61A Extra Lecture 5

Announcements

Data Representations

Functions with Shared Local State

Interactive Diagram

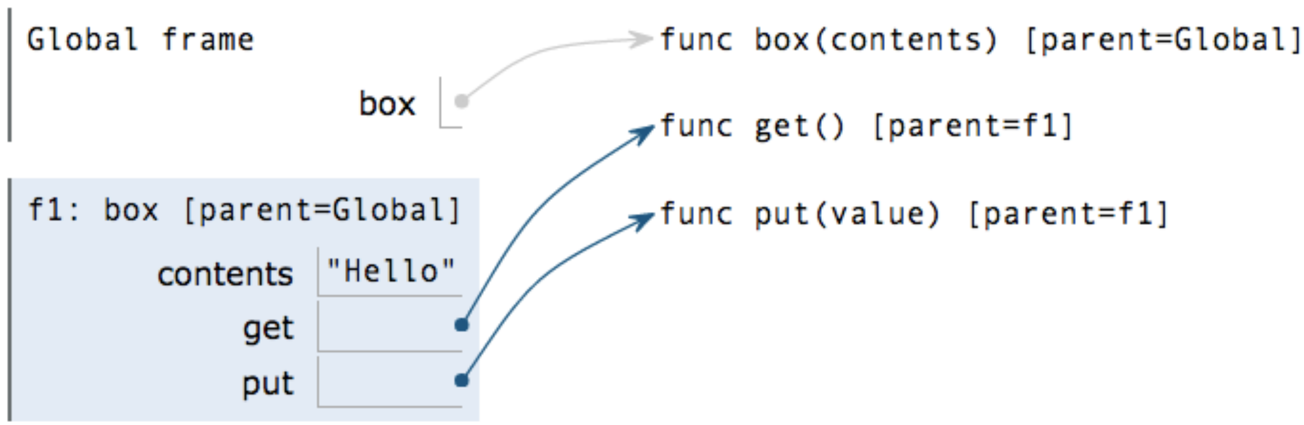
Functions with Shared Local State

```python
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()
```

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()
```



Interactive Diagram

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
```

Pairs Implemented as Functions

```
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

```
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

Pairs Implemented as Functions

```
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))
```

Pairs Implemented as Functions

```
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
```

Pairs Implemented as Functions

```
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
```

Pairs Implemented as Functions

```
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
```

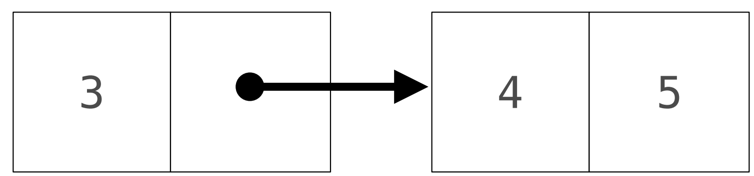
Pairs Implemented as Functions

```
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
```



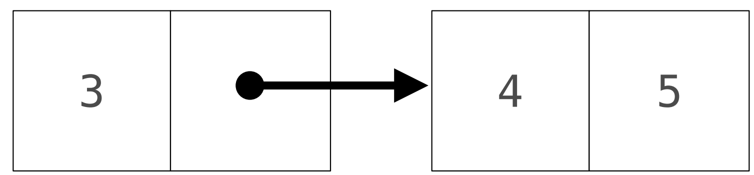
Pairs Implemented as Functions

```
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)

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)

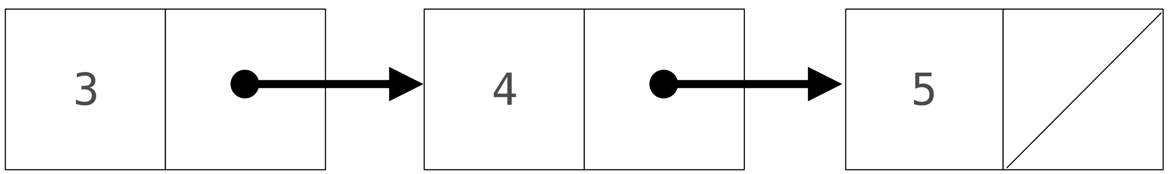
- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list

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)

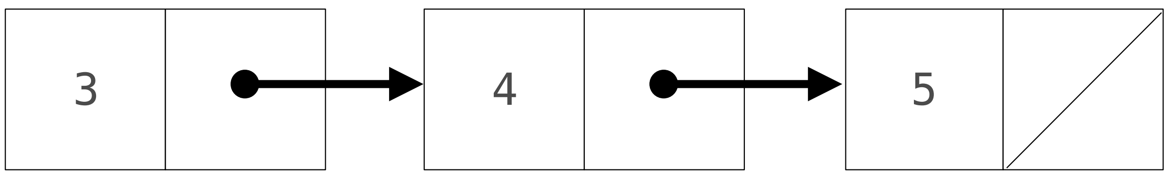
- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list



Linked Lists (Sneak Preview)

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

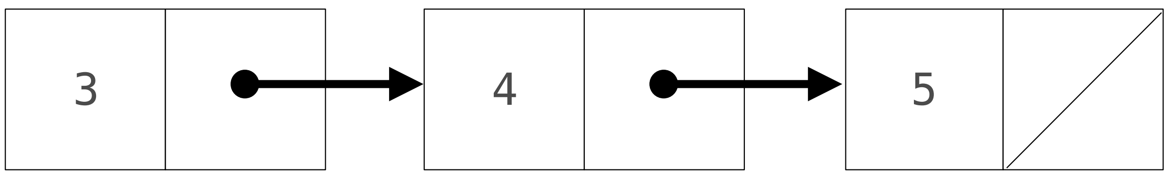
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list




