61A Lecture 6

Monday, February 2

Announcements

Announcements

- Homework 2 due Monday 2/2 @ 11:59pm

Announcements

- Homework 2 due Monday 2/2 @ 11:59pm
- Project 1 due Thursday 2/5 @ 11:59pm

Announcements

- Homework 2 due Monday 2/2 @ 11:59pm
- Project 1 due Thursday 2/5 @ 11:59pm
  - Project party on Tuesday 2/3 5pm–6:30pm in 2050 VLSB

Announcements

- Homework 2 due Monday 2/2 @ 11:59pm
- Project 1 due Thursday 2/5 @ 11:59pm
  - Project party on Tuesday 2/3 5pm–6:30pm in 2050 VLSB
  - Partner party on Wednesday 2/4 3pm–4pm in Wozniak Lounge, Soda Hall

Announcements

- Homework 2 due Monday 2/2 @ 11:59pm
- Project 1 due Thursday 2/5 @ 11:59pm
  - Project party on Tuesday 2/3 5pm-6:30pm in 2050 VLSB
  - Partner party on Wednesday 2/4 3pm-4pm in Wozniak Lounge, Soda Hall
  - Earn 1 bonus point if you finish by Wednesday 2/4 @ 11:59pm

Announcements

- Homework 2 due Monday 2/2 @ 11:59pm
- Project 1 due Thursday 2/5 @ 11:59pm
  - Project party on Tuesday 2/3 5pm–6:30pm in 2050 VLSB
  - Partner party on Wednesday 2/4 3pm–4pm in Wozniak Lounge, Soda Hall
  - Earn 1 bonus point if you finish by Wednesday 2/4 @ 11:59pm
  - Composition: Programs should be concise, well-named, understandable, and easy to follow

Announcements

- Homework 2 due Monday 2/2 @ 11:59pm
- Project 1 due Thursday 2/5 @ 11:59pm
  - Project party on Tuesday 2/3 5pm–6:30pm in 2050 VLSB
  - Partner party on Wednesday 2/4 3pm–4pm in Wozniak Lounge, Soda Hall
  - Earn 1 bonus point if you finish by Wednesday 2/4 @ 11:59pm
  - Composition: Programs should be concise, well-named, understandable, and easy to follow
- Extra lecture 2 on Thursday 2/5 5pm–6:30pm in 2050 VLSB

Announcements

- Homework 2 due Monday 2/2 @ 11:59pm
- Project 1 due Thursday 2/5 @ 11:59pm
  - Project party on Tuesday 2/3 5pm-6:30pm in 2050 VLSB
  - Partner party on Wednesday 2/4 3pm-4pm in Wozniak Lounge, Soda Hall
  - Earn 1 bonus point if you finish by Wednesday 2/4 @ 11:59pm
  - Composition: Programs should be concise, well-named, understandable, and easy to follow
- Extra lecture 2 on Thursday 2/5 5pm-6:30pm in 2050 VLSB
  - Hog strategies & church numerals

Announcements

- Homework 2 due Monday 2/2 @ 11:59pm
- Project 1 due Thursday 2/5 @ 11:59pm
  - Project party on Tuesday 2/3 5pm–6:30pm in 2050 VLSB
  - Partner party on Wednesday 2/4 3pm–4pm in Wozniak Lounge, Soda Hall
  - Earn 1 bonus point if you finish by Wednesday 2/4 @ 11:59pm
  - Composition: Programs should be concise, well-named, understandable, and easy to follow
- Extra lecture 2 on Thursday 2/5 5pm–6:30pm in 2050 VLSB
  - Hog strategies & church numerals
- Midterm 1 on Monday 2/9 7pm–9pm

Announcements

- Homework 2 due Monday 2/2 @ 11:59pm
- Project 1 due Thursday 2/5 @ 11:59pm
  - Project party on Tuesday 2/3 5pm-6:30pm in 2050 VLSB
  - Partner party on Wednesday 2/4 3pm-4pm in Wozniak Lounge, Soda Hall
  - Earn 1 bonus point if you finish by Wednesday 2/4 @ 11:59pm
  - Composition: Programs should be concise, well-named, understandable, and easy to follow
- Extra lecture 2 on Thursday 2/5 5pm-6:30pm in 2050 VLSB
  - Hog strategies & church numerals
- Midterm 1 on Monday 2/9 7pm-9pm
  - Conflict? Fill out the conflict form today! http://goo.gl/2P5fKq

Recursive Functions

Recursive Functions

Recursive Functions

Definition: A function is called recursive if the body of that function calls itself, either directly or indirectly.

Recursive Functions

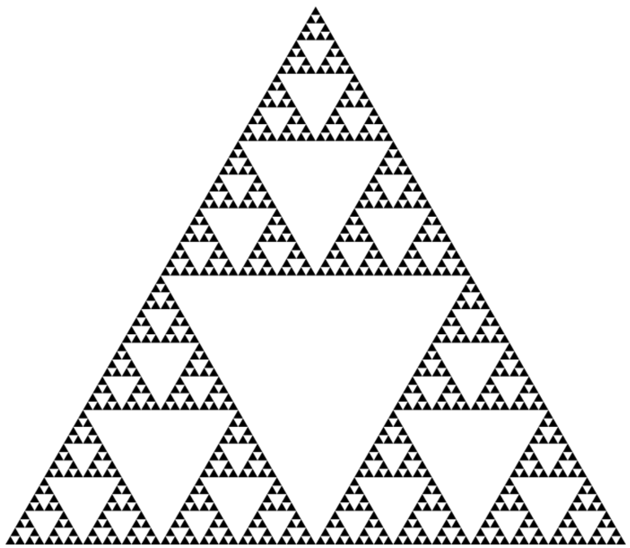
Definition: A function is called recursive if the body of that function calls itself, either directly or indirectly.

