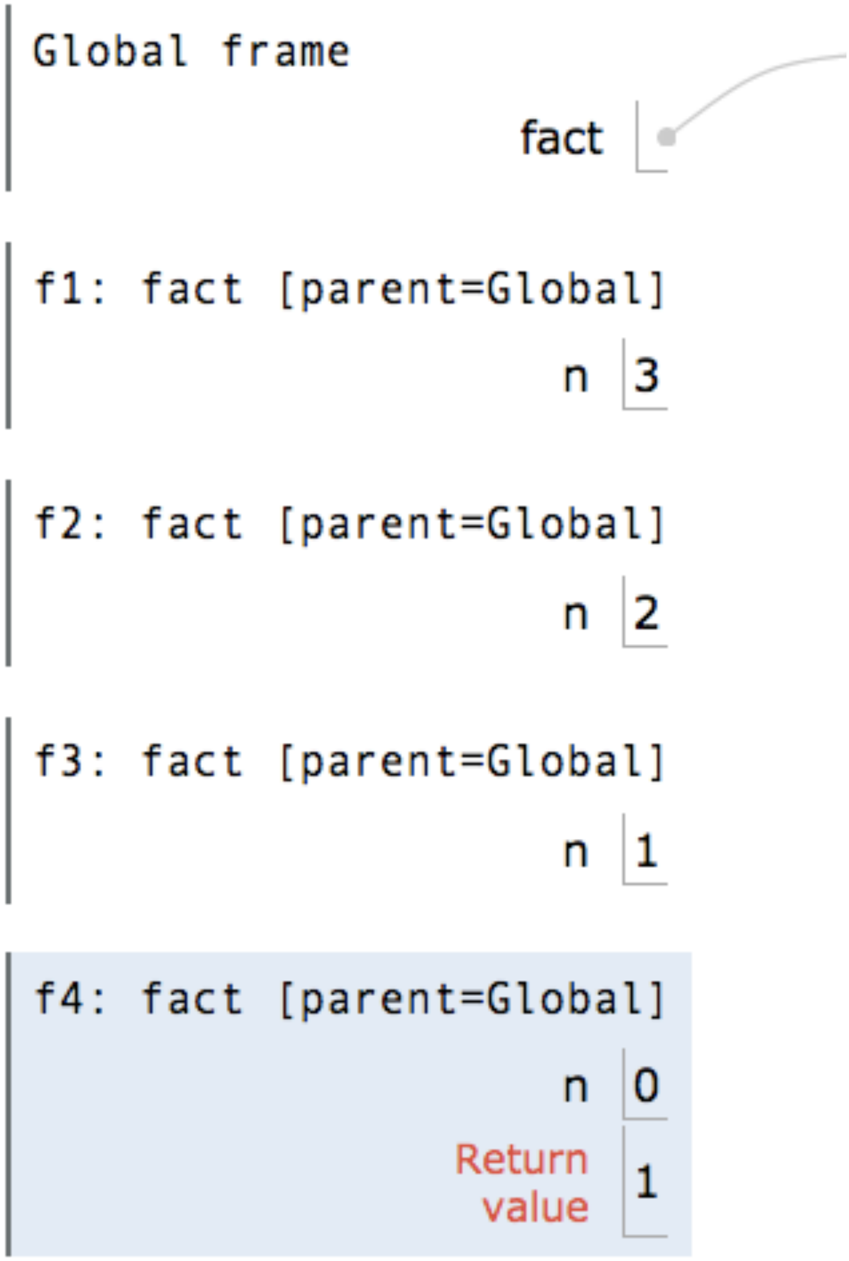
```
Global frame
                  fact  ●

f1: fact [parent=Global]
                     n  3

f2: fact [parent=Global]
                     n  2

f3: fact [parent=Global]
                     n  1

f4: fact [parent=Global]
                     n  0
          Return value  1
```

Recursion in Environment Diagrams (demo)

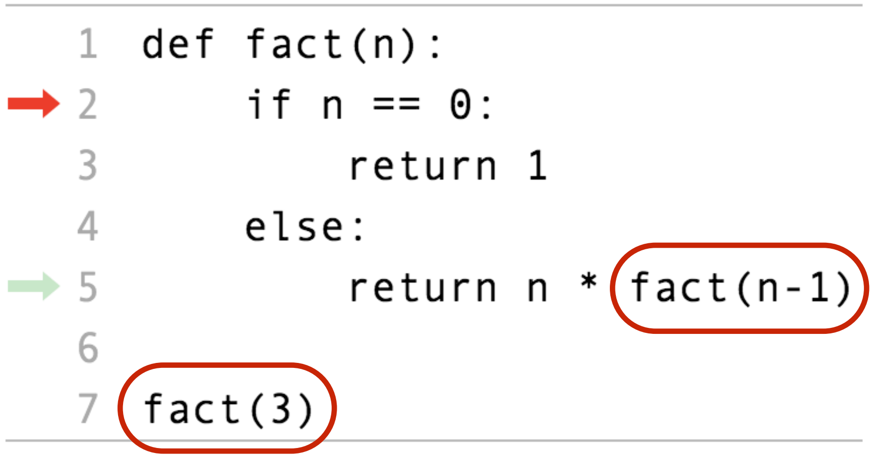


```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
```