```
nil = None
```

Linked Lists (Sneak Preview)

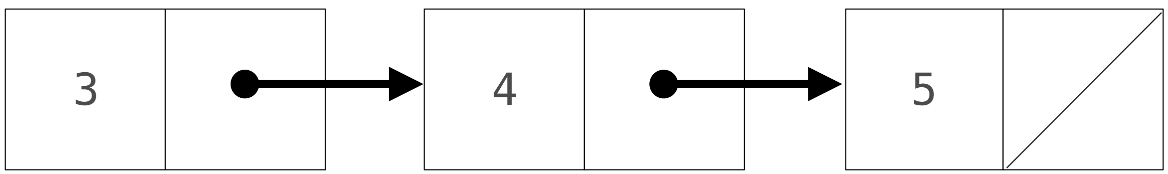
- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list




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

Linked Lists (Sneak Preview)

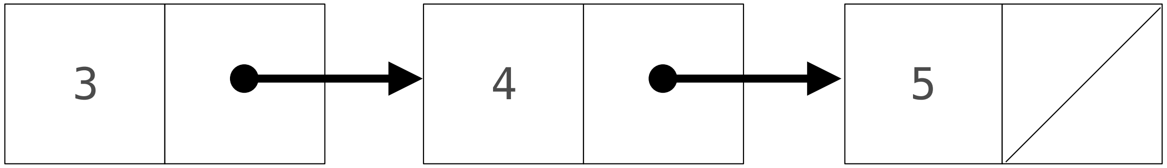
- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list




```
nil = None
def list_len(s):
    if s is nil:
```

Linked Lists (Sneak Preview)

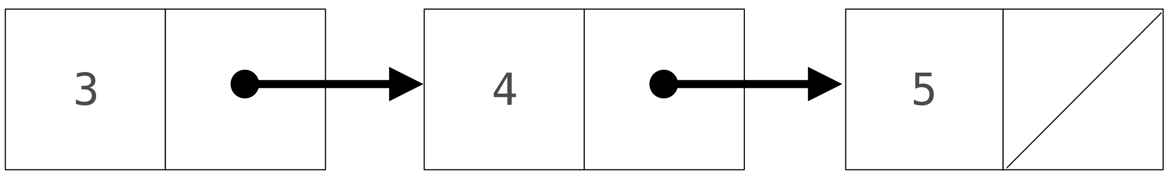
- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list




```
nil = None
def list_len(s):
    if s is nil:
        return 0
```

Linked Lists (Sneak Preview)

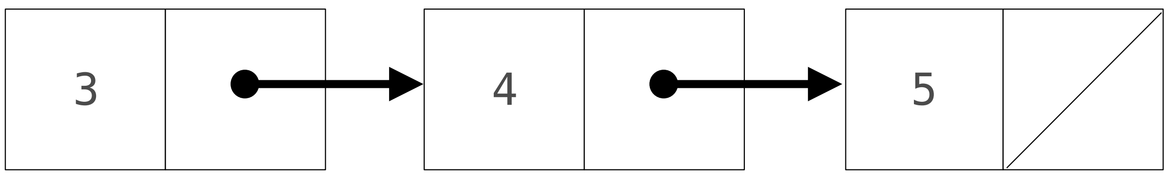
- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list