Implication: Executing the body of a recursive function may require applying that function.

Recursive Functions

Definition: A function is called recursive if the body of that function calls itself, either directly or indirectly.

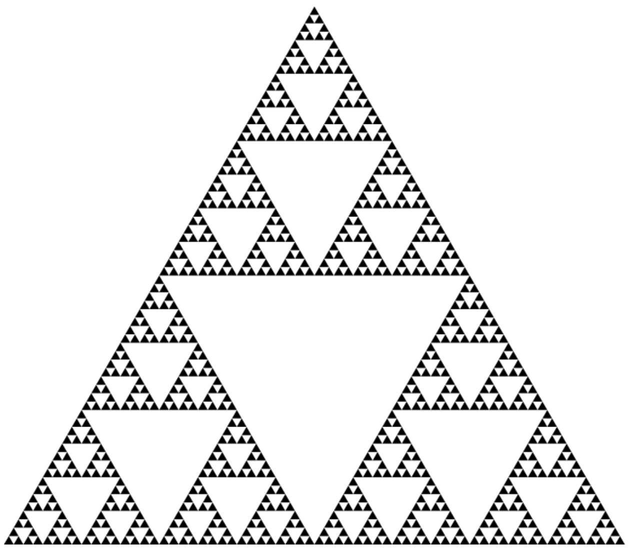
Implication: Executing the body of a recursive function may require applying that function.

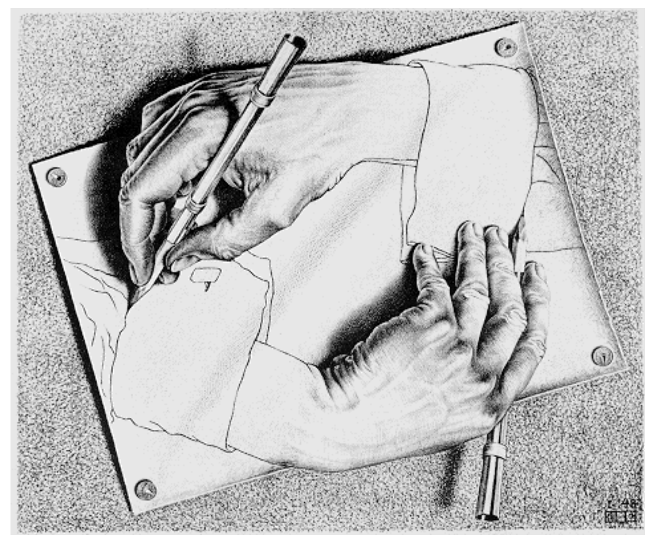


Recursive Functions

Definition: A function is called recursive if the body of that function calls itself, either directly or indirectly.

Implication: Executing the body of a recursive function may require applying that function.





Drawing Hands, by M. C. Escher (lithograph, 1948)

Digit Sums

`2+0+1+5 = 8`

Digit Sums

2+0+1+5 = 8

- If a number a is divisible by 9, then `sum_digits(a)` is also divisible by 9.

Digit Sums

2+0+1+5 = 8

- If a number a is divisible by 9, then `sum_digits(a)` is also divisible by 9.
- Useful for typo detection!

Digit Sums

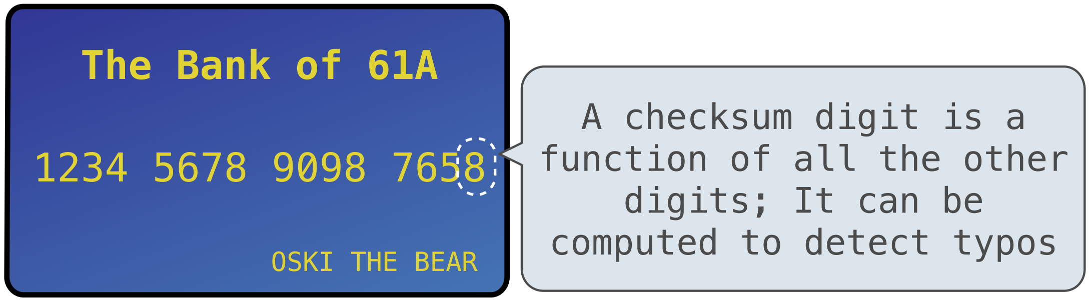
2+0+1+5 = 8

- If a number a is divisible by 9, then `sum_digits(a)` is also divisible by 9.
- Useful for typo detection!

