61A LECTURE 10 – MUTABLE DATA

Steven Tang and Eric Tzeng

July 10, 2013

Announcements

- Do the homework!
- Keep on studying for Midterm 1!

A Function with Evolving Behavior

Let's model a bank account that has a balance of $100

Return value: remaining balance

Argument: amount to withdraw

```
>>> withdraw(25)
75
```

Different return value!

Second withdrawal of the same amount

```
>>> withdraw(25)
50
```

Where's this balance stored?

```
>>> withdraw(60)
'Insufficient funds'
```

```
>>> withdraw(15)
35
```

```
>>> withdraw = make_withdraw(100)
```

Within the function!

First attempts

```
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."""
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

Local variable ‘balance’ referenced before assignment…

Python particulars

Python pre-computes which frame contains each name before executing the body of a function.

Therefore, within the body of a function, all instances of a name must refer to the same frame.

```
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw

wd = make_withdraw(20)
wd(5)
```

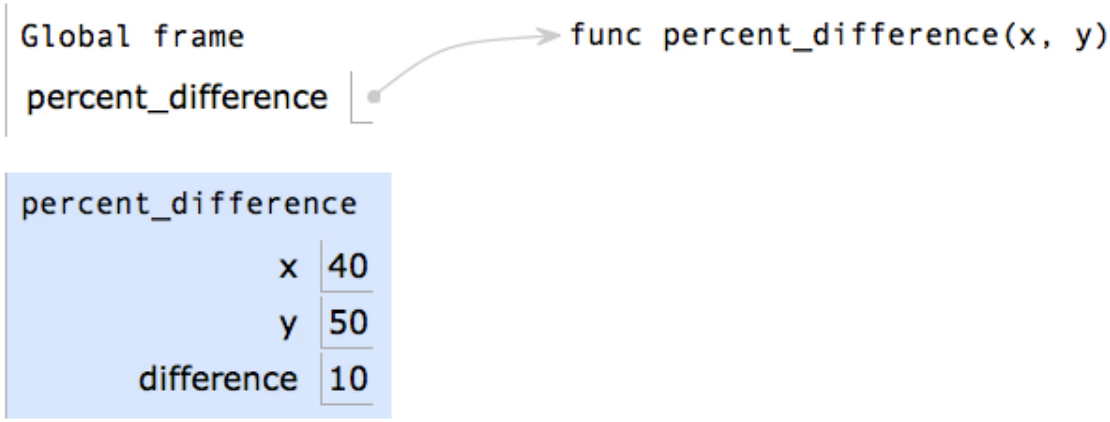
Local assignment

UnboundLocalError: local variable 'balance' referenced before assignment

Reminder: local assignment

```
def percent_difference(x, y):
        difference = abs(x-y)
        return 100 * difference / x
diff = percent_difference(40, 50)
```

Assignment binds name(s) to value(s) in the first frame of the current environment



Execution rule for assignment statements:

1. Evaluate all expressions right of =, from left to right.
2. Bind the names on the left the resulting values in the first frame of the current environment.

The effect of nonlocal statements

