Announcements
Data Representations
Functions with Shared Local State

Interactive Diagram
Functions with Shared Local State

def box(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
    return get, put

get, put = box('Hello')
before = get()
put('Goodbye')
after = get
Functions with Shared Local State

def box(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
    return get, put

get, put = box('Hello')
before = get()
put('Goodbye')
after = get()
Pairs Implemented as Functions
Pairs Implemented as Functions

def pair(x, y):
    def dispatch(m):
        if m == 'first':
            return x
        elif m == 'second':
            return y
    return dispatch

5
def pair(x, y):
    def dispatch(m):
        if m == 'first':
            return x
        elif m == 'second':
            return y
    return dispatch

This function represents the pair (x, y)
Pairs Implemented as Functions

```python
def pair(x, y):
    def dispatch(m):
        if m == 'first':
            return x
        elif m == 'second':
            return y
    return dispatch
```

This function represents the pair \((x, y)\)

Constructor is a higher-order function
def pair(x, y):
    def dispatch(m):
        if m == 'first':
            return x
        elif m == 'second':
            return y
    return dispatch

>>> p = pair(3, pair(4, 5))
def pair(x, y):
    def dispatch(m):
        if m == 'first':
            return x
        elif m == 'second':
            return y
    return dispatch

This function represents the pair (x, y)

Constructor is a higher-order function

>>> p = pair(3, pair(4, 5))
>>> p('first')
3
def pair(x, y):
    def dispatch(m):
        if m == 'first':
            return x
        elif m == 'second':
            return y
    return dispatch

This function represents the pair (x, y)

Constructor is a higher-order function

>>> p = pair(3, pair(4, 5))
>>> p('first')
3
>>> p('second')('first')
4
```python
def pair(x, y):
    def dispatch(m):
        if m == 'first':
            return x
        elif m == 'second':
            return y
    return dispatch
```

This function represents the pair \((x, y)\)

```bash
>>> p = pair(3, pair(4, 5))
>>> p('first')
3
>>> p('second')('first')
4
>>> p('second')('second')
5
```

Constructor is a higher-order function
def pair(x, y):
    def dispatch(m):
        if m == 'first':
            return x
        elif m == 'second':
            return y
    return dispatch

This function represents the pair (x, y)

Constructor is a higher-order function

>>> p = pair(3, pair(4, 5))
>>> p('first')
3
>>> p('second')('first')
4
>>> p('second')('second')
5
def pair(x, y):
    def dispatch(m):
        if m == 'first':
            return x
        elif m == 'second':
            return y
    return dispatch

This function represents the pair (x, y)

Constructor is a higher-order function

>>> p = pair(3, pair(4, 5))
>>> p('first')
3
>>> p('second')('first')
4
>>> p('second')('second')
5

(Demo)
Linked Lists (Sneak Preview)
An empty list is called "nil" and represented as None
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

![Diagram of a linked list with elements 3, 4, and 5]
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None

- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

\[ \text{nil} = \text{None} \]
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None

- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

```python
nil = None
def list_len(s):
```

![Diagram of linked list with 3, 4, and 5 nodes]
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

```
nil = None
def list_len(s):
    if s is nil:
```
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None

- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

```python
def list_len(s):
    if s is nil:
        return 0
```

`nil = None`
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

```
nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
        return 0
```
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None

- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

```
nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))

def append(s, x):
```

```
3 → 4 → 5
```
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

```
nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))

def append(s, x):
    if s is nil:
    ```
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

```
nil = None

def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))

def append(s, x):
    if s is nil:
        return pair(x, nil)
```

```
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

```python
def append(s, x):
    if s is nil:
        return pair(x, nil)
    else:
        return 1 + list_len(s('second'))
```

```
nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))
```
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

```
nil = None

def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))

def append(s, x):
    if s is nil:
        return pair(x, nil)
    else:
        first, second = s('first'), s('second')
        return pair(first, pair(x, second))
```
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

```
def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))

def append(s, x):
    if s is nil:
        return pair(x, nil)
    else:
        first, second = s('first'), s('second')
        return pair(first, append(second, x))
```

```
nil = None
```
Linked Lists (Sneak Preview)

- An empty list is called "nil" and represented as None

- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list

```
nil = None

def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))

def append(s, x):
    if s is nil:
        return pair(x, nil)
    else:
        first, second = s('first'), s('second')
        return pair(first, append(second, x))
```

(Demo)
An Inefficient Dictionary Implementation

- A list of key–value pairs can be used to implement dictionary behavior
An Inefficient Dictionary Implementation