Digit Sums

2+0+1+5 = 8

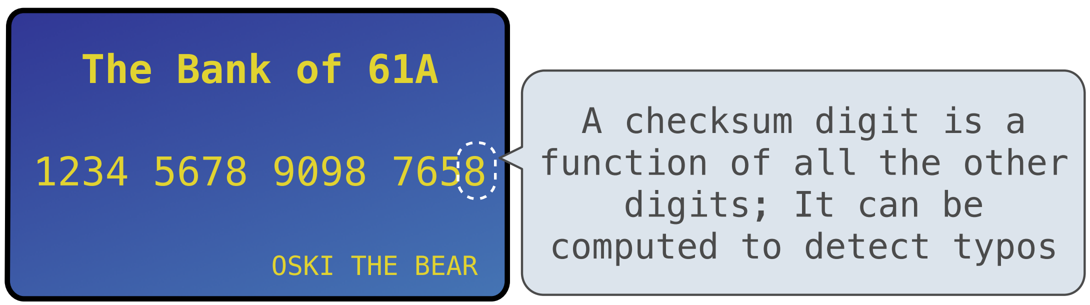
- If a number a is divisible by 9, then `sum_digits(a)` is also divisible by 9.
- Useful for typo detection!



Digit Sums

2+0+1+5 = 8

- If a number a is divisible by 9, then sum_digits(a) is also divisible by 9.
- Useful for typo detection!



- Credit cards actually use the Luhn algorithm, which we'll implement after digit_sum.

Sum Digits Without a While Statement

Sum Digits Without a While Statement

```
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10
```

Sum Digits Without a While Statement

```
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10

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

Sum Digits Without a While Statement

```
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10


def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
```

Sum Digits Without a While Statement

```
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10

def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
```

Sum Digits Without a While Statement

```
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10

def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

The Anatomy of a Recursive Function

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

The Anatomy of a Recursive Function

- The def statement header is similar to other functions

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

The Anatomy of a Recursive Function

- The def statement header is similar to other functions

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

The Anatomy of a Recursive Function

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

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

The Anatomy of a Recursive Function

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

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

The 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

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

The 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

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

The 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 sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

The 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 sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

The 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 sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

(Demo)

Recursion in Environment Diagrams

Recursion in Environment Diagrams

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

fact(3)
```

Interactive Diagram

Recursion in Environment Diagrams

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

fact(3)
```

(Demo)