```
nonlocal <name>, <name 2>, ...
```

Effect: Future assignments to that name change its pre-existing binding in the **first non-local frame** of the current environment in which that name is bound.

Python Docs: an "enclosing scope"

From the Python 3 language reference:

Names listed in a nonlocal statement must refer to pre-existing bindings in an enclosing scope. Names listed in a nonlocal statement must not collide with pre-existing bindings in the local scope.

http://docs.python.org/release/3.1.3/reference/simple_stmts.html#the-nonlocal-statement

http://www.python.org/dev/peps/pep-3104/

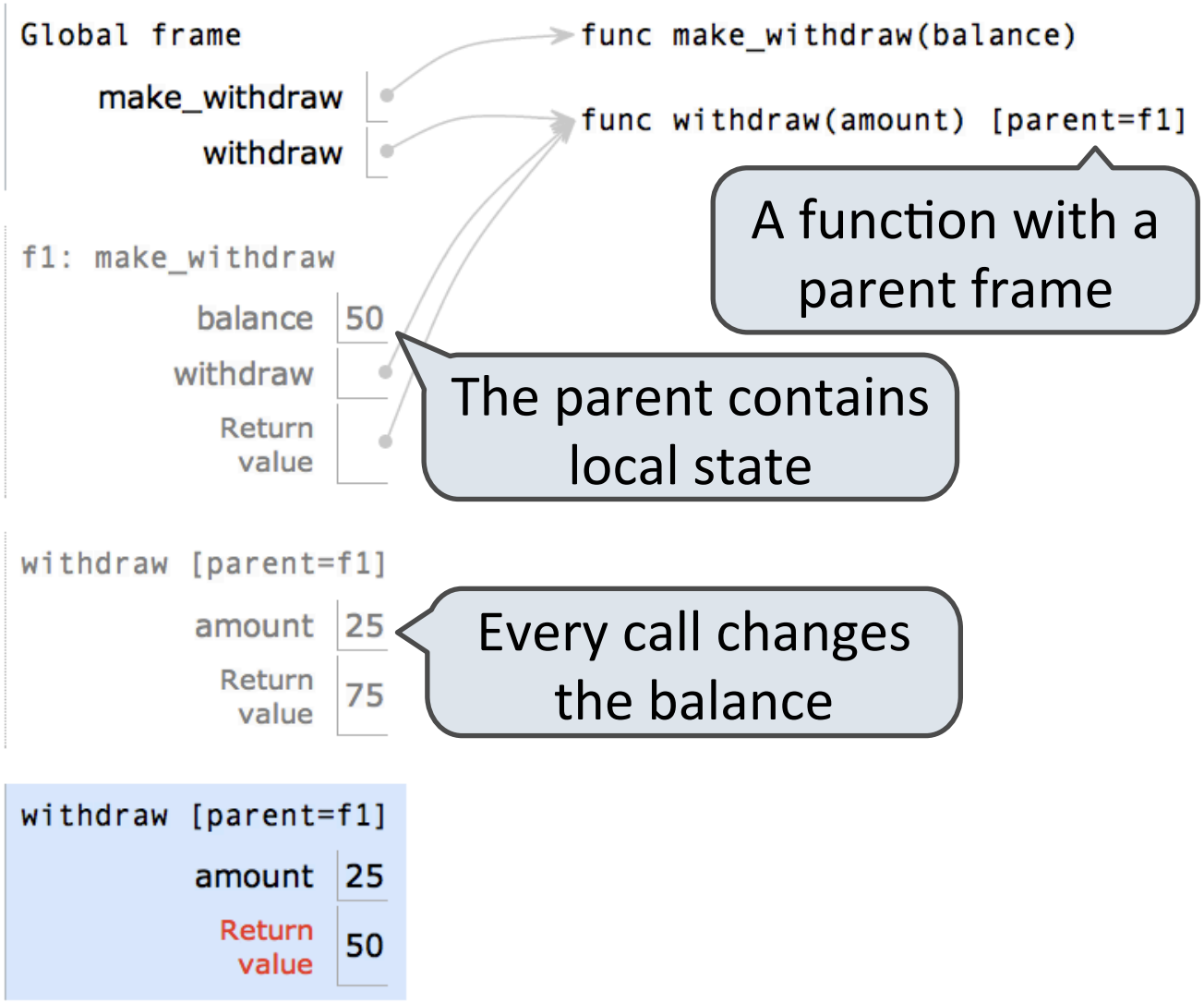
Non-Local Assignment

```
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."""
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

Declare the name "balance" nonlocal

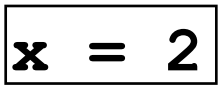
Re-bind balance where it was bound previously