```
nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
```

Linked Lists (Sneak Preview)

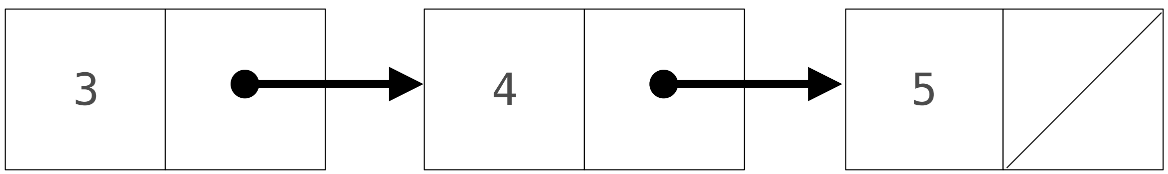
- An empty list is called "nil" and represented as None
- A non-empty list is represented as a pair
  - The first element of the pair is the first element of the list
  - The second element of the pair is the rest of the list




```
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

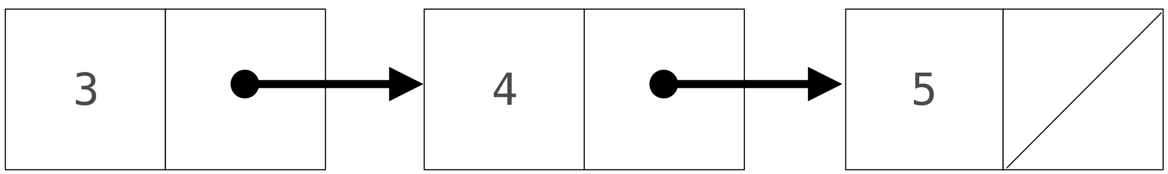



```
nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))
```

```
def append(s, x):
```

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

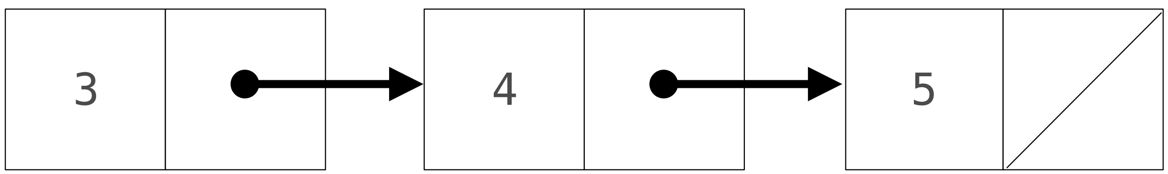



```
nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))
```

```
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

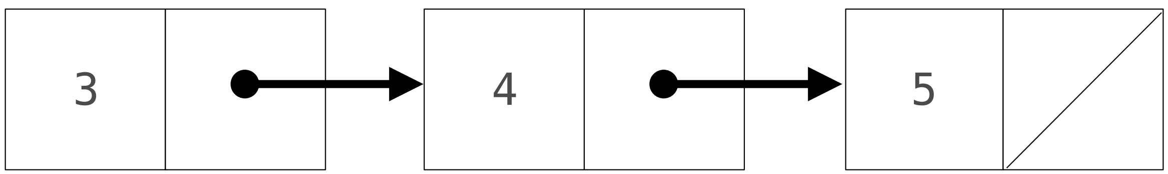



```
nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))
```

```
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



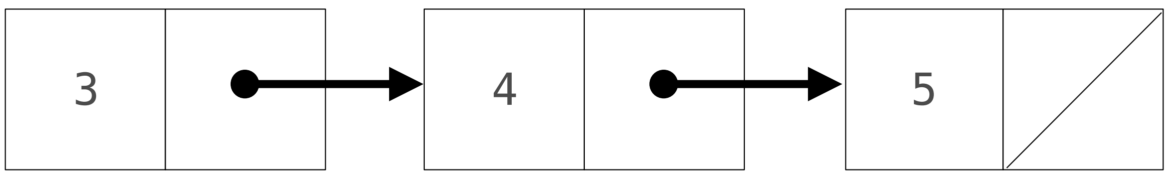

```
nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))
```