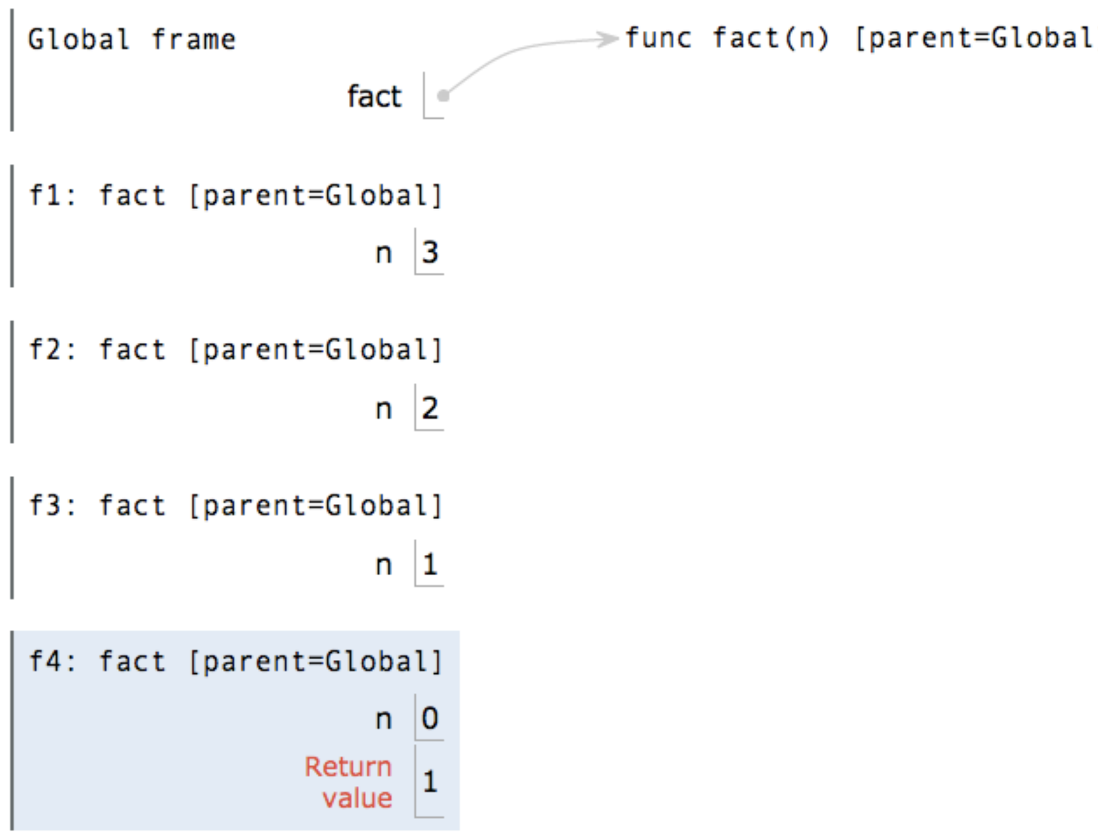
Interactive Diagram

Recursion in Environment Diagrams

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

fact(3)
```

(Demo)



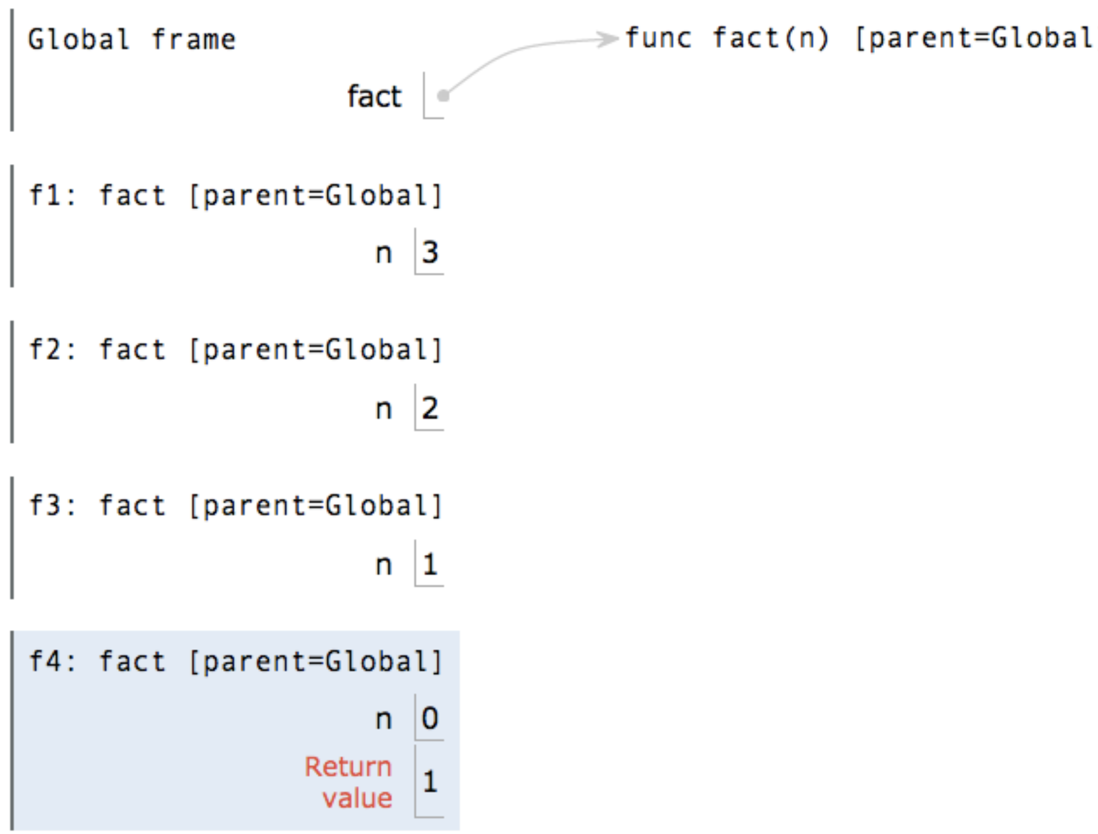
Interactive Diagram

Recursion in Environment Diagrams

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

fact(3)
```

(Demo)



- The same function fact is called multiple times.