Persistent Local State



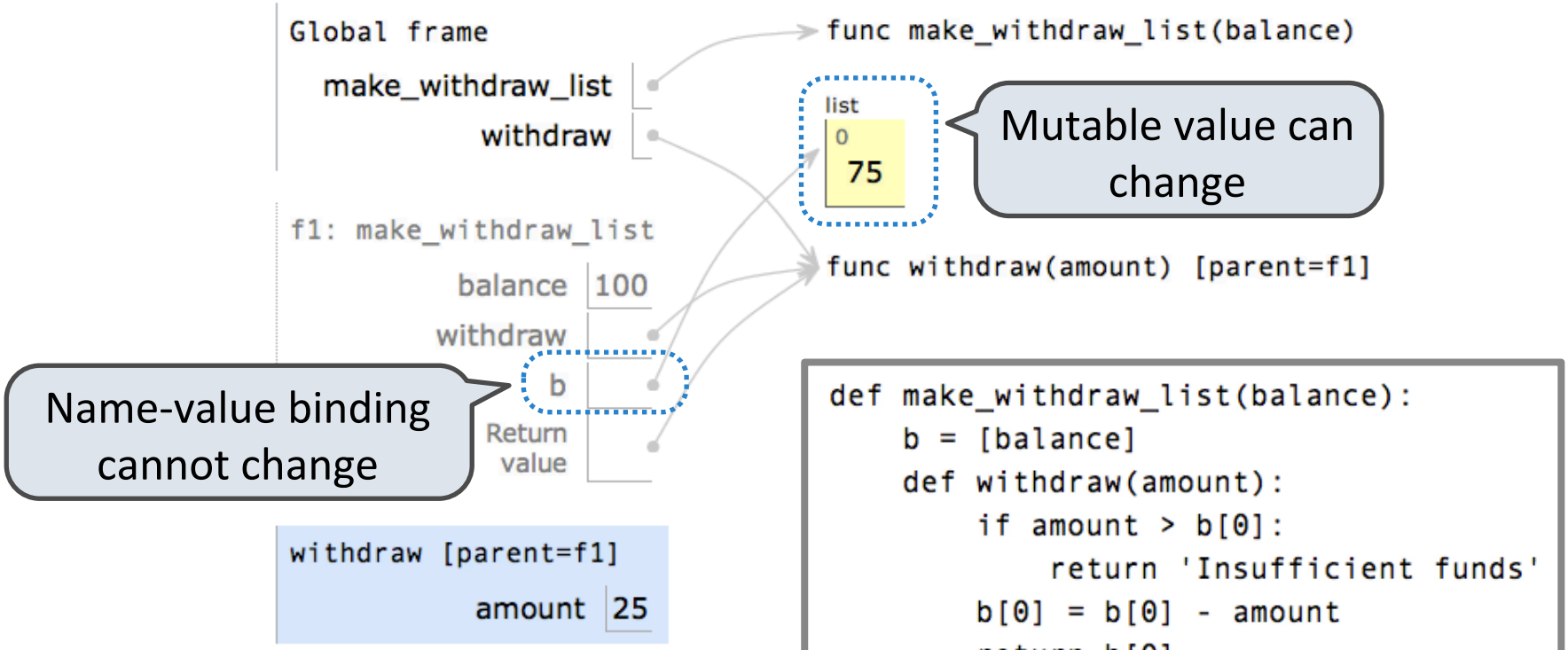
Effects of Assignment Statements

Status	Effect
• No nonlocal statement • "x" is not bound locally	Create a new binding from name "x" to object 2 in the first frame of the current environment.
• No nonlocal statement • "x" is bound locally	Re-bind name "x" to object 2 in the first frame of the current env.
• nonlocal x • "x" is bound in a non-local frame	Re-bind "x" to 2 in the first non-local frame of the current environment in which it is bound.
• nonlocal x • "x" is not bound in a non-local frame	SyntaxError: no binding for nonlocal 'x' found
• nonlocal x • "x" is bound in a non-local frame • "x" also bound locally	SyntaxError: name 'x' is parameter and nonlocal



Mutable Values and Persistent State

Mutable values can be changed without a nonlocal statement.