- The same function `fact` is called multiple times

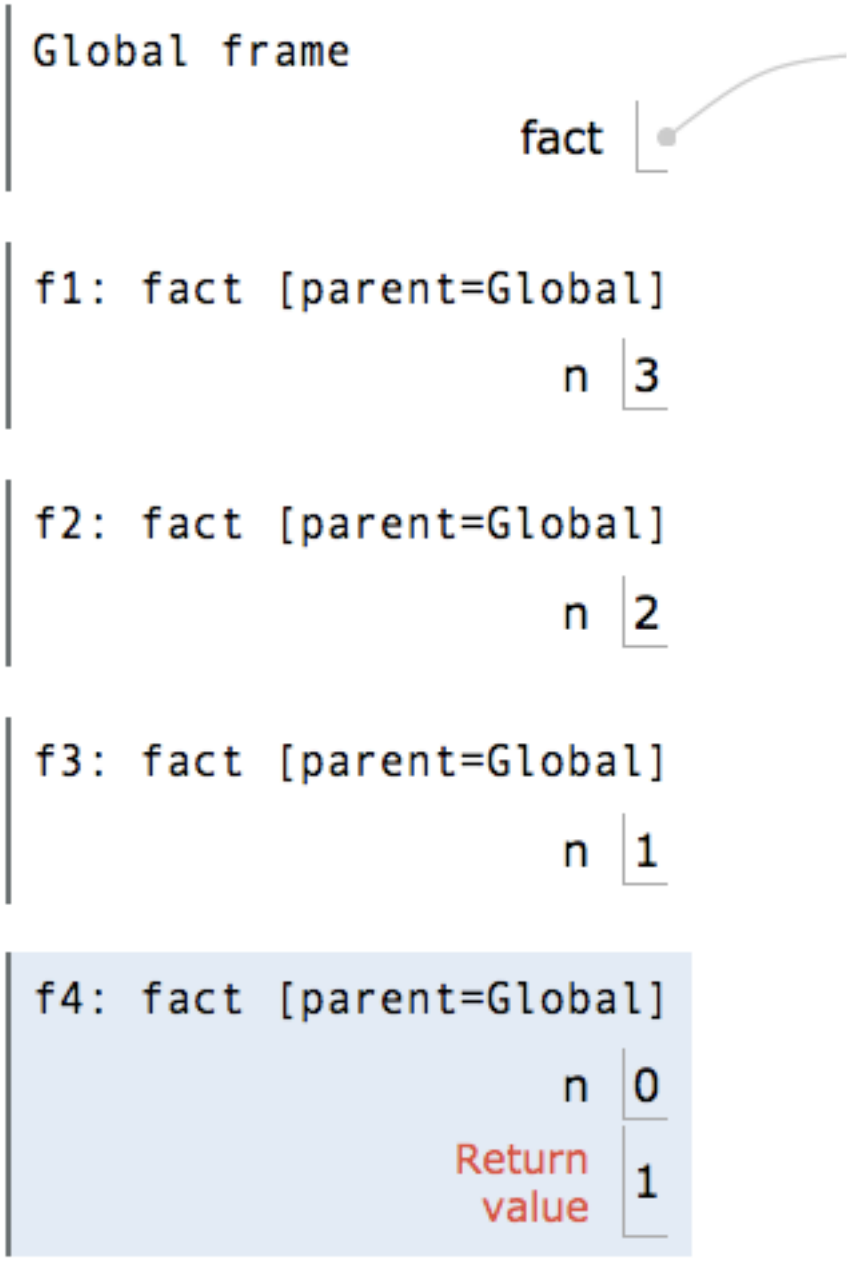


Recursion in Environment Diagrams (demo)