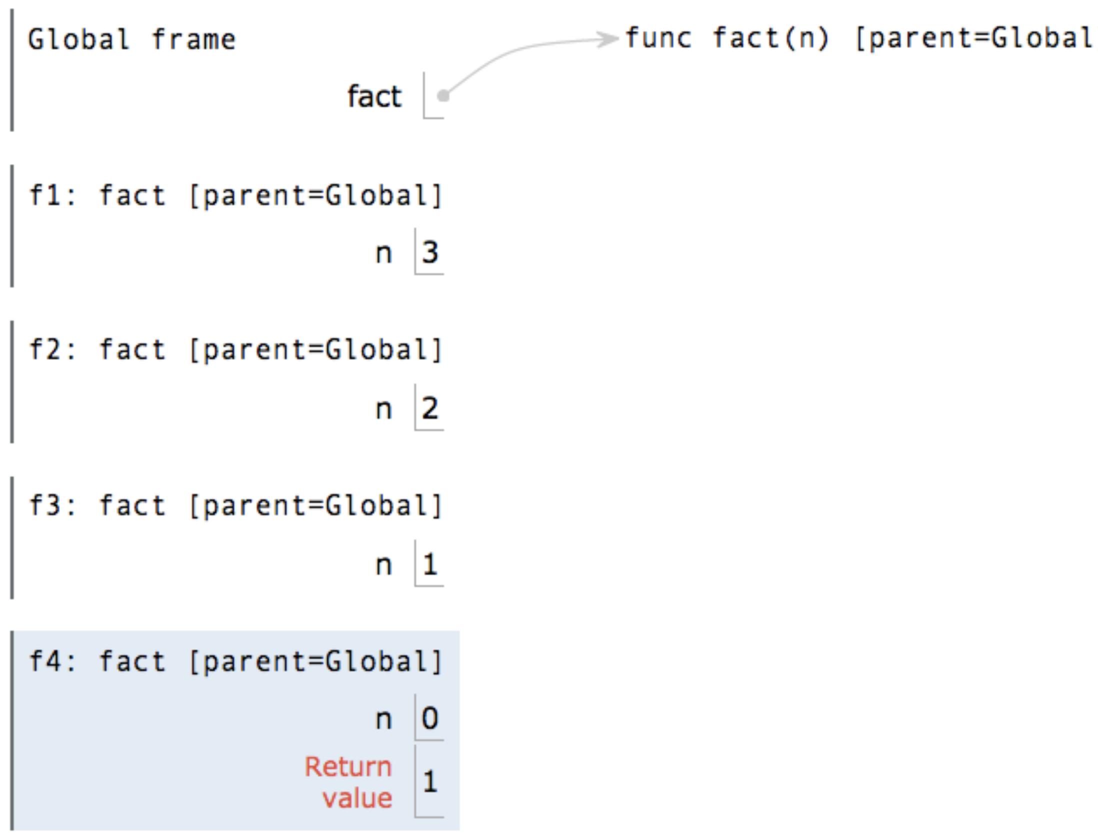
Interactive Diagram

Recursion in Environment Diagrams

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

fact(3)
```

(Demo)



- The same function fact is called multiple times.

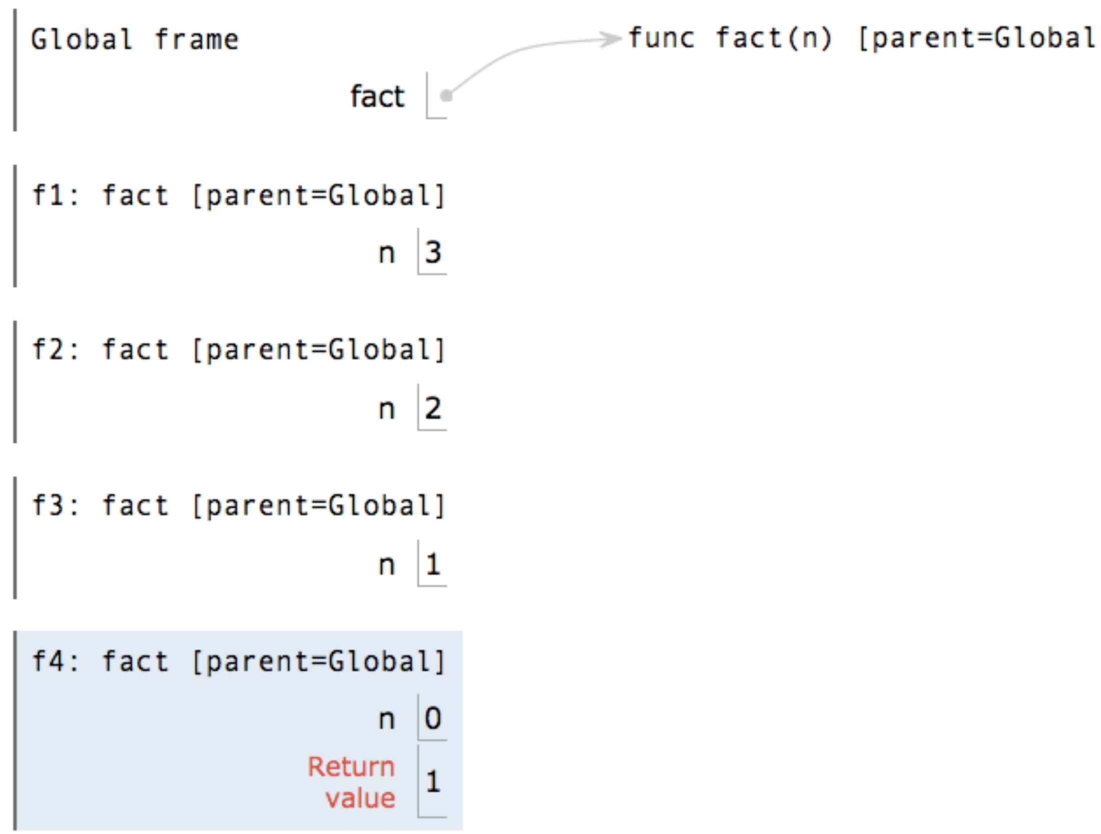
Interactive Diagram

Recursion in Environment Diagrams

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

fact(3)
```

(Demo)



- The same function fact is called multiple times.
- Different frames keep track of the different arguments in each call.

