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

```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?

Demonstration:

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.

Demonstration:

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

\[ a + b = c \quad \text{and} \quad p \times v = n \times k \times t \]
\[ a = c - b \quad \text{and} \quad 9 \times c = 5 \times (f - 32) \]

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.

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

Both sides equal: they must be the same quantity

A Constraint Network for Temperature Conversion

Combination idea: All intermediate quantities have values too.

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

Both sides equal: they must be the same quantity

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)
    adder(v, y, fahrenheit)
    constant(w, 9)
    constant(x, 5)
    constant(y, 32)
    constraint = adder_constraint(a, b, c)
    constraint["user"] = 32
    constraint["value"] = [a, b, c]
    constraint["fahrenheit"] = multiplier_constraint(9, constraint, fahrenheit)
    constraint["celsius"] = adder_constraint(adder_constraint(9, constraint, fahrenheit), 32)
    constraint["has_value"] = True
    return constraint
```

Implementing an Adder Constraint

```python
def adder_constraint(a, b, c):
    # Add constraint that a + b = c.
    constraint = ("user", a + b)
    constraint["value"] = c
    return constraint
```

The Messages of a Connector

```python
connector = make_converter('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 a Multiplier Constraint

```python
def multiplier_constraint(u, constraint, w):
    # Implementing a multiplier constraint:
    # constraint = ("user", constraint["value"] * w)
    # constraint["value"] = constraint["new_value"]
    # constraint["has_value"] = True
    # constraint["balance"] = 0
    # constraint["withdraw"] = False
    # constraint["deposit"] = False
    constraint["user"] = w
    constraint["value"] = constraint["user"] * constraint["balance"]
    constraint["balance"] = constraint["user"]
    constraint["has_value"] = True
    return constraint
```
Generalizing to a Multiplication Constraint

```
def make_ternary_constraint(a, b, c, ab, ac, bc):
    """The constraint that a(b/c) + b and c(b/a) = a.""
    av, bv, cv = connector(a, b, c)
    ad = constraint(a, b, c)
    add, sub, mul, truediv = operator
    **c** = constraint(c, b, a)
    add ab, ac, bc:

    def new_value():
        av, bv, cv = connector():
        a: set_val()
        b: set_val()
        c: set_val()

    def make_converter(celsius, fahrenheit):
        celsius = make_connector()
        celsius: set_val()
        fahrenheit = make_connector()
        fahrenheit: set_val()

    def constant(y, x):
        return

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

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

Implementing a Connector

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

def set_value(source, value):
    nonlocal informant
    val = connector():
    if val is None;
        informant, connector: val = source, value
        if name is not None;
            print(name, 'set value:', val)
            inform_all_except(source, new_val, constraints)
    else:
        print()
        informant:
        if val == value:
            print('Contradiction detected:', val, 'eq', value)

    def forget_value(source):
        nonlocal informant
        informant = connector():
        informant, connector: val = None, None
        if name is 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
```