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

Return value: remaining balance

Argument: amount to withdraw

Different return value!

Second withdrawal of the same amount

Where's this balance stored?

Within the parent frame of the function!

A function has a body and a parent environment
Persistent Local State Using Environments

The parent frame contains the balance, the local state of the withdraw function

All calls to the same function have the same parent

Every call decreases the same balance by (a possibly different) amount
Reminder: Local Assignment

Execution rule for assignment statements:

1. Evaluate all expressions right of =, from left to right

2. Bind the names on the left to the resulting values in the current frame
Non-Local Assignment & Persistent Local State

```python
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

(Demo)
```
Non-Local Assignment
The Effect of Nonlocal Statements

```python
nonlocal <name>, <name>, ...
```

**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

http://www.python.org/dev/peps/pep-3104/
## The Many Meanings of Assignment Statements

<table>
<thead>
<tr>
<th>Status</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>• No nonlocal statement <strong>&quot;x&quot; is not bound locally</strong></td>
<td><strong>x = 2</strong> Create a new binding from name &quot;x&quot; to object 2 in the first frame of the current environment</td>
</tr>
<tr>
<td>• No nonlocal statement <strong>&quot;x&quot; is bound locally</strong></td>
<td>Re-bind name &quot;x&quot; to object 2 in the first frame of the current environment</td>
</tr>
<tr>
<td>• nonlocal x <strong>&quot;x&quot; is bound in a non-local frame</strong></td>
<td>Re-bind &quot;x&quot; to 2 in the first non-local frame of the current environment in which &quot;x&quot; is bound</td>
</tr>
<tr>
<td>• nonlocal x <strong>&quot;x&quot; is not bound in a non-local frame</strong></td>
<td>SyntaxError: no binding for nonlocal 'x' found</td>
</tr>
<tr>
<td>• nonlocal x <strong>&quot;x&quot; is bound in a non-local frame</strong></td>
<td>SyntaxError: name 'x' is parameter and nonlocal</td>
</tr>
<tr>
<td>• nonlocal x <strong>&quot;x&quot; also bound locally</strong></td>
<td></td>
</tr>
</tbody>
</table>
Python Particulars

Python pre-computes which frame contains each name before executing the body of a function. Within the body of a function, all instances of a name must refer to the same frame.

```python
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)
```

UnboundLocalError: local variable 'balance' referenced before assignment
Mutable Values & Persistent Local State

Mutable values can be changed *without* a nonlocal statement.

goo.gl/y4TyFZ
Multiple Mutable Functions

(Demo)
Referential Transparency, Lost

• 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)), add(3, 5))
mul(add(2, 24), add(3, 5))
mul(26, add(3, 5))
```

• Mutation operations violate the condition of referential transparency because they do more than just return a value; they change the environment.
Environment Diagrams
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)

oski(abs)