Interactive Diagram

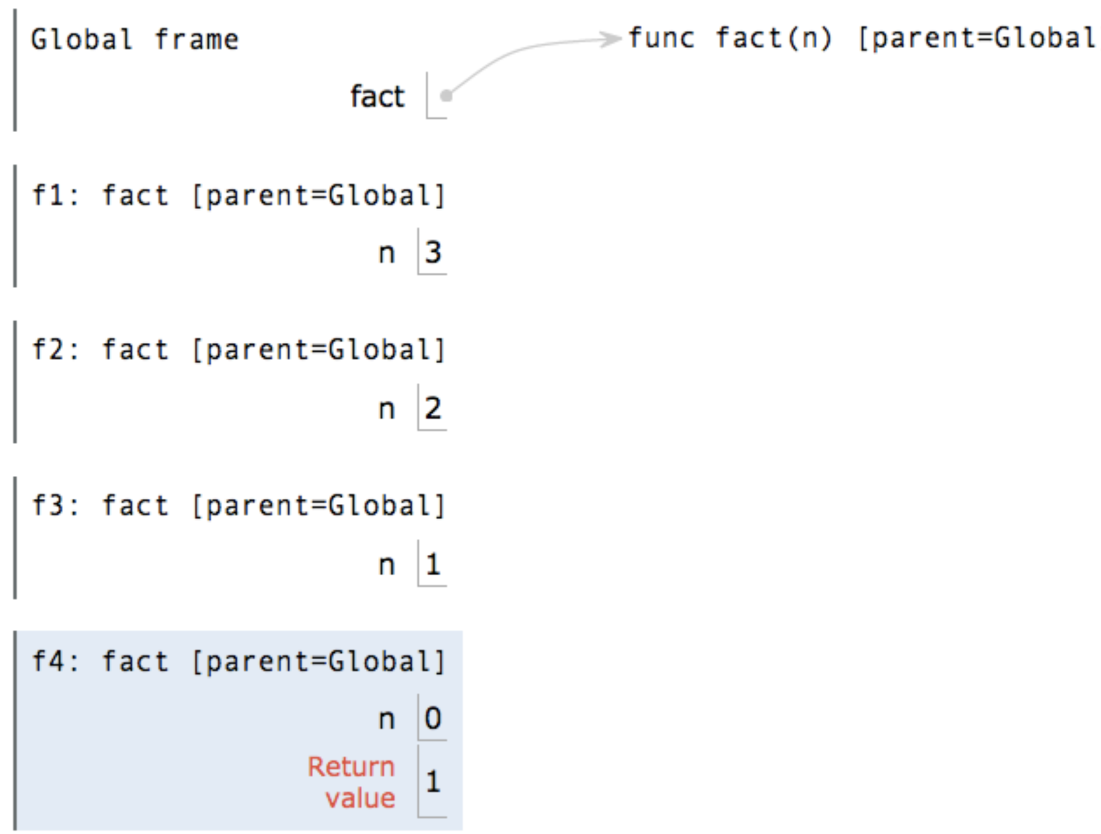
Recursion in Environment Diagrams

```
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 which is the current environment.

(Demo)



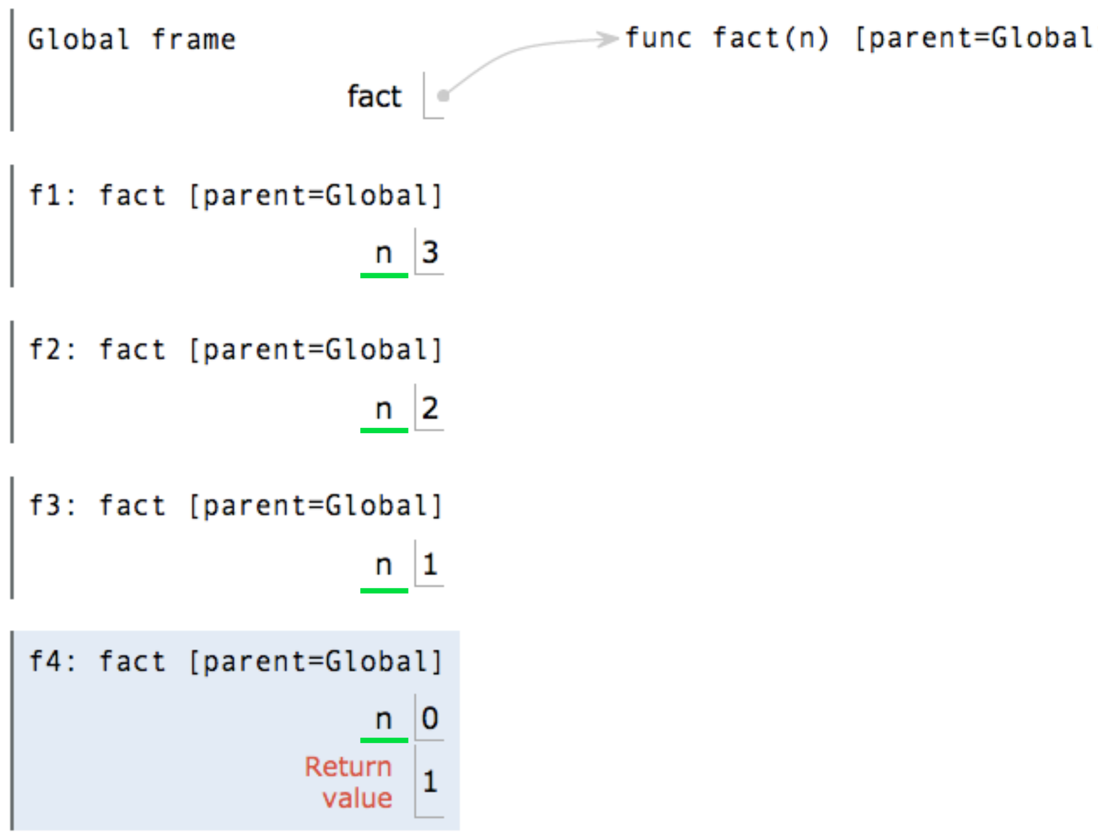
Interactive Diagram

Recursion in Environment Diagrams

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

fact(3)
```

(Demo)



- 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 which is the current environment.

