61A Lecture 13

Wednesday, September 28
Limitations on Dictionaries
Limitations on Dictionaries

Dictionaries are unordered collections of key-value pairs.
Limitations on Dictionaries

Dictionaries are unordered collections of key-value pairs.

Dictionaries do have two restrictions:
Limitations on Dictionaries

Dictionaries are unordered collections of key–value pairs.

Dictionaries do have two restrictions:

• A key of a dictionary cannot be an object of a mutable built-in type.
Limitations on Dictionaries

Dictionaries are unordered collections of key–value pairs.

Dictionaries do have two restrictions:

• A key of a dictionary **cannot be** an object of a **mutable** built-in type.

• Two **keys cannot be equal**. There can be at most one value for a given key.
Limitations on Dictionaries

Dictionaries are unordered collections of key–value pairs.

Dictionaries do have two restrictions:

- A key of a dictionary cannot be an object of a mutable built-in type.

- Two keys cannot be equal. There can be at most one value for a given key.

This first restriction is tied to Python's underlying implementation of dictionaries.
Limitations on Dictionaries

Dictionaries are unordered collections of key-value pairs.

Dictionaries do have two restrictions:

• A key of a dictionary cannot be an object of a mutable built-in type.

• Two keys cannot be equal. There can be at most one value for a given key.

This first restriction is tied to Python's underlying implementation of dictionaries.

The second restriction is an intentional consequence of the dictionary abstraction.
Implementing Dictionaries
Implementing Dictionaries

def make_dict():
    """Return a functional implementation of a dictionary."""
Implementing Dictionaries

```python
def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []
```
Implementing Dictionaries

def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    return getitem
Implementing Dictionaries

```python
def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):
```

"""Tuesday, September 27, 2011"""
Implementing Dictionaries

def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
Implementing Dictionaries

```python
def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
        return
    records.append([key, value])
```
def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])

    def dispatch(message, key=None, value=None):
Implementing Dictionaries

```python
def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
        return records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
```

Tuesday, September 27, 2011
Implementing Dictionaries

```python
def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v

    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
```

Tuesday, September 27, 2011
Implementing Dictionaries

def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
Implementing Dictionaries

```python
def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

def getitem(key):
    for k, v in records:
        if k == key:
            return v

def setitem(key, value):
    for item in records:
        if item[0] == key:
            item[1] = value
            return
    records.append([key, value])

def dispatch(message, key=None, value=None):
    if message == 'getitem':
        return getitem(key)
    elif message == 'setitem':
        setitem(key, value)
    elif message == 'keys':
        return tuple(k for k, _ in records)
    elif message == 'values':
        return tuple(v for _, v in records)
```

Tuesday, September 27, 2011
Implementing Dictionaries

```python
def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
    
    records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
        elif message == 'values':
            return tuple(v for _, v in records)
    
    return dispatch
```

Tuesday, September 27, 2011
Implementing Dictionaries

```python
def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
        return records.append([key, value])

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
        elif message == 'values':
            return tuple(v for _, v in records)

    return dispatch
```

Question: Do we need a nonlocal statement here?
Implementing Dictionaries

```python
def make_dict():
    """Return a functional implementation of a dictionary."""
    records = []

    def getitem(key):
        for k, v in records:
            if k == key:
                return v
    def setitem(key, value):
        for item in records:
            if item[0] == key:
                item[1] = value
                return
        records.append([key, value])
    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
        elif message == 'keys':
            return tuple(k for k, _ in records)
        elif message == 'values':
            return tuple(v for _, v in records)
        return dispatch

return dispatch
```

Question: Do we need a nonlocal statement here?
Message Passing
Message Passing

An approach to organizing the relationship among different pieces of a program
Message Passing

An approach to organizing the relationship among different pieces of a program

Different objects pass messages to each other
Message Passing

An approach to organizing the relationship among different pieces of a program

Different objects pass messages to each other

• What is your fourth element?
Message Passing

An approach to organizing the relationship among different pieces of a program

Different objects pass messages to each other

• What is your fourth element?
• Change your third element to this new value. (please)
Message Passing

An approach to organizing the relationship among different pieces of a program

Different objects pass messages to each other

• What is your fourth element?
• Change your third element to this new value. (please)

Encapsulates the behavior of all operations on a piece of data within one function that responds to different messages.
Message Passing

An approach to organizing the relationship among different pieces of a program

Different objects pass messages to each other

- What is your fourth element?
- Change your third element to this new value. (please)

Encapsulates the behavior of all operations on a piece of data within one function that responds to different messages.