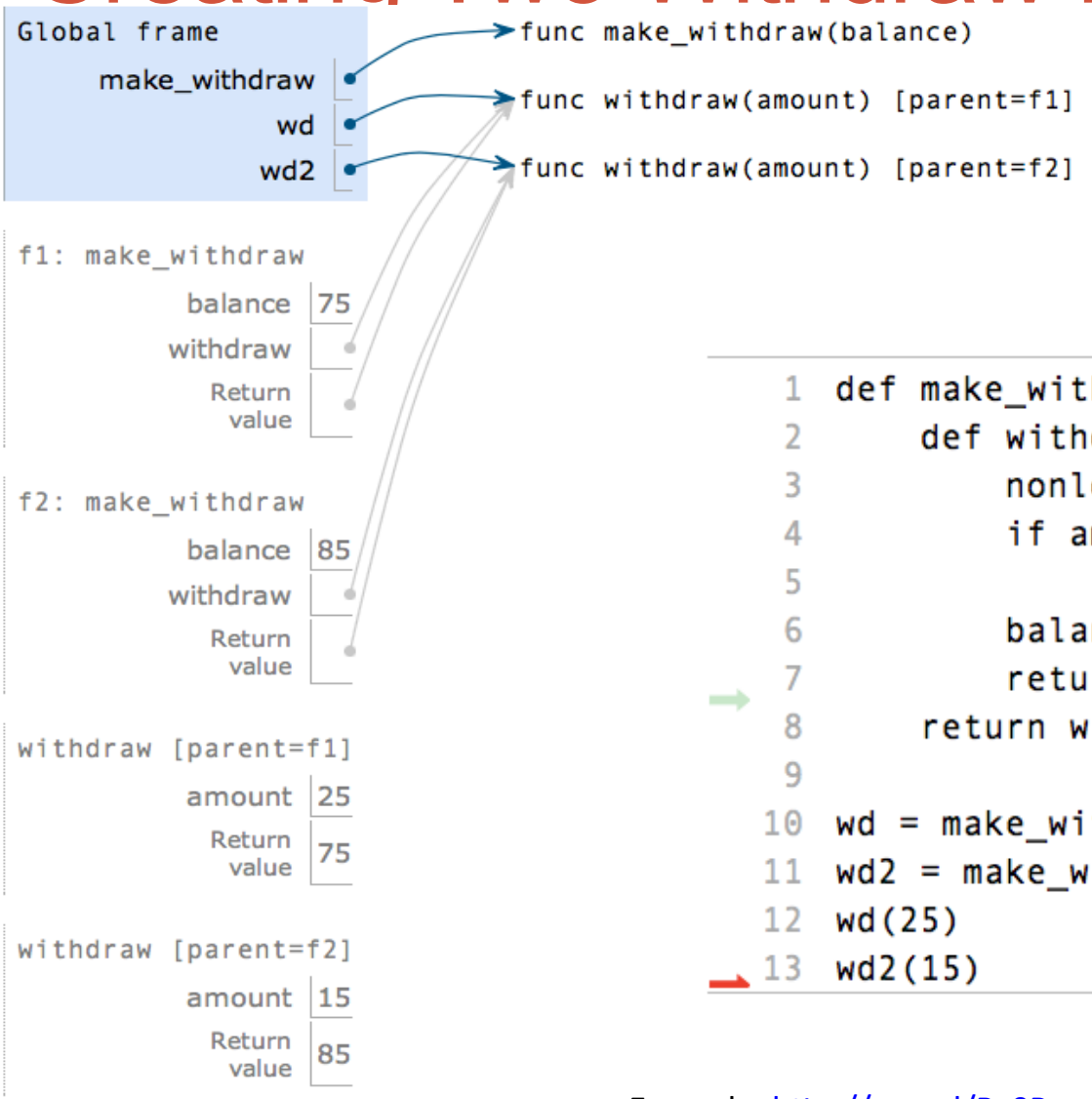



```
def make_withdraw_list(balance):
    b = [balance]
    def withdraw(amount):
        if amount > b[0]:
            return 'Insufficient funds'
        b[0] = b[0] - amount
        return b[0]
    return withdraw

withdraw = make_withdraw_list(100)
withdraw(25)
```