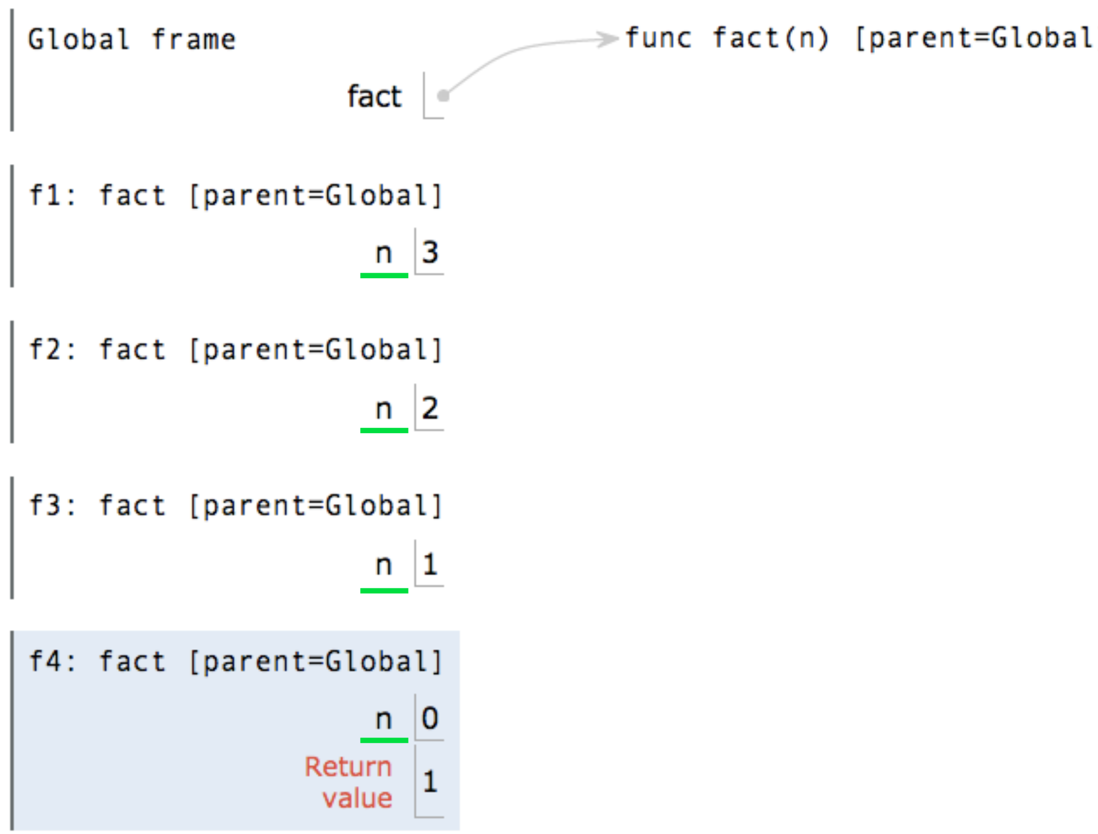
Interactive Diagram

Recursion in Environment Diagrams

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

fact(3)
```

(Demo)



- 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 which is the current environment.
- Each call to fact solves a simpler problem than the last: smaller n.

Interactive Diagram

Iteration vs Recursion

Iteration vs Recursion

Iteration is a special case of recursion

Iteration vs Recursion

`Iteration is a special case of recursion`

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Iteration vs Recursion

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

Iteration vs Recursion

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

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

Iteration vs Recursion

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

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

Using recursion:

Iteration vs Recursion

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

```
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

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

```
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

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

```
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:

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

Iteration vs Recursion

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

```
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

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

```
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

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

```
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

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

```
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

n, fact

Verifying Recursive Functions

The Recursive Leap of Faith

The Recursive Leap of Faith

Photo by Kevin Lee, Preikestolen, Norway

The Recursive Leap of Faith

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

Photo by Kevin Lee, Preikestolen, Norway

The Recursive Leap of Faith

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

Is fact implemented correctly?

Photo by Kevin Lee, Preikestolen, Norway

The Recursive Leap of Faith

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

Is fact implemented correctly?

1. Verify the base case.

Photo by Kevin Lee, Preikestolen, Norway

The Recursive Leap of Faith

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

Is fact implemented correctly?

1. Verify the base case.

2. Treat **fact** as a functional abstraction!

Photo by Kevin Lee, Preikestolen, Norway

The Recursive Leap of Faith

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

Is fact implemented correctly?

1. Verify the base case.
2. Treat **fact** as a functional abstraction!
3. Assume that fact(n-1) is correct.

Photo by Kevin Lee, Preikestolen, Norway

The Recursive Leap of Faith

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

Is fact implemented correctly?

1. Verify the base case.
2. Treat **fact** as a functional abstraction!
3. Assume that fact(n-1) is correct.
4. Verify that fact(n) is correct, assuming that fact(n-1) correct.

