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
Second withdrawal of the same amount
Where’s this balance stored?
Within the function!

First attempts
```python
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.

Reminder: local assignment
```python
def percent_difference(x, y):
    difference = x - y
    return 100 * difference / x

g = 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.

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

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.

**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

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

```
def make_withdraw_list(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        return balance
    return withdraw
```

Persistent Local State

A function with a parent frame

The parent contains local state

Every call changes the balance

The effects of assignment statements:

Status | Effect
--- | ---

• No nonlocal statement | Create a new binding from name "x" to object 2 in the first frame of the current environment.

• "x" is bound locally | Re-bind name "x" to object 2 in the first frame of the current env.

• nonlocal x | Re-bind "x" to 2 in the first non-local frame of the current environment in which it is bound.

• nonlocal x | SyntaxError: no binding for nonlocal 'x' found

• "x" is not bound in a non-local frame | SyntaxError: name 'x' is parameter and nonlocal

• "x" is bound in a non-local frame | "x" also bound locally

Mutable Values and Persistent State

Mutable values can be changed without a nonlocal statement.

```
def make_withdraw_list(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        return balance
    return withdraw
```

Example: http://goo.gl/10tJ

Example: http://goo.gl/kJAiF

Creating Two Withdraw Functions

```
def make_withdraw(balance):
    def withdraw(amount):
        return balance
    return withdraw
```

Example: http://goo.gl/k10t

Example: http://goo.gl/k10t

Example: http://goo.gl/k10t
Multiple References to a Withdraw Function

Example: [http://goo.gl/VEL0P](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.

<table>
<thead>
<tr>
<th>Weasley Account</th>
<th>Potter Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10</td>
<td>$1,000,000</td>
</tr>
</tbody>
</table>

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.

\[
\text{mul}(\text{add}(2, \text{mul}(4, 6)), 3) \\
\text{mul}(\text{add}(2, 24), 3) \\
\text{mul}(26, 3)
\]

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

```python
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
```

Two separate functions to manage! Can we make this easier?
Dispatch Functions
A technique for packing multiple behaviors into one function

```python
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

Mutable Container with Message Passing

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

Mutable Recursive Lists

```python
def mutable_rlist():
    contents = empty_rlist
    def dispatch(message, value=None):
        if message == 'lam':
            return lam_rlist(contents)
        elif message == 'getitem_rlist':
            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

```python
records = [['cain', 2.79], ['bumparnes', 3.37], ['vogelsong', 3.37], ['lincescum', 5.18], ['sito', 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

```python
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
        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

```python
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
```

Quesion: 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.)