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

fact(3)
```

- The same function `fact` is called multiple times



Global frame

fact

f1: fact [parent=Global]

n 3

f2: fact [parent=Global]

n 2

f3: fact [parent=Global]

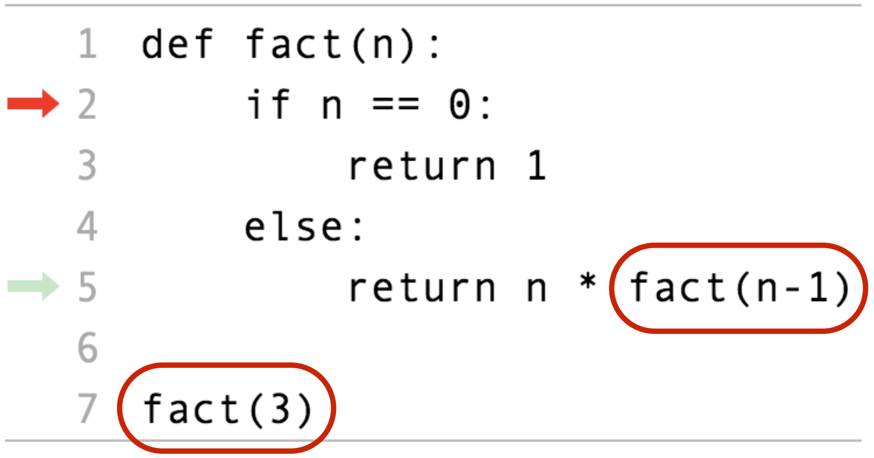
n 1

f4: fact [parent=Global]

n 0

Return value 1

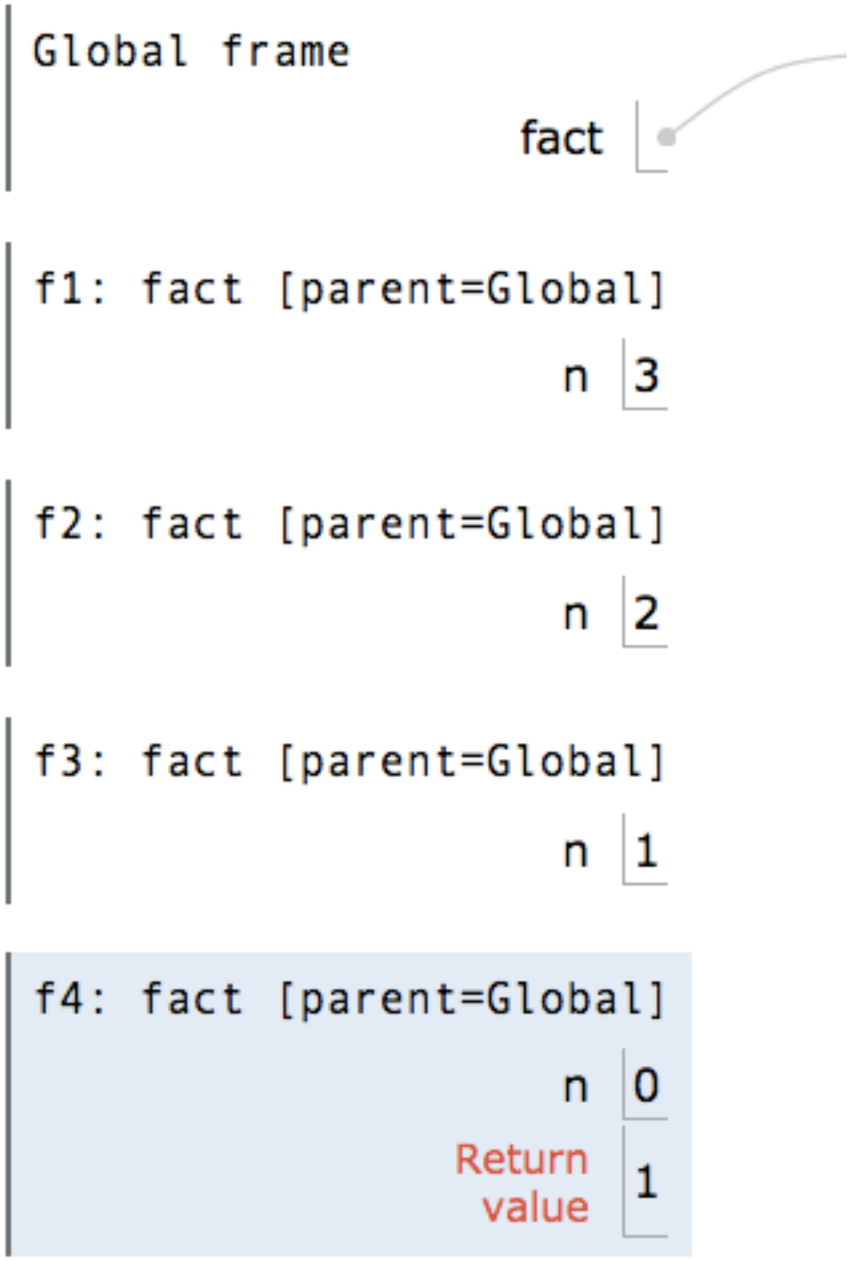
Recursion in Environment Diagrams (demo)



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

fact(3)
```

- The same function `fact` is called multiple times

- Different frames keep track of the different arguments in each call



Global frame

fact

f1: fact [parent=Global]

n 3

f2: fact [parent=Global]

n 2

f3: fact [parent=Global]

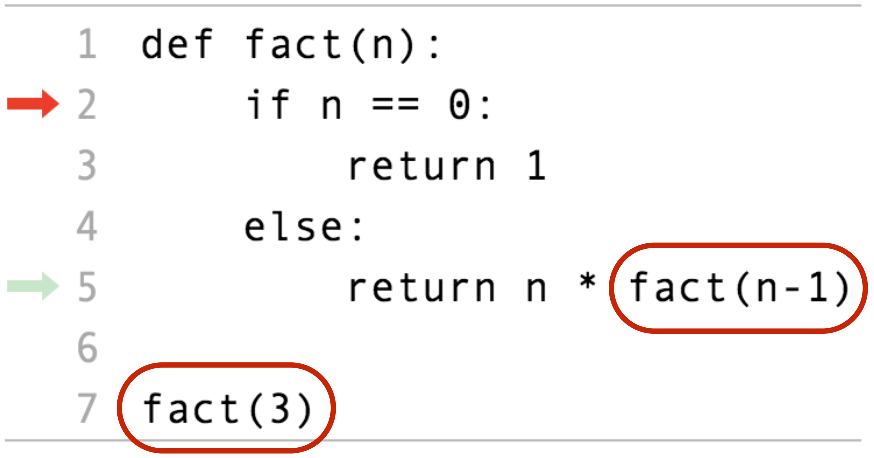
n 1

f4: fact [parent=Global]