- A list of key-value pairs can be used to implement dictionary behavior

```python
>>> d = dict_dispatch()
>>> d('set')('I', 1)
>>> d('set')('V', 5)
>>> d('set')('X', 10)
```
An Inefficient Dictionary Implementation

• A list of key-value pairs can be used to implement dictionary behavior

```python
>>> d = dict_dispatch()
>>> d('set')('I', 1)
>>> d('set')('V', 5)
>>> d('set')('X', 10)
```

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'I'</td>
<td>1</td>
</tr>
<tr>
<td>'V'</td>
<td>5</td>
</tr>
<tr>
<td>'X'</td>
<td>10</td>
</tr>
</tbody>
</table>
An Inefficient Dictionary Implementation

• A list of key-value pairs can be used to implement dictionary behavior

```python
>>> d = dictDispatch()
>>> d('set')('I', 1)
>>> d('set')('V', 5)
>>> d('set')('X', 10)
```

(Demo)
Dispatch Dictionaries
Dispatch Dictionaries
Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:
Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:
- Equality tests are repetitive
Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:
- Equality tests are repetitive
- We can't add new messages without re-writing the dispatch function
Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:
- Equality tests are repetitive
- We can't add new messages without re-writing the dispatch function

A dispatch dictionary has messages as keys and functions (or data objects) as values
Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:
- Equality tests are repetitive
- We can't add new messages without re-writing the dispatch function

A dispatch dictionary has messages as keys and functions (or data objects) as values.

Dictionaries handle the message look-up logic; we can concentrate on implementing behavior.
Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:
- Equality tests are repetitive
- We can't add new messages without re-writing the dispatch function

A dispatch dictionary has messages as keys and functions (or data objects) as values. Dictionaries handle the message look-up logic; we can concentrate on implementing behavior.

```python
def box_dispatch(contents):
    def dispatch(m):
        if m == 'contents':
            return contents
        if m == 'put':
            def put(value):
                nonlocal contents
                contents = value
                return put
        return dispatch
```
Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:
  - Equality tests are repetitive
  - We can't add new messages without re-writing the dispatch function

A dispatch dictionary has messages as keys and functions (or data objects) as values

Dictionaries handle the message look-up logic; we can concentrate on implementing behavior

```python
def box_dispatch(contents):
    def dispatch(m):
        if m == 'contents':
            return contents
        if m == 'put':
            def put(value):
                nonlocal contents
                contents = value
                return put
    return dispatch

def box_dict(contents):
    def put(value):
        d['contents'] = value
    d = {'contents': contents, 'put': put}
    return d
```
Dispatch Dictionaries

Enumerating different messages in a conditional statement isn't very convenient:

- Equality tests are repetitive
- We can't add new messages without re-writing the dispatch function

A dispatch dictionary has messages as keys and functions (or data objects) as values

Dictionaries handle the message look-up logic; we can concentrate on implementing behavior

```python
def box_dispatch(contents):
    def dispatch(m):
        if m == 'contents':
            return contents
        if m == 'put':
            def put(value):
                nonlocal contents
                contents = value
                return put
    return dispatch

def box_dict(contents):
    def put(value):
        d['contents'] = value
        d = {'contents': contents, 'put': put}
    return d

(Demo)
```
Constraint Networks
Solving for Variables in an Equation
Solving for Variables in an Equation

\[ a + b = c \]
Solving for Variables in an Equation

\[ a + b = c \]
\[ a = c - b \]
Solving for Variables in an Equation

\[ a + b = c \]
\[ a = c - b \]
\[ b = c - a \]
Solving for Variables in an Equation

\[ a + b = c \]
\[ a = c - b \]
\[ b = c - a \]

Algebraic equations are *declarative*: They describe a relation among different quantities.
Solving for Variables in an Equation

\[ a + b = c \]
\[ a = c - b \]
\[ b = c - a \]

Algebraic equations are *declarative*: They describe a relation among different quantities.

Python functions are *procedural*: They describe how to compute a result from a set of input arguments.
Algebraic equations are *declarative*: They describe a relation among different quantities.

Python functions are *procedural*: They describe how to compute a result from a set of input arguments.

Constraint programming:
Solving for Variables in an Equation

\[ a + b = c \]
\[ a = c - b \]
\[ b = c - a \]

Algebraic equations are *declarative*: They describe a relation among different quantities.

Python functions are *procedural*: They describe how to compute a result from a set of input arguments.

Constraint programming:
- We define the relationship between quantities.
Solving for Variables in an Equation

\[ a + b = c \]
\[ a = c - b \]
\[ b = c - a \]

Algebraic equations are *declarative*: They describe a relation among different quantities.