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

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

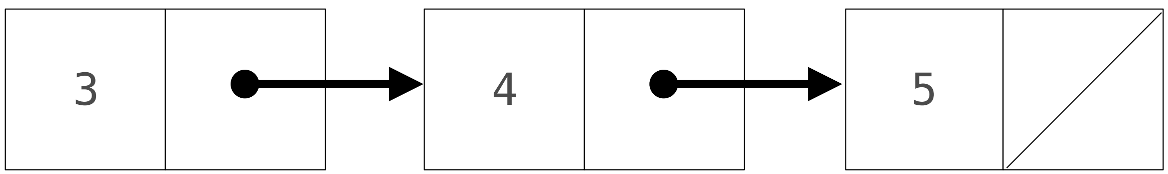



```
nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))
```

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

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

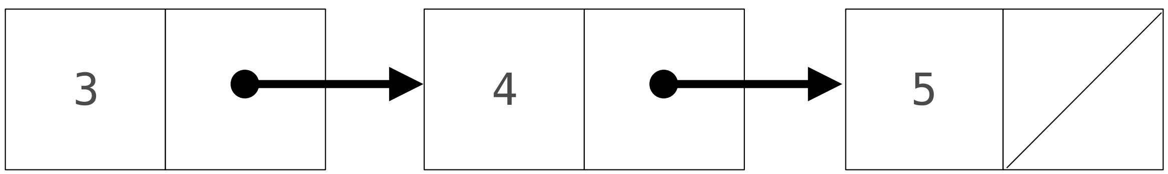



```
nil = None
def list_len(s):
    if s is nil:
        return 0
    else:
        return 1 + list_len(s('second'))
```

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

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(         ), s(         )
        return pair(first, append(second, x))
```

(Demo)

An Inefficient Dictionary Implementation

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

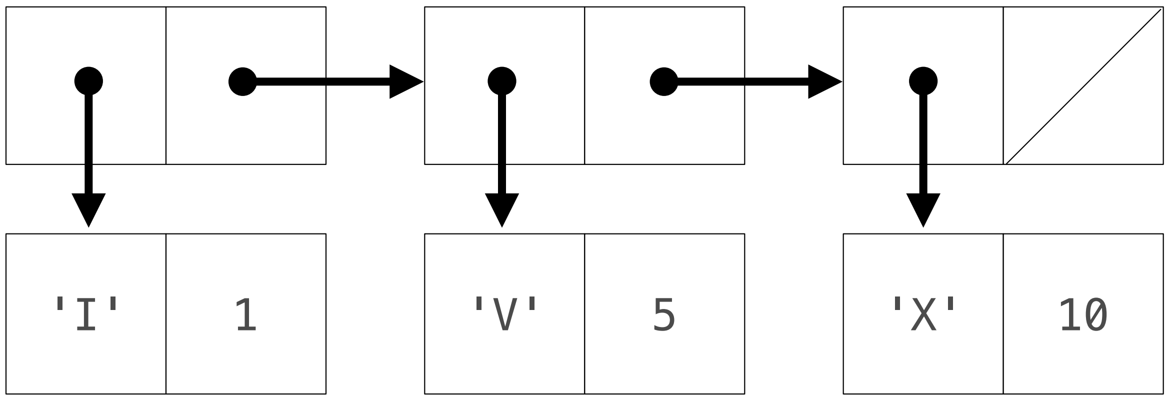
An Inefficient Dictionary Implementation

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

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

An Inefficient Dictionary Implementation

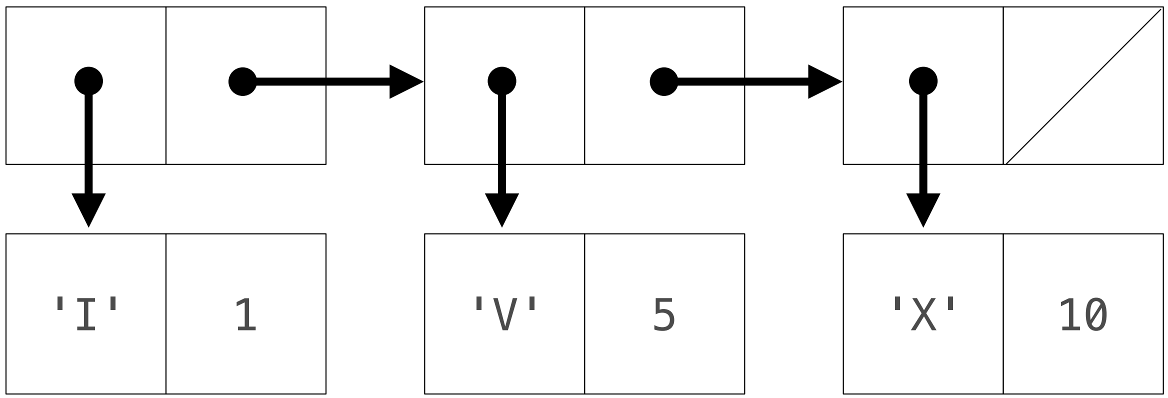
- A list of key-value pairs can be used to implement dictionary behavior