n 0

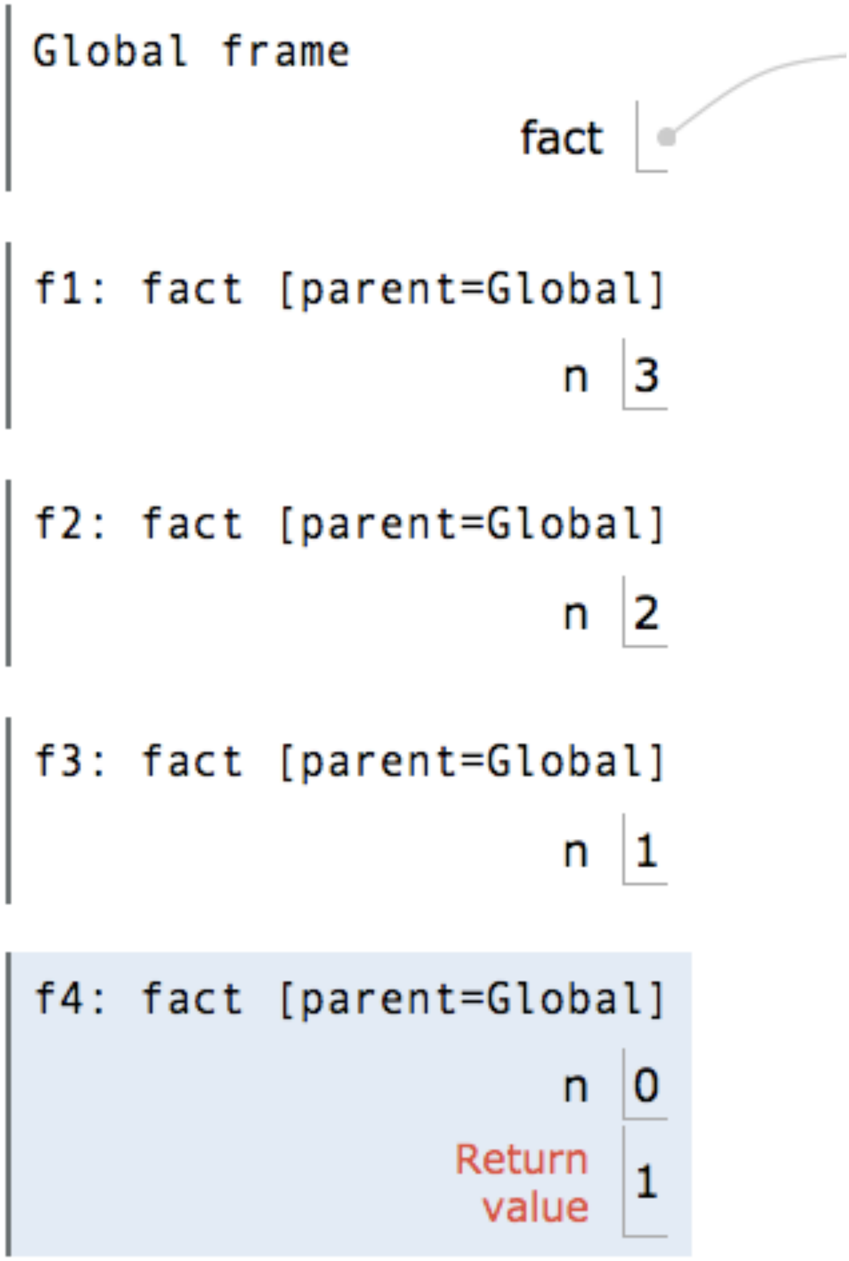
Return value 1

Recursion in Environment Diagrams (demo)