Important historical interest: the message passing approach strongly influenced object-oriented programming (next lecture).
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 writing new code
Dispatch Dictionaries

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

- Equality tests are repetitive
- We can't add new messages without writing new code

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 writing new code

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

Dictionaries handle the message look-up logic; we concentrate on implementing useful 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 writing new code

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

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

In Javascript, all objects are just dictionaries
Enumerating different messages in a conditional statement isn't very convenient:

- Equality tests are repetitive
- We can't add new messages without writing new code

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

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

Demo

In Javascript, all objects are just dictionaries.
Example: Constraint Programming
Example: Constraint Programming

\[ a + b = c \]
Example: Constraint Programming

\[ a + b = c \]
\[ a = c - b \]
Example: Constraint Programming

\[ a + b = c \]
\[ a = c - b \]
\[ b = c - a \]
Example: Constraint Programming

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

Algebraic equations are *declarative*. They describe how different quantities relate to one another.
Example: Constraint Programming

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

Algebraic equations are *declarative*. They describe how different quantities relate to one another.

Python functions are *procedural*. They describe how to compute a particular result from a particular set of inputs.
Example: Constraint Programming

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

Algebraic equations are *declarative*. They describe how different quantities relate to one another.

Python functions are *procedural*. They describe how to compute a particular result from a particular set of inputs.

Constraint programming:
Example: Constraint Programming

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

Algebraic equations are *declarative*. They describe how different quantities relate to one another.

Python functions are *procedural*. They describe how to compute a particular result from a particular set of inputs.

Constraint programming:

- We define the relationship between quantities
Example: Constraint Programming

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

Algebraic equations are *declarative*. They describe how different quantities relate to one another.

Python functions are *procedural*. They describe how to compute a particular result from a particular set of inputs.

Constraint programming:
- We define the relationship between quantities
- We provide values for the "known" quantities
Example: Constraint Programming

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

Algebraic equations are *declarative*. They describe how different quantities relate to one another.

Python functions are *procedural*. They describe how to compute a particular result from a particular set of inputs.

Constraint programming:

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

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

Algebraic equations are *declarative*. They describe how different quantities relate to one another.

Python functions are *procedural*. They describe how to compute a particular result from a particular set of inputs.

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.
Example: Constraint Programming

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

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

Algebraic equations are *declarative*. They describe how different quantities relate to one another.

Python functions are *procedural*. They describe how to compute a particular result from a particular set of inputs.

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.
Example: Constraint Programming

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

Algebraic equations are *declarative*. They describe how different quantities relate to one another.

Python functions are *procedural*. They describe how to compute a particular result from a particular set of inputs.

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}.
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.

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

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.

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

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.

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

This quantity relates directly to \text{celsius}

This quantity relates directly to \text{fahrenheit}

Both sides 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 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 \text{celsius}

This quantity relates directly to \text{fahrenheit}

Both sides equal: they must be the same quantity
Anatomy of a Constraint

celsius

\[ \begin{array}{c}
a \\
\ast \\
b \\
w
\end{array} \]

\[ \begin{array}{c}
c \\
\ast \\
u
\end{array} \]
Anatomy of a Constraint

Blue names are "connectors"
Anatomy of a Constraint

Blue names are "connectors"

Boxes are "constraints"
Anatomy of a Constraint

Blue names are "connectors"

Boxes are "constraints"

Constraints compute values for "unknown" connectors
Anatomy of a Constraint

- **Connectors** represent quantities that have values.