Photo by Kevin Lee, Preikestolen, Norway

Verifying Digit Sum

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Verifying Digit Sum

The sum_digits function computes the sum of positive n correctly because:

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Verifying Digit Sum

The sum_digits function computes the sum of positive n correctly because:

The sum of the digits of . **(base case)**

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Verifying Digit Sum

The sum_digits function computes the sum of positive n correctly because:

The sum of the digits of . **(base case)**

Assuming **(abstraction)**

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Verifying Digit Sum

The sum_digits function computes the sum of positive n correctly because:

The sum of the digits of . **(base case)**

Assuming **(abstraction)**

for all , **(simpler case)**

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Verifying Digit Sum

The sum_digits function computes the sum of positive n correctly because:

The sum of the digits of . **(base case)**

Assuming **(abstraction)**

for all , **(simpler case)**

sum_digits(n) will be . **(conclusion)**

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Verifying Digit Sum

The sum_digits function computes the sum of positive n correctly because:

The sum of the digits of any n < 10 is n. **(base case)**

Assuming **(abstraction)**

for all , **(simpler case)**

sum_digits(n) will be . **(conclusion)**

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Verifying Digit Sum

The sum_digits function computes the sum of positive n correctly because:

The sum of the digits of any n < 10 is n. **(base case)**

Assuming sum_digits(k) correctly sums the digits of k **(abstraction)**

for all , **(simpler case)**

sum_digits(n) will be . **(conclusion)**

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Verifying Digit Sum

The sum_digits function computes the sum of positive n correctly because:

The sum of the digits of any n < 10 is n. **(base case)**

Assuming sum_digits(k) correctly sums the digits of k **(abstraction)**

for all k with fewer digits than n, **(simpler case)**

sum_digits(n) will be . **(conclusion)**

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Verifying Digit Sum

The sum_digits function computes the sum of positive n correctly because:

The sum of the digits of any n < 10 is n. **(base case)**

Assuming sum_digits(k) correctly sums the digits of k **(abstraction)**

for all k with fewer digits than n, **(simpler case)**

sum_digits(n) will be sum_digits(n//10) plus the last digit of n. **(conclusion)**

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Mutual Recursion

The Luhn Algorithm

The Luhn Algorithm

`Used to verify credit card numbers`

The Luhn Algorithm

Used to verify credit card numbers

From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm

The Luhn Algorithm

Used to verify credit card numbers

From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm

- From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).

The Luhn Algorithm

Used to verify credit card numbers

From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm

- From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).

- Take the sum of all the digits.

The Luhn Algorithm

Used to verify credit card numbers

From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm

- From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).

- Take the sum of all the digits.

1	3	8	7	4	3

The Luhn Algorithm

Used to verify credit card numbers

From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm

- From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).

- Take the sum of all the digits.

1	3	8	7	4	3
2	3	1+6=7	7	8	3

The Luhn Algorithm

Used to verify credit card numbers

From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm

- From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).

- Take the sum of all the digits.

1	3	8	7	4	3	
2	3	1+6=7	7	8	3	= 30

The Luhn Algorithm

Used to verify credit card numbers

From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm

- From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).
- Take the sum of all the digits.

1	3	8	7	4	3	
2	3	1+6=7	7	8	3	= 30

The Luhn sum of a valid credit card number is a multiple of 10.

The Luhn Algorithm

Used to verify credit card numbers

From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm

- From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5).
- Take the sum of all the digits.

1	3	8	7	4	3	
2	3	1+6=7	7	8	3	= 30

The Luhn sum of a valid credit card number is a multiple of 10. (Demo)

Recursion and Iteration

Converting Recursion to Iteration

Converting Recursion to Iteration

```
Can be tricky: Iteration is a special case of recursion.
```

Converting Recursion to Iteration

Can be tricky: Iteration is a special case of recursion.

Idea: Figure out what state must be maintained by the iterative function.

Converting Recursion to Iteration

Can be tricky: Iteration is a special case of recursion.

Idea: Figure out what state must be maintained by the iterative function.

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Converting Recursion to Iteration

Can be tricky: Iteration is a special case of recursion.

Idea: Figure out what state must be maintained by the iterative function.

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

What's left to sum

Converting Recursion to Iteration

Can be tricky: Iteration is a special case of recursion.

Idea: Figure out what state must be maintained by the iterative function.

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

Converting Recursion to Iteration

Can be tricky: Iteration is a special case of recursion.

Idea: Figure out what state must be maintained by the iterative function.

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

A partial sum

What's left to sum

(Demo)

Converting Iteration to Recursion

Converting Iteration to Recursion

`More formulaic: Iteration is a special case of recursion.`

Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.

Idea: The state of an iteration can be passed as arguments.

Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.

Idea: The state of an iteration can be passed as arguments.

```
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum
```

Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.

Idea: The state of an iteration can be passed as arguments.

```
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum
```

```
def sum_digits_rec(n, digit_sum):
    if n == 0:
        return digit_sum
    else:
        n, last = split(n)
        return sum_digits_rec(n, digit_sum + last)
```

Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.

Idea: The state of an iteration can be passed as arguments.

```
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum
```

Updates via assignment become...

```
def sum_digits_rec(n, digit_sum):
    if n == 0:
        return digit_sum
    else:
        n, last = split(n)
        return sum_digits_rec(n, digit_sum + last)
```

Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion.

Idea: The state of an iteration can be passed as arguments.

```
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum
```

Updates via assignment become...

```
def sum_digits_rec(n, digit_sum):
    if n == 0:
        return digit_sum
    else:
        n, last = split(n)
        return sum_digits_rec(n, digit_sum + last)
```

...arguments to a recursive call