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

fact(3)
```

- The same function `fact` is called multiple times
- Different frames keep track of the different arguments in each call
- What `n` evaluates to depends upon the current environment



Global frame

fact

f1: fact [parent=Global]

n 3

f2: fact [parent=Global]

n 2

f3: fact [parent=Global]

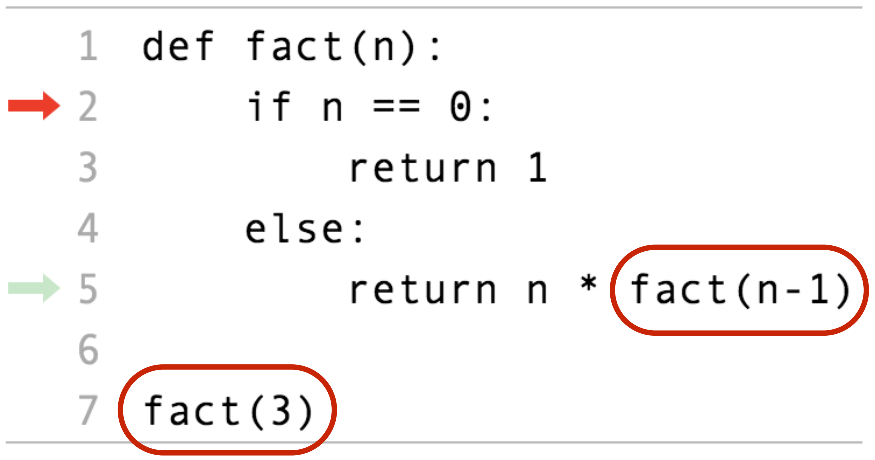
n 1

f4: fact [parent=Global]

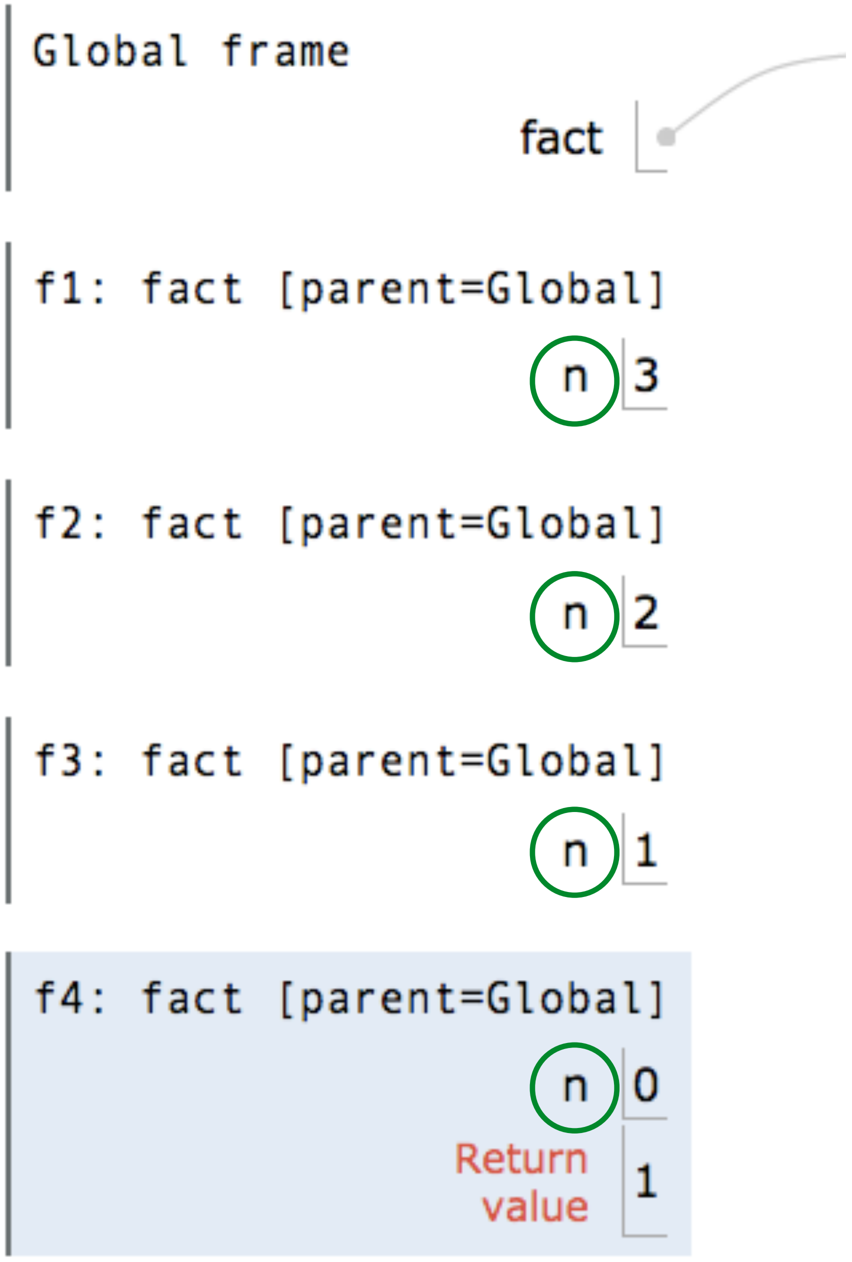
n 0

Return value 1

Recursion in Environment Diagrams (demo)

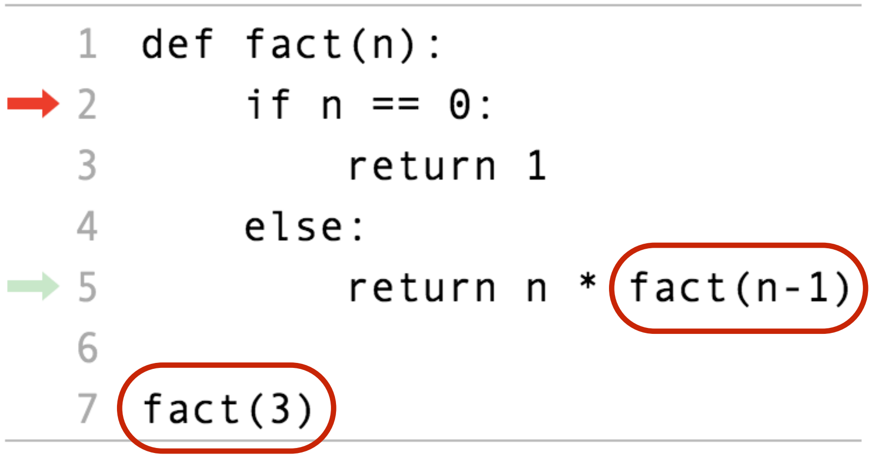