```
>>> 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




```
>>> d = dict_dispatch()
>>> 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

```
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

```
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

```
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

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:

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

```
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

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

$$a + b = c$$
$$a = c - b$$
$$b = c - a$$

$$p * v = n * k * t$$

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
```

Boltzmann's constant

```
p * v = n * k * t
```

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

$$a + b = c$$
$$a = c - b$$
$$b = c - a$$

Boltzmann's constant

$$p * v = n * k * t$$
$$9 * c = 5 * (f - 32)$$

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 * celsius = 5 * (fahrenheit - 32)
```

A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

```
9 * celsius = 5 * (fahrenheit - 32)
```

A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

$\boxed{9 * \textbf{celsius}} = 5 * (\textbf{fahrenheit} - 32)$

This quantity relates directly to **celsius**

A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

u

$\boxed{9 * \textbf{celsius}} = 5 * (\textbf{fahrenheit} - 32)$

This quantity relates directly to **celsius**

A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

$$\overbrace{9 * \textbf{celsius}}^{u} = 5 * (\textbf{fahrenheit} - 32)$$

This quantity relates directly to **celsius**

This quantity relates directly to **fahrenheit**

A Constraint Network for Temperature Conversion

`Combination idea: All intermediate quantities have values too.`

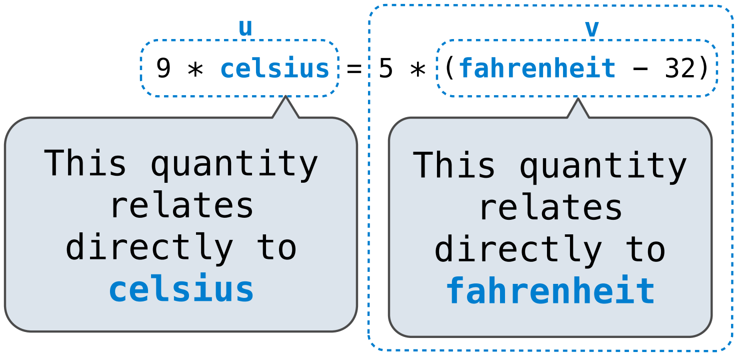
$$\overbrace{9 * \texttt{celsius}}^{u} = 5 * \overbrace{(\texttt{fahrenheit} - 32)}^{v}$$

This quantity relates directly to **celsius**

This quantity relates directly to **fahrenheit**

A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.



A Constraint Network for Temperature Conversion