Python functions are *procedural*: They describe how to compute a result from a set of input arguments.

Constraint programming:
- We define the relationship between quantities
- We provide values for the "known" quantities
Solving for Variables in an Equation

\[
\begin{align*}
    a + b &= c \\
    a &= c - b \\
    b &= c - a
\end{align*}
\]

Algebraic equations are *declarative*: They describe a relation among different quantities.

Python functions are *procedural*: They describe how to compute a result from a set of input arguments.

**Constraint programming:**
- We define the relationship between quantities.
- We provide values for the "known" quantities.
- The system computes values for the "unknown" quantities.
Solving for Variables in an Equation

\[
a + b = c \\
a = c - b \\
b = c - a
\]

Algebraic equations are *declarative*: They describe a relation among different quantities

*ONE WAY* Python functions are *procedural*: They describe how to compute a result from a set of input arguments

Constraint programming:
- We define the relationship between quantities
- We provide values for the "known" quantities
- The system computes values for the "unknown" quantities

**Challenge**: We want a general means of combination.
Solving for Variables in an Equation

\[
\begin{align*}
    a + b &= c \\
    a &= c - b \\
    b &= c - a \\
    p \times v &= n \times k \times t
\end{align*}
\]

Algebraic equations are *declarative*: They describe a relation among different quantities.

Python functions are *procedural*: They describe how to compute a result from a set of input arguments.

Constraint programming:
- We define the relationship between quantities
- We provide values for the "known" quantities
- The system computes values for the "unknown" quantities

**Challenge:** We want a general means of combination.
Solving for Variables in an Equation

\[ a + b = c \]
\[ a = c - b \]
\[ b = c - a \]

Algebraic equations are *declarative*: They describe a relation among different quantities.

Python functions are *procedural*: They describe how to compute a result from a set of input arguments.

Constraint programming:
- We define the relationship between quantities
- We provide values for the "known" quantities
- The system computes values for the "unknown" quantities

**Challenge:** We want a general means of combination.
Solving for Variables in an Equation

\[
\begin{align*}
  a + b &= c \\
  a &= c - b \\
  b &= c - a \\
  p \times v &= n \times k \times t \\
  9 \times c &= 5 \times (f - 32)
\end{align*}
\]

Boltzmann’s constant

Algebraic equations are *declarative*: They describe a relation among different quantities

Python functions are *procedural*: They describe how to compute a result from a set of input arguments

Constraint programming:
- We define the relationship between quantities
- We provide values for the "known" quantities
- The system computes values for the "unknown" quantities

**Challenge:** We want a general means of combination.
A Constraint Network for Temperature Conversion

\[ 9 \times \text{celsius} = 5 \times (\text{fahrenheit} - 32) \]
A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

\[ 9 \times \text{celsius} = 5 \times (\text{fahrenheit} - 32) \]
A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

\[ 9 \times \text{celsius} = 5 \times (\text{fahrenheit} - 32) \]

This quantity relates directly to \text{celsius}
A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

\[ 9 \times \text{celsius} = 5 \times (\text{fahrenheit} - 32) \]

This quantity relates directly to \text{celsius}.
A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

\[ 9 \times \text{celsius} = 5 \times (\text{fahrenheit} - 32) \]

This quantity relates directly to \text{celsius}.

This quantity relates directly to \text{fahrenheit}.
A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

\[ 9 \times \text{celsius} = 5 \times (\text{fahrenheit} - 32) \]
A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

\[ u = 9 \times \text{celsius} \]

This quantity relates directly to \text{celsius}

\[ v = 5 \times (\text{fahrenheit} - 32) \]

This quantity relates directly to \text{fahrenheit}
A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

\[ u = 9 \times \text{celsius} = 5 \times (\text{fahrenheit} - 32) \]

This quantity relates directly to \text{celsius}

This quantity relates directly to \text{fahrenheit}

Both sides of the equation are equal: they must be the same quantity
A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

\[ 9 \times \text{celsius} = 5 \times (\text{fahrenheit} - 32) \]

Both sides of the equation are equal: they must be the same quantity.
A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

\[ 9 \times \text{celsius} = 5 \times (\text{fahrenheit} - 32) \]

This quantity relates directly to \textit{celsius}

This quantity relates directly to \textit{fahrenheit}

Both sides of the equation are equal: they must be the same quantity
A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

Both sides of the equation are equal: they must be the same quantity.

This quantity relates directly to celsius

This quantity relates directly to fahrenheit

\[ 9 \times \text{celsius} = 5 \times (\text{fahrenheit} - 32) \]