```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
```



- The same function `fact` is called multiple times
- Different frames keep track of the different arguments in each call
- What `n` evaluates to depends upon the current environment

Recursion in Environment Diagrams (demo)



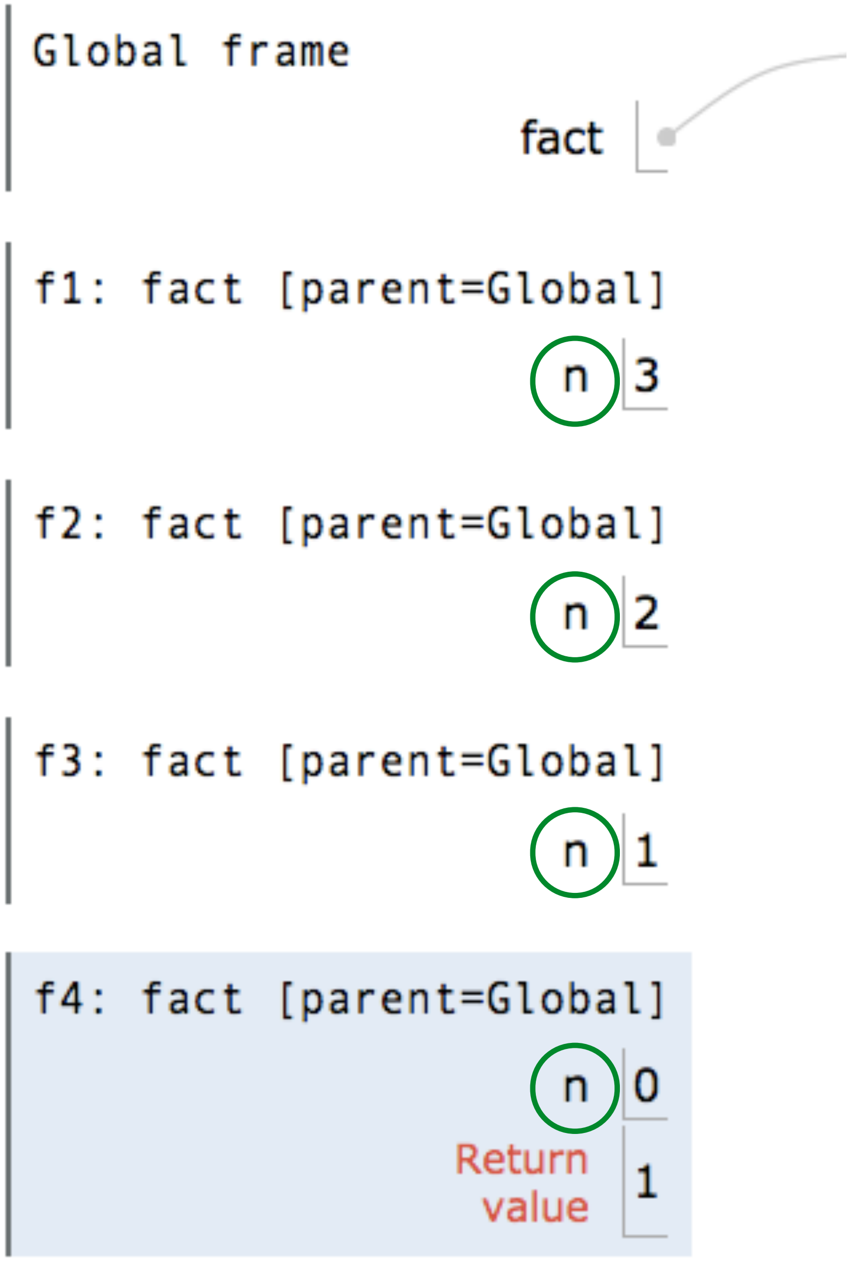
```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
```

- The same function `fact` is called multiple times

- Different frames keep track of the different arguments in each call

- What `n` evaluates to depends upon the current environment

- Each call to `fact` solves a simpler problem than the last: smaller `n`



Global frame

fact

f1: fact [parent=Global]

n 3

f2: fact [parent=Global]

n 2

f3: fact [parent=Global]

n 1

f4: fact [parent=Global]

n 0

Return value 1

Better: the Recursive Leap of Faith

```python
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Better: the Recursive Leap of Faith

```
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Is `factorial` **implemented correctly?**

Better: the Recursive Leap of Faith

```python
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Is `factorial` **implemented correctly?**

1. Verify the base case(s).

Better: the Recursive Leap of Faith

```
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Is `factorial` **implemented correctly?**

1. Verify the base case(s).
    1. Are they *correct*?

Better: the Recursive Leap of Faith

```python
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Is `factorial` implemented correctly?

