61A Lecture 13

Wednesday, September 28

Dictionaries

```
{'Dem': 0}
```


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

```
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?

Demo

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

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

```
a + b = c
a = c - b
b = c - a
```

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

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.

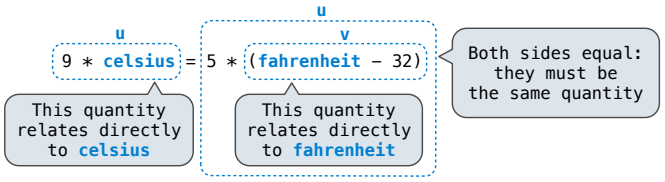
Constraint programming:

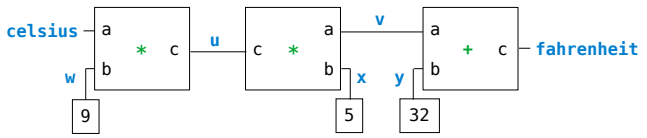
- We define the relationship between quantities
- We provide values for the "known" quantities
- The system computes values for the "unknown" quantities

Challenge: We want a general means of combination.

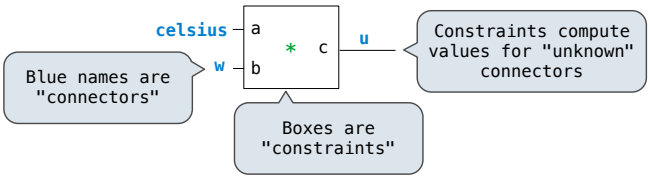
A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.





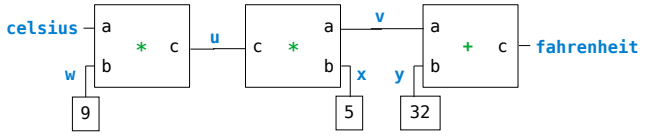
Anatomy of a Constraint



- *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

```
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

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.

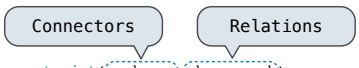
`connector['forget'](source)` tells the connector that the source is requesting it to forget its value.

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

Implementing an Adder Constraint

```
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




```
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

```
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)

    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
```