Blue names are "connectors"

Boxes are "constraints"

Constraints compute values for "unknown" connectors
Anatomy of a Constraint

- **Connectors** represent quantities that have values.
- **Constraints** spread information among connectors.
Blue names are "connectors"

Boxes are "constraints"

- **Connectors** represent quantities that have values.
- **Constraints** spread information among connectors.
- A constraint can receive two messages from its connectors:
Anatomy of a Constraint

Blue names are "connectors"

 Boxes are "constraints"

• Connectors represent quantities that have values.
• Constraints spread information among connectors.
• A constraint can receive two messages from its connectors:
  • 'new_val' indicates that some connector that is connected to the constraint has a new value.
Anatomy of a Constraint

- **Connectors** represent quantities that have values.
- **Constraints** spread information among connectors.
- A constraint can receive two messages from its connectors:
  - `'new_val'` indicates that some connector that is connected to the constraint has a new value.
  - `'forget'` indicates that some connector that is connected to the constraint has forgotten its value.
Constructing a Constraint Network
Constructing a Constraint Network

celsius = make_connector('Celsius')
Constructing a Constraint Network

celsius

celsius = make_connector('Celsius')
Constructing a Constraint Network

celsius

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
Constructing a Constraint Network

celsius

fahrenheit

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
Constructing a Constraint Network

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""
    u, v, w, x, y = [make_connector() for _ in range(5)]

celsius

fahrenheit

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""
    u, v, w, x, y = [make_connector() for _ in range(5)]

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""
    u, v, w, x, y = [make_connector() for _ in range(5)]
    multiplier(celsius, w, u)

celsius = make_connector('Celsius')
fahrenheit = make_converter(celsius, fahrenheit)
def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""
    u, v, w, x, y = [make_connector() for _ in range(5)]
    multiplier(celsius, w, u)

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""
    u, v, w, x, y = [make_connector() for _ in range(5)]
    multiplier(celsius, w, u)
    multiplier(v, x, u)

    celsius = make_connector('Celsius')
    fahrenheit = make_connector('Fahrenheit')
    make_converter(celsius, fahrenheit)
def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""
    u, v, w, x, y = [make_connector() for _ in range(5)]
    multiplier(celsius, w, u)
    multiplier(v, x, u)

    celsius = make_connector('Celsius')
    fahrenheit = make_connector('Fahrenheit')
    make_converter(celsius, fahrenheit)
Constructing a Constraint Network

def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""
    u, v, w, x, y = [make_connector() for _ in range(5)]
    multiplier(celsius, w, u)
    multiplier(v, x, u)
    adder(v, y, fahrenheit)

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""
    u, v, w, x, y = [make_connector() for _ in range(5)]
    multiplier(celsius, w, u)
    multiplier(v, x, u)
    adder(v, y, fahrenheit)

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""
    u, v, w, x, y = [make_connector() for _ in range(5)]
    multiplier(celsius, w, u)
    multiplier(v, x, u)
    adder(v, y, fahrenheit)
    constant(w, 9)

    celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
Constructing a Constraint Network

```python
def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""
    u, v, w, x, y = [make_connector() for _ in range(5)]
    multiplier(celsius, w, u)
    multiplier(v, x, u)
    adder(v, y, fahrenheit)
    constant(w, 9)
    constant(x, 5)

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
```

Tuesday, September 27, 2011
def make_converter(celsius, fahrenheit):
    """Make a temperature conversion network."""
    u, v, w, x, y = [make_connector() for _ in range(5)]
    multiplier(celsius, w, u)
    multiplier(v, x, u)
    adder(v, y, fahrenheit)
    constant(w, 9)
    constant(x, 5)
    constant(y, 32)

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
Constructing a Constraint Network

```python
def make_converter(celsius, fahrenheit):
    '''Make a temperature conversion network.'''
u, v, w, x, y = [make_connector() for _ in range(5)]
multiplier(celsius, w, u)
multiplier(v, x, u)
adder(v, y, fahrenheit)
constant(w, 9)
constant(x, 5)
constant(y, 32)
celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
```

Demo Tuesday, September 27, 2011
The Messages of a Connector
The Messages of a Connector

```python
calculator = make_calculator('Celsius')
```
The Messages of a Connector

\[
\text{connector} = \text{make\_connector('Celsius')}
\]

\[
\text{connector['set\_val']}(\text{source}, \text{value}) \text{ indicates that the source is requesting the connector to set its current value to value.}
\]
The Messages of a Connector

```
connector = make_connector('Celsius')
```

`connector['set_val'](source, value)` indicates that the source is requesting the connector to set its current value to value.

