Announcements
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.
Digit Sums

\[ 2 + 0 + 1 + 5 = 8 \]
Digit Sums

\[2+0+1+5 = 8\]

• If a number \(a\) is divisible by 9, then \(\text{sum_digits}(a)\) is also divisible by 9
Digit Sums

If a number \( a \) is divisible by 9, then \( \text{sum_digits}(a) \) is also divisible by 9.

- Useful for typo detection!

\[
2 + 0 + 1 + 5 = 8
\]
Digit Sums

If a number $a$ is divisible by 9, then $\text{sum_digits}(a)$ is also divisible by 9.

Useful for typo detection!

\[2+0+1+5 = 8\]
Digit Sums

If a number $a$ is divisible by 9, then $\text{sum_digits}(a)$ is also divisible by 9

Useful for typo detection!

2 + 0 + 1 + 5 = 8

A checksum digit is a function of all the other digits; it can be computed to detect typos.
Digit Sums

2 + 0 + 1 + 5 = 8

• If a number \( a \) is divisible by 9, then \( \text{sum_digits}(a) \) is also divisible by 9
• Useful for typo detection!

Credit cards actually use the Luhn algorithm, which we'll implement after \( \text{digit_sum} \)
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 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."""
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
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
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

```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
```
The Anatomy of a Recursive Function

- The def statement header is similar to other functions

```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
```
The Anatomy of a Recursive Function

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

```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
```
The Anatomy of a Recursive Function

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

```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
```
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

```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
```
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

```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
```
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

```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
```
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.

```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
```
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

```python
1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6
7 fact(3)
```
Recursion in Environment Diagrams

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

fact(3)
```

(Demo)
Recursion in Environment Diagrams

```
1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6
7 fact(3)
```

(Demo)

```
Global frame

func fact(n) [parent=Global]

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
```

Interactive Diagram
Recursion in Environment Diagrams

```python
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(3)
```

- The same function `fact` is called multiple times

(Demo)

- Interactive Diagram
Recursion in Environment Diagrams

• The same function `fact` is called multiple times

```python
1  def fact(n):
  2      if n == 0:
  3          return 1
  4      else:
  5          return n * fact(n-1)
  6
7  fact(3)
```

(Demo)

Global frame

```
func fact(n) [parent=Global]
```

```
f1: fact [parent=Global]
    fact
    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

The same function `fact` is called multiple times.

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

```python
1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6
7 fact(3)
```

(Demo)

```
Global frame

func fact(n) [parent=Global]

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

```python
1. def fact(n):
2.     if n == 0:
3.         return 1
4.     else:
5.         return n * fact(n-1)
6.
7. 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.

(Demo)

<table>
<thead>
<tr>
<th>Global frame</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fact</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f1: fact [parent=Global]</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code> 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f2: fact [parent=Global]</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code> 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f3: fact [parent=Global]</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code> 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f4: fact [parent=Global]</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code> 0 [Return value] 1</td>
</tr>
</tbody>
</table>
Recursion in Environment Diagrams

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.

```
1  def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n-1)
6
7  fact(3)
```

(Demo)

```
Global frame
  func fact(n) [parent=Global]

  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

The same function \texttt{fact} is called multiple times.

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

What \texttt{n} evaluates to depends upon the current environment.

Each call to \texttt{fact} solves a simpler problem than the last: smaller \texttt{n}.


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

fact(3)
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:

```python
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:

```python
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:

```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)
```
Iteration vs Recursion

Iteration is a special case of recursion

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

Using while:

```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:
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using while:

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

Math:

\[ n! = \prod_{k=1}^{n} k \]

Using recursion:

```python
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:

```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:

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

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

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using while:

```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:

\[ n! = \prod_{k=1}^{n} k \]

Names:
Iteration vs Recursion

Iteration is a special case of recursion

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

Using while:

```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:

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

Names: $n, \text{total}, k, \text{fact_iter}$
Iteration vs Recursion

Iteration is a special case of recursion

\[ 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \]