Example: http://goo.gl/kJAiF

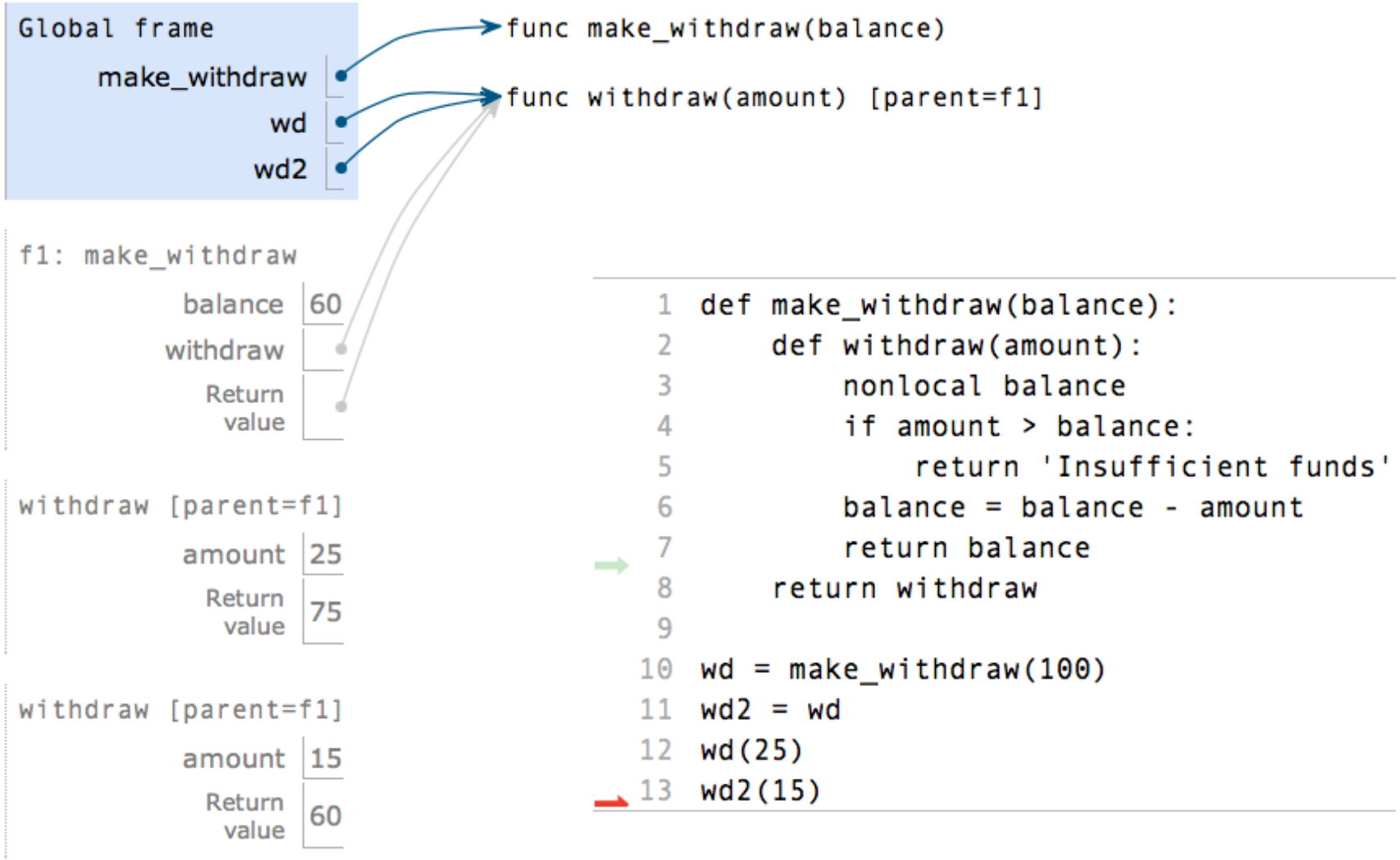
Creating Two Withdraw Functions




```
1  def make_withdraw(balance):
2      def withdraw(amount):
3          nonlocal balance
4          if amount > balance:
5              return 'Insufficient funds'
6          balance = balance - amount
7          return balance
8      return withdraw
9
10 wd = make_withdraw(100)
11 wd2 = make_withdraw(100)
12 wd(25)
13 wd2(15)
```

Example: http://goo.gl/Bc0Rc

Multiple References to a Withdraw Function



Example: http://goo.gl/VEL0P

The Benefits of Non-Local Assignment

- Ability to maintain some state that is local to a function, but evolves over successive calls to that function.
- The binding for balance in the first non-local frame of the environment associated with an instance of withdraw is inaccessible to the rest of the program.
- An abstraction of a bank account that manages its own internal state.