`connector['has_val']()` returns whether the connector already has a value.
connector = make_connector('Celsius')

connector['set_val'](source, value) indicates that the source is requesting the connector to set its current value to value.

connector['has_val']() returns whether the connector already has a value.

connector['val'] is the current value of the connector.
The Messages of a Connector

```
connector = make_connector('Celsius')
```

`connector['set_val']`(source, value) indicates that the source is requesting the connector to set its current value to value.

`connector['has_val']`() returns whether the connector already has a value.

`connector['val']` is the current value of the connector.

`connector['forget']`(source) tells the connector that the source is requesting it to forget its value.
The Messages of a Connector

```
connector = make_connector('Celsius')
```

`connector['set_val'](source, value)` indicates that the source is requesting the connector to set its current value to `value`.

`connector['has_val']()` returns whether the connector already has a value.

`connector['val']` is the current value of the connector.

`connector['forget'](source)` tells the connector that the source is requesting it to forget its value.

`connector['connect'](source)` tells the connector to participate in a new constraint, the `source`.
Implementing an Adder Constraint
Implementing an Adder Constraint

```python
def adder_constraint(a, b, c):
    """The constraint that a + b = c."

>>> a, b, c = [make_connector(name) for name in ('a', 'b', 'c')]
>>> constraint = adder_constraint(a, b, c)
>>> a['set_val']('user', 2)
a = 2
>>> b['set_val']('user', 3)
b = 3
c = 5
"""
```
Implementing an Adder Constraint

```python
def adder_constraint(a, b, c):
    """The constraint that \( a + b = c \)."""

    >>> a, b, c = [make_connector(name) for name in ('a', 'b', 'c')]
    >>> constraint = adder_constraint(a, b, c)
    >>> a['set_val']('user', 2)
    a = 2
    >>> b['set_val']('user', 3)
    b = 3
    c = 5
    """

    def new_value():
        ....
```

The code snippet above defines a function `adder_constraint(a, b, c)` which takes three connectors `a`, `b`, and `c` as inputs and establishes the constraint that `a + b = c`. The function is then used to set values for `a`, `b`, and `c` to 2, 3, and 5 respectively. The `new_value()` function is also defined, though its implementation is not shown in the snippet.
Implementing an Adder Constraint

def adder_constraint(a, b, c):
    """The constraint that \( a + b = c \)."

>>> a, b, c = [make_connector(name) for name in ('a', 'b', 'c')]
>>> constraint = adder_constraint(a, b, c)
>>> a['set_val']('user', 2)
a = 2
>>> b['set_val']('user', 3)
b = 3
c = 5
"""

def new_value():
    # We will implement this function momentarily!
Implementing an Adder Constraint

```python
def adder_constraint(a, b, c):
    """The constraint that a + b = c."

    >>> a, b, c = [make_connector(name) for name in ('a', 'b', 'c')]
    >>> constraint = adder_constraint(a, b, c)
    >>> a['set_val']('user', 2)
    a = 2
    >>> b['set_val']('user', 3)
    b = 3
    c = 5
    """

    def new_value():
        ...
        # We will implement this function momentarily!

    def forget_value():
        for connector in (a, b, c):
            connector['forget'](constraint)

    constraint = {'new_val': new_value, 'forget': forget_value}

    for connector in (a, b, c):
        connector['connect'](constraint)

    return constraint
```
Generalizing to a Multiplication Constraint
from operator import add, sub, mul, truediv

def adder(a, b, c):
    """The constraint that $a + b = c$.""
    return make_ternary_constraint(a, b, c, add, sub, sub)