Using while:

```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:

\[ n! = \prod_{k=1}^{n} k \]

Using while:

\[ n! = \prod_{k=1}^{n} k \]

Using recursion:

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

Names:

Using while:

\[ n, \text{total}, k, \text{fact_iter} \]

Using recursion:

\[ n, \text{fact} \]
Verifying Recursive Functions
The Recursive Leap of Faith
The Recursive Leap of Faith

Photo by Kevin Lee, Preikestolen, Norway
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

Is fact implemented correctly?
The Recursive Leap of Faith

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

Is fact implemented correctly?

1. Verify the base case
The Recursive Leap of Faith

```python
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!
The Recursive Leap of Faith

```python
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
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
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 `sum_digits` function computes the sum of positive n correctly because:

```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 \( n < 10 \) is \( n \) \hspace{1cm} \text{(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 \( \text{n} < 10 \) is \( \text{n} \). \( \text{(base case)} \)

Assuming \( \text{sum_digits(k)} \) correctly sums the digits of \( k \) \( \text{(assume correct)} \)

```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 any n < 10 is n. (base case)

Assuming sum_digits(k) correctly sums the digits of k for all k with fewer digits than n, (assume correct)

for all n ≥ 10, (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 $n < 10$ is $n$. \hspace{1cm} \text{(base case)}

Assuming \hspace{1cm} \text{(assume correct)}

for all \hspace{1cm} \text{(simpler case)}

sum_digits(n) will \hspace{1cm} \text{(conclusion)}

```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 any \( n < 10 \) is \( n \). \hfill \text{(base case)}

Assuming \text{sum_digits}(k) \text{ correctly sums the digits of } k \text{ for all } k \text{ with fewer digits than } n \text{,} \hfill \text{(assume correct)}

for all \hfill \text{(simpler case)}

sum_digits(n) will \hfill \text{(conclusion)}

```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 any \( n < 10 \) is \( n \). \hfill (base case)

Assuming \texttt{sum_digits(k)} correctly sums the digits of \( k \) \hfill (assume correct)

for all \hfill (simpler case)

\texttt{sum_digits(n)} will \hfill (conclusion)

\begin{verbatim}
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
\end{verbatim}
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 \( n \). \hspace{1cm} \text{(base case)}

Assuming \( \text{sum_digits}(k) \) correctly sums the digits of \( k \)

for all \( k \) with fewer digits than \( n \), \hspace{1cm} \text{(assume correct)}

\( \text{sum_digits}(n) \) will \hspace{1cm} \text{(simpler case)}

\( \text{(conclusion)} \)

```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 any `n < 10` is `n`. (base case)

Assuming `sum_digits(k)` correctly sums the digits of `k` (assume correct)

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

`sum_digits(n)` will add the digit sum of `n // 10` to `n % 10` (conclusion)

```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
```
Mutual Recursion
The Luhn Algorithm
The Luhn Algorithm

Used to verify credit card numbers
The Luhn Algorithm

Used to verify credit card numbers

The Luhn Algorithm

Used to verify credit card numbers


- **First**: 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


• First: 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)

• Second: Take the sum of all the digits
The Luhn Algorithm

Used to verify credit card numbers


• **First**: 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)

• **Second**: Take the sum of all the digits

```
1  3  8  7  4  3
```
The Luhn Algorithm

Used to verify credit card numbers


- **First:** 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)

- **Second:** Take the sum of all the digits

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>1+6=7</td>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>
The Luhn Algorithm

Used to verify credit card numbers


• **First:** 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)

• **Second:** Take the sum of all the digits

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1+6=7</td>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

= 30
The Luhn Algorithm

Used to verify credit card numbers


• **First:** 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)

• **Second:** Take the sum of all the digits

```
+-----+-----+-----+-----+-----+-----+
| 1   | 3   | 8   | 7   | 4   | 3   |
| 2   | 3   | 1+6=7| 7   | 8   | 3   |
+-----+-----+-----+-----+-----+-----+
```

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

Used to verify credit card numbers


• **First:** 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)

• **Second:** 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.

```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
```
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.

```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
```

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.

```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

# A partial sum
# 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.

```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

# A partial sum
# 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.

```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
```

(Demo)
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.

```python
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.

```python
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.

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

```python
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.

```python
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)
```

Updates via assignment become...

...arguments to a recursive call