1. Verify the base case(s).
    1. Are they *correct*?
    2. Are they *exhaustive*?

Better: the Recursive Leap of Faith

```
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Is `factorial` **implemented correctly?**

1. Verify the base case(s).
    1. Are they *correct*?
    2. Are they *exhaustive*?

Now, harness the power of *functional abstraction*!

Better: the Recursive Leap of Faith

```
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Is `factorial` implemented correctly?

1. Verify the base case(s).
   1. Are they *correct*?
   2. Are they *exhaustive*?

Now, harness the power of *functional abstraction*!

2. Assume that `factorial(n-1)` is correct.

Better: the Recursive Leap of Faith

```
def factorial(n):
    """Return the factorial of n."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Is `factorial` **implemented correctly?**

1. Verify the base case(s).
    1. Are they *correct*?
    2. Are they *exhaustive*?

Now, harness the power of *functional abstraction*!

2. Assume that `factorial(n-1)` is correct.
3. Verify that `factorial(n)` is correct.

Writing Recursion

Writing Recursion

```
def sum_digits(n):
    """Return the sum of the digits of n.

    >>> sum_digits(2016)
    9
    """
```

Writing Recursion

```
def sum_digits(n):
    """Return the sum of the digits of n.

    >>> sum_digits(2016)
    9
    """
```

```
if n < 0:
    return 0
```

```
if n == 1:
    return 1
```

```
if n < 10:
    return n
```

```
if n < 100:
    return n
```

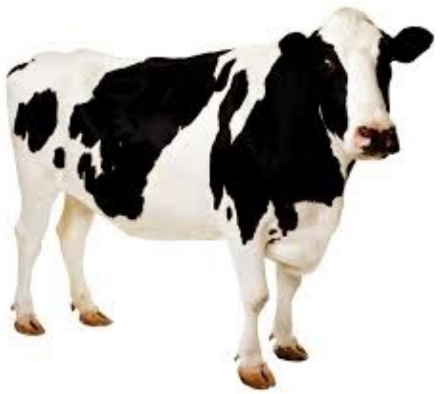
Writing Recursion

```
def sum_digits(n):
    """Return the sum of the digits of n.

    >>> sum_digits(2016)
    9
    """
```

```
if n < 0:
    return 0
```

```
if n == 1:
    return 1
```

```
if n < 10:
    return n
```



```
if n < 100:
    return n
```

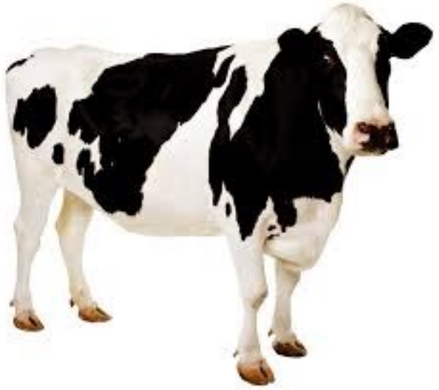
Writing Recursion

```
def sum_digits(n):
    """Return the sum of the digits of n.

    >>> sum_digits(2016)
    9
    """
```

```
if n < 0:
    return 0
```

```
if n == 1:
    return 1
```

```
if n < 10:
    return n
```



```
if n < 100:
    return n
```

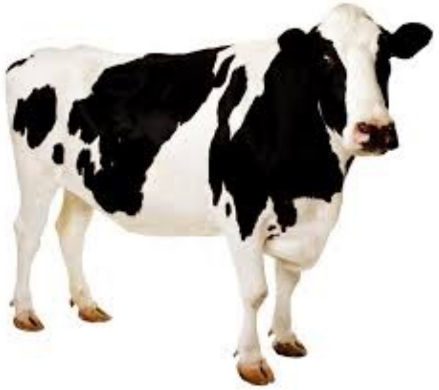
Writing Recursion

```
def sum_digits(n):
    """Return the sum of the digits of n.

    >>> sum_digits(2016)
    9
    """
```

```
if n < 0:
    return 0
```

```
if n == 1:
    return 1
```

```
if n < 10:
    return n
```



```
if n < 100:
    return n
```

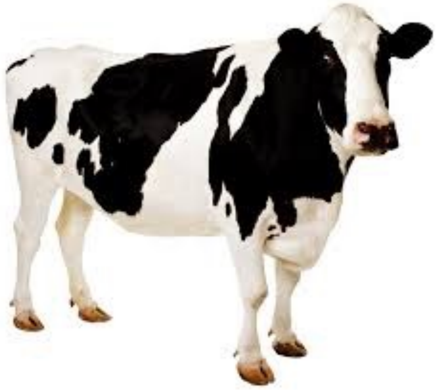
Writing Recursion

```
def sum_digits(n):
    """Return the sum of the digits of n.

    >>> sum_digits(2016)
    9
    """
```

```
if n < 0:
    return 0
```

```
if n == 1:
    return 1
```

```
if n < 10:
    return n
```

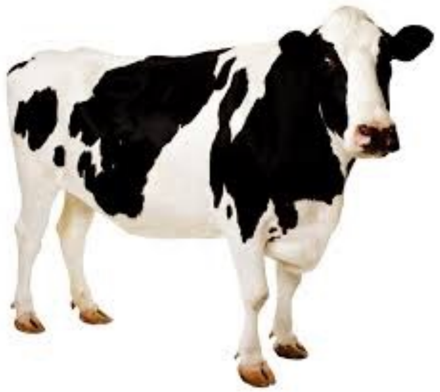


```
if n < 100:
    return n
```

Writing Recursion

```
def sum_digits(n):
    """Return the sum of the digits of n.