def multiplier(a, b, c):
    """The constraint that $a * b = c$.""
    return make_ternary_constraint(a, b, c, mul, truediv, truediv)
Generalizing to a Multiplication Constraint

```python
def make_ternary_constraint(a, b, c, ab, ca, cb):
    """The constraint that ab(a,b)=c and ca(c,a)=b and cb(c,b)=a."""
    def new_value():
        av, bv, cv = [connector['has_val']() for connector in (a, b, c)]
        if av == ab and bv == ca and cv == cb:
            return True
        else:
            return False

    return new_value

def adder(a, b, c):
    """The constraint that a + b = c."""
    return make_ternary_constraint(a, b, c, add, sub, sub)

def multiplier(a, b, c):
    """The constraint that a * b = c."""
    return make_ternary_constraint(a, b, c, mul, truediv, truediv)
```

from operator import add, sub, mul, truediv

def adder(a, b, c):
    """The constraint that a + b = c."""
    return make_ternary_constraint(a, b, c, add, sub, sub)

def multiplier(a, b, c):
    """The constraint that a * b = c."""
    return make_ternary_constraint(a, b, c, mul, truediv, truediv)
Generalizing to a Multiplication Constraint

Connectors

```python
def make_ternary_constraint(a, b, c, ab, ca, cb):
    """The constraint that ab(a,b)=c and ca(c,a)=b and cb(c,b)=a.""

def new_value():
    av, bv, cv = [connector[ 'has_val']() for connector in (a, b, c)]

from operator import add, sub, mul, truediv

def adder(a, b, c):
    """The constraint that a + b = c.""
    return make_ternary_constraint(a, b, c, add, sub, sub)

def multiplier(a, b, c):
    """The constraint that a * b = c.""
    return make_ternary_constraint(a, b, c, mul, truediv, truediv)
```

```
def make_ternary_constraint(a, b, c, ab, ca, cb):
    '''The constraint that ab(a; b) = c and ca(c; a) = b and cb(c, b) = a.'''

def new_value():
    av, bv, cv = [connector[ 'has_val'](value) for connector in (a, b, c)]

from operator import add, sub, mul, truediv

def adder(a, b, c):
    '''The constraint that a + b = c.'''
    return make_ternary_constraint(a, b, c, add, sub, sub)

def multiplier(a, b, c):
    '''The constraint that a * b = c.'''
    return make_ternary_constraint(a, b, c, mul, truediv, truediv)

Generalizing to a Multiplication Constraint

Connectors

Relations

```python
def make_ternary_constraint(a, b, c, ab, ca, cb):
    """The constraint that ab(a; b) = c and ca(c; a) = b and cb(c, b) = a.""

def new_value():
    av, bv, cv = [connector[ 'has_val'](value) for connector in (a, b, c)]

from operator import add, sub, mul, truediv

def adder(a, b, c):
    """The constraint that a + b = c.""
    return make_ternary_constraint(a, b, c, add, sub, sub)

def multiplier(a, b, c):
    """The constraint that a * b = c.""
    return make_ternary_constraint(a, b, c, mul, truediv, truediv)
```
Generalizing to a Multiplication Constraint

```
def make_ternary_constraint(a, b, c, ab, ca, cb):
    """The constraint that ab(a,b)=c and ca(c,a)=b and cb(c,b)=a."""
    def new_value():
        av, bv, cv = [connector[ 'has_val']() for connector in (a, b, c)]
        if av and bv:
            c[ 'set_val'](constraint, ab(a[ 'val' ], b[ 'val' ]))

from operator import add, sub, mul, truediv

def adder(a, b, c):
    """The constraint that a + b = c."""
    return make_ternary_constraint(a, b, c, add, sub, sub)

def multiplier(a, b, c):
    """The constraint that a * b = c."""
    return make_ternary_constraint(a, b, c, mul, truediv, truediv)
```
Generalizing to a Multiplication Constraint

```python
# Constraint programming

def make_ternary_constraint(a, b, c):
    """The constraint that a * b = c.""
    return make_converter(c, f, a, b, c)

multiplier = make_ternary_constraint
```

```python
from operator import add, sub, mul, truediv

def adder(a, b, c):
    """The constraint that a + b = c.""
    return make_ternary_constraint(a, b, c, add, sub, sub)

def multiplier(a, b, c):
    """The constraint that a * b = c.""
    return make_ternary_constraint(a, b, c, mul, truediv, truediv)
```
Generalizing to a Multiplication Constraint

