Functions with Shared Local State

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

ger, p = box('Hello')
before = get()
p('Goodbye')
after = get()
```

Interactive Diagram
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

• 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, rest = s('first'), s('second')
        return pair(first, append(rest, x))
```

(Demo)
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)
Dispatch Dictionaries
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

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

Boltzmann’s constant

\[
p \times v = n \times k \times t
\]

9 \times c = 5 \times (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

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 * celsius = 5 * (fahrenheit - 32)