```
Combination idea: All intermediate quantities have values too.
```

```
      u                    v
9 * celsius = 5 * (fahrenheit - 32)
```

This quantity relates directly to **celsius**

This quantity relates directly to **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.

$$\overbrace{9 * \texttt{celsius}}^{u} = \overbrace{5 * \overbrace{(\texttt{fahrenheit} - 32)}^{v}}^{u}$$

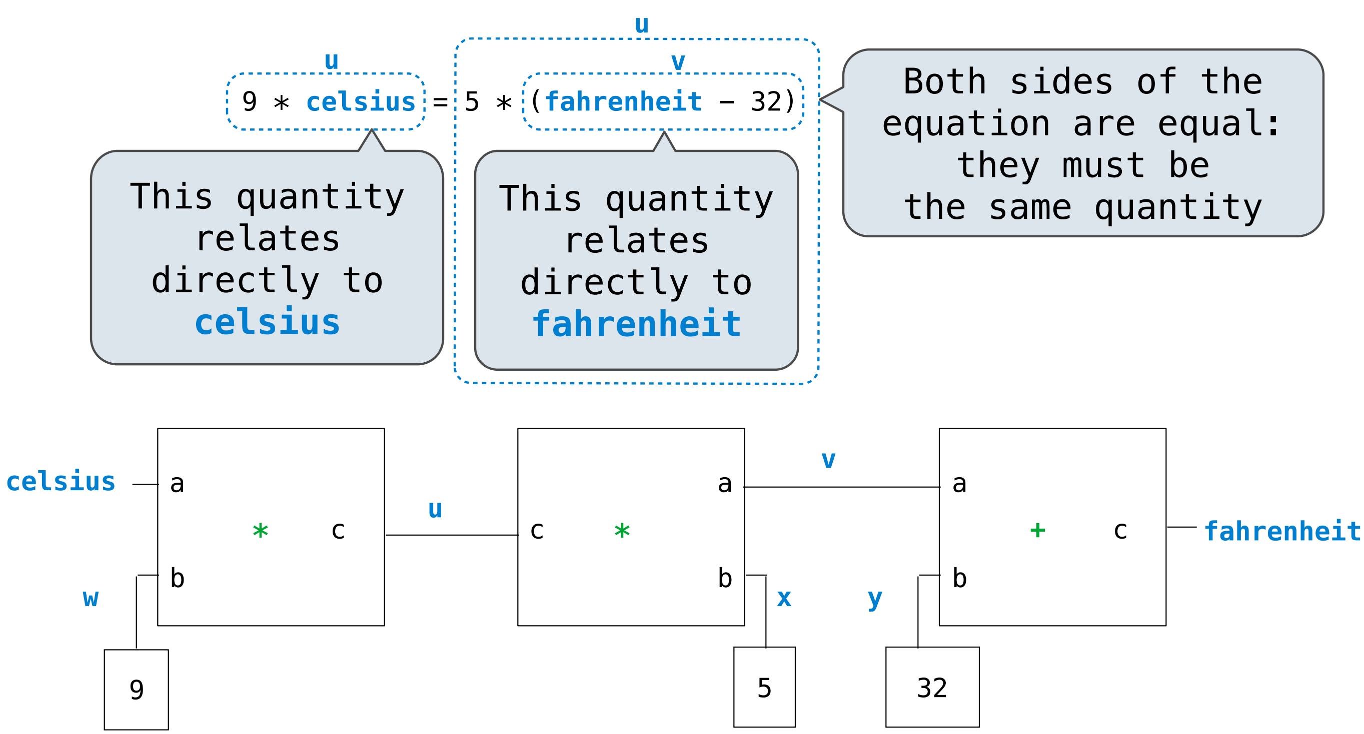
This quantity relates directly to **celsius**

This quantity relates directly to **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.



A Constraint Network for Temperature Conversion

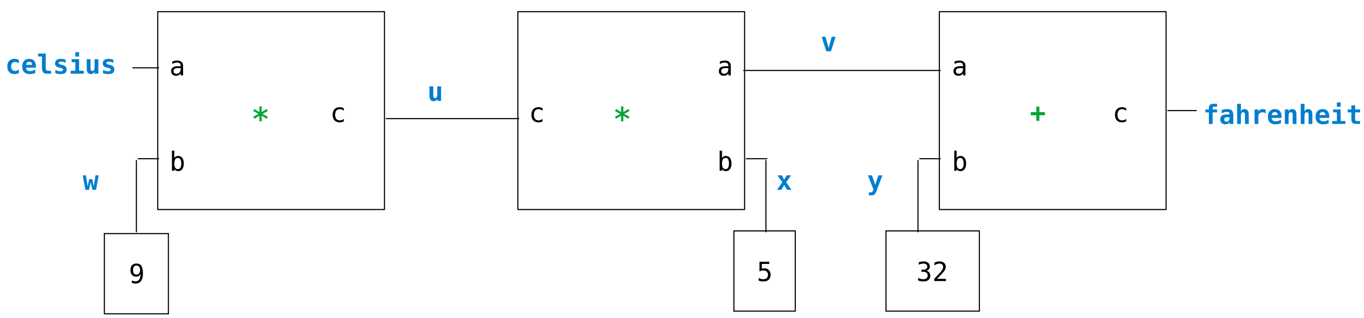
Combination idea: All intermediate quantities have values too.

$$\overset{u}{\boxed{9 * \textbf{celsius}}} = \overset{u}{\boxed{5 * \overset{v}{\boxed{(\textbf{fahrenheit} - 32)}}}}$$

This quantity relates directly to **celsius**

This quantity relates directly to **fahrenheit**

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



(Demo)