    >>> sum_digits(2016)
    9
    """
```

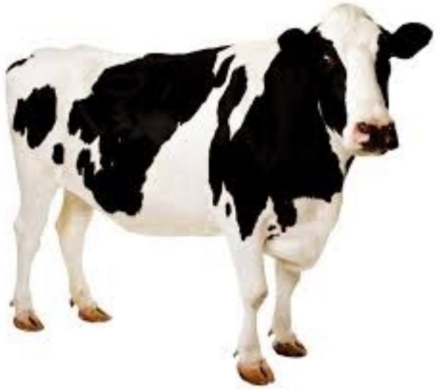
```
if n < 10:
    return n
```



Writing Recursion (demo)

```
def sum_digits(n):
    """Return the sum of the digits of n.

    >>> sum_digits(2016)
    9
    """
```

```
if n < 10:
    return n
```



Writing Recursion (demo)

```
def sum_digits(n):
    """Return the sum of the digits of n.

    >>> sum_digits(2016)
    9
    """
```

Writing Recursion (demo)

```python
def sum_digits(n):
    """Return the sum of the digits of n.

    >>> sum_digits(2016)
    9
    """
    if n < 10:
        return n
    else:
        return sum_digits(n//10) + n%10
```

Iteration vs Recursion

Iteration vs Recursion

- Iteration is a special case of recursion

Iteration vs Recursion

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Iteration vs Recursion (demo)

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Iteration vs Recursion (demo)

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Using iteration:

Iteration vs Recursion (demo)

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Using iteration:

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Iteration vs Recursion (demo)

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Using iteration:

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

Iteration vs Recursion (demo)

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Using iteration:

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

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

Iteration vs Recursion (demo)

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Using iteration:

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

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

Math:

Iteration vs Recursion (demo)

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Using iteration:

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Math: $n! = \prod_{k=1}^{n} k$

Iteration vs Recursion (demo)

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Using iteration:

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

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

Math: $n! = \prod_{k=1}^{n} k$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Iteration vs Recursion (demo)

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Using iteration:

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

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

Math: $$n! = \prod_{k=1}^{n} k$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Names:

Iteration vs Recursion (demo)

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Using iteration:

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Using recursion:

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

Math: $$n! = \prod_{k=1}^{n} k$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Names: `n, total, k, fact_iter`

Iteration vs Recursion (demo)

- Iteration is a special case of recursion
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky

Using iteration:

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total*k, k+1
    return total
```

Math: $$n! = \prod_{k=1}^{n} k$$

Names: n, total, k, fact_iter

Using recursion:

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

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

n, fact

Recursion on Sequences

Recursion on Sequences

- We’ve seen iteration as one way of working with sequences, but iteration is a special case of recursion

Recursion on Sequences

- We’ve seen iteration as one way of working with sequences, but iteration is a special case of recursion
- This means that we can also use recursion to solve problems involving sequences!

Recursion on Sequences (demo)

- We’ve seen iteration as one way of working with sequences, but iteration is a special case of recursion
- This means that we can also use recursion to solve problems involving sequences!

Recursion on Sequences (demo)

- We’ve seen iteration as one way of working with sequences, but iteration is a special case of recursion
- This means that we can also use recursion to solve problems involving sequences!

```python
def reverse(word):
    """Return the reverse of the string word."""
    if len(word) < 2:
        return word
    else:
        return reverse(word[1:]) + word[0]
```

Summary

Summary

- *Recursive functions* call themselves, either directly or indirectly, in the function body

Summary

- *Recursive functions* call themselves, either directly or indirectly, in the function body
  - The motivation for this is to *break down* the problem into smaller, easier to solve problems

Summary

- *Recursive functions* call themselves, either directly or indirectly, in the function body
  - The motivation for this is to *break down* the problem into smaller, easier to solve problems
  - For example, computing the factorial of a smaller number, or the reverse of a shorter string

Summary

- *Recursive functions* call themselves, either directly or indirectly, in the function body
  - The motivation for this is to *break down* the problem into smaller, easier to solve problems
  - For example, computing the factorial of a smaller number, or the reverse of a shorter string

- Recursive functions have *base cases*, which are not recursive, and *recursive cases*

Summary

- *Recursive functions* call themselves, either directly or indirectly, in the function body
  - The motivation for this is to *break down* the problem into smaller, easier to solve problems
  - For example, computing the factorial of a smaller number, or the reverse of a shorter string

- Recursive functions have *base cases*, which are not recursive, and *recursive cases*
  - The best way to verify recursive functions is with functional abstraction!

Summary

- *Recursive functions* call themselves, either directly or indirectly, in the function body
  - The motivation for this is to *break down* the problem into smaller, easier to solve problems
  - For example, computing the factorial of a smaller number, or the reverse of a shorter string

- Recursive functions have *base cases*, which are not recursive, and *recursive cases*
  - The best way to verify recursive functions is with functional abstraction!
  - Use the *leap of faith*