```python
def make_ternary_constraint(a, b, c, ab, ca, cb):
    """The constraint that ab(a,b)=c and ca(c,a)=b and cb(c,b)=a.""

def new_value():
    av, bv, cv = [connector[ 'has_val']() for connector in (a, b, c)]

    if av and bv:
        c[ 'set_val'](constraint, ab(a[ 'val'], b[ 'val']))

    elif av and cv:
        b[ 'set_val'](constraint, ac(c[ 'val'], a[ 'val']))

    elif bv and cv:
        a[ 'set_val'](constraint, cb(c[ 'val'], b[ 'val']))

from operator import add, sub, mul, truediv

def adder(a, b, c):
    """The constraint that a + b = c.""
    return make_ternary_constraint(a, b, c, add, sub, sub)

def multiplier(a, b, c):
    """The constraint that a * b = c.""
    return make_ternary_constraint(a, b, c, mul, truediv, truediv)
```
Implementing a Connector
Implementing a Connector

```python
def make_connector(name=None):
```

Tuesday, September 27, 2011
Implementing a Connector

def make_connector(name=None):

c = {
    'val': None,
    'set_val': set_value,
    'forget': forget_value,
    'has_val': lambda: connector['val'] is not None,
    'connect': lambda source: constraints.append(source)}
Implementing a Connector

def make_connector(name=None):

    connector = {'val': None,
                 'set_val': set_value,
                 'forget': forget_value,
                 'has_val': lambda: connector['val'] is not None,
                 'connect': lambda source: constraints.append(source)}

    return connector
Implementing a Connector

```python
def make_connector(name=None):
    informant = None
    constraints = []

    connector = {'val': None,
                 'set_val': set_value,
                 'forget': forget_value,
                 'has_val': lambda: connector['val'] is not None,
                 'connect': lambda source: constraints.append(source)}

    return connector
```

Explanation:
- The `make_connector` function creates a connector object that manages constraints.
- It initializes `informant` to `None` and `constraints` as an empty list.
- The `connector` dictionary contains methods like `set_value`, `forget_value`, and `connect` to interact with the constraints.
- The `return` statement returns the connector object.

This example demonstrates how to create a simple connector that can be used to manage constraints in a system.
Implementing a Connector

```python
def make_connector(name=None):
    informant = None
    constraints = []

def set_value(source, value):
    nonlocal informant
    val = connector['val']
    if val is None:
        informant, connector['val'] = source, value
    if name is not None:
        print(name, '=', value)
        inform_all_except(source, 'new_val', constraints)
    else:
        if val != value:
            print('Contradiction detected:', val, 'vs', value)

connector = {
    'val': None,
    'set_val': set_value,
    'forget': forget_value,
    'has_val': lambda: connector['val'] is not None,
    'connect': lambda source: constraints.append(source)
}

return connector
```

```python
>>> celsius = make_converter(celsius, fahrenheit)
>>> fahrenheit = make_connector(Fahrenheit = 77.0)
>>> celsius = make_connector(Fahrenheit = 77.0, Celsius = 25)
>>> make_converter(celsius, fahrenheit)
Fahrenheit is forgotten
Celsius is forgotten
Contradiction detected: 77.0 vs 212
```
Implementing a Connector

def make_connector(name=None):
    informant = None
    constraints = []

def set_value(source, value):
    nonlocal informant
    val = connector['val']
    if val is None:
        informant, connector['val'] = source, value
        if name is not None:
            print(name, ' val', value)
        inform_all_except(source, 'new_val', constraints)
    else:
        if val != value:
            print('Contradiction detected:', val, ' vs', value)

def forget_value(source):
    nonlocal informant
    if informant == source:
        informant, connector['val'] = None, None
        if name is not None:
            print(name, 'is forgotten')
        inform_all_except(source, 'forget', constraints)

connector = {'val': None,
             'set_val': set_value,
             'forget': forget_value,
             'has_val': lambda: connector['val'] is not None,
             'connect': lambda source: constraints.append(source)}

return connector