Break!

What have we accomplished

- We've created a form of data that can:
  - Keep track of a changing state (the account balance)
  - Perform actions based on that state (withdraw money, or complain about insufficient funds)
- Rest of lectures is variations on this theme
- This is exciting! Allows us to solve more interesting problems
- But we lost something in the process…

Referential transparency

Expressions are referentially transparent if substituting an expression with its value does not change the meaning of a program.

```
mul(add(2, mul(4, 6)), 3)

mul(add(2,    24    ), 3)

mul(      26          , 3)
```

Mutation is a *side effect* (like printing)

Side effects violate the condition of referential transparency because they do more than just return a value; they change the state of the computer.

A Mutable Container

```
def container(contents):
    """Return a container that is manipulated by two
    functions.

    >>> get, put = container('hello')
    >>> get()
    'hello'
    >>> put('world')
    >>> get()
    'world'
    """

    def get():
        return contents

    def put(value):
        nonlocal contents
        contents = value

    return put, get
```

Two separate functions to manage! Can we make this easier?

Dispatch Functions

A technique for packing multiple behaviors into one function

```
def pair(x, y):
    """Return a function that behaves like a pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

Message argument can be anything, but strings are most common

The body of a dispatch function is always the same:

- One conditional statement with several clauses
- Headers perform equality tests on the message

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

Important historical role:
The message passing approach strongly influenced object-oriented programming
(next lecture)

Mutable Container with Message Passing

```
def container_dispatch(contents):
    def dispatch(message,
                 value=None):
        nonlocal contents
        if message == 'get':
            return contents
        if message == 'put':
            contents = value

    return dispatch
```

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

Mutable Recursive Lists

```
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push':
            contents = make_rlist(value, contents)
        elif message == 'pop':
            item = first(contents)
            contents = rest(contents)
            return item
        elif message == 'str':
            return str_rlist(contents)
    return dispatch
```

Building Dictionaries with Lists

Now that we have lists, we can use them to build dictionaries

We store key-value pairs as 2-element lists inside another list

```
records = [['cain',       2.79],
           ['bumgarner',  3.37],
           ['vogelsong',  3.37],
           ['lincecum',   5.18],
           ['zito',       4.15]]
```

Dictionary operations:

- **`getitem(key)`** : Look at each record until we find a stored key that matches **`key`**
- **`setitem(key, value)`** : Check if there is a record with the given key. If so, change the stored value to **`value`**. If not, add a new record that stores **`key`** and **`value`**.

Implementing Dictionaries

```
def dictionary():
    """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?

This huge if-clause is still rather unsightly! Can we do better?

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.

An Account as a Dispatch Dictionary

```
def account(balance):
    """Return an account that is represented as a
    dispatch dictionary."""

    def withdraw(amount):
        if amount > dispatch['balance']:
            return 'Insufficient funds'
        dispatch['balance'] -= amount
        return dispatch['balance']

    def deposit(amount):
        dispatch['balance'] += amount
        return dispatch['balance']

    dispatch = {'balance': balance, 'withdraw': withdraw,
                'deposit': deposit}

    return dispatch
```

Question: Why `dispatch['balance']` and not `balance`?

The Story So Far About Data

Data abstraction: Enforce a separation between how data values are represented and how they are used.

Abstract data types: A representation of a data type is valid if it satisfies certain behavior conditions.

Message passing: We can organize large programs by building components that relate to each other by passing messages.

Dispatch functions/dictionaries: A single object can include many different (but related) behaviors that all manipulate the same local state.

(All of these techniques can be implemented using only functions and assignment.)