A Function with Behavior That Varies Over Time

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

Argument: amount to withdraw

Return value: remaining balance

>>> withdraw(25)
75

Different return value!

>>> withdraw(25)
50

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35

>>> withdraw = make_withdraw(100)

Within the function!

Persistent Local State

Non-Local Assignment & Persistent Local State

The Effect of Nonlocal Statements

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.

Example: http://goo.gl/wcF71

Assignment binds names to values in the current local frame

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

Effect: Future references to that name refer to 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#nonlocal-statement
http://docs.python.org/3/library/enclosing-scope
The Many Meanings of Assignment Statements

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

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.

```python
def make_withdraw(balance):  
    def withdraw(amount):  
        if amount > balance:  
            return 'Insufficient funds'  
        return balance - amount  
    return withdraw
wd = make_withdraw(20)  
wd(5)
```

Mutable Values & Persistent Local State

Mutable values can be changed without a nonlocal statement.

Creating Two Different Withdraw Functions

The Benefit 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.

Multiple References to a Single Withdraw Function

Demo

<table>
<thead>
<tr>
<th>John's Account</th>
<th>Steven's Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10</td>
<td>$1,000,000</td>
</tr>
</tbody>
</table>
Sameness and Change

- As long as we never modify objects, we can regard a compound object to be precisely the totality of its pieces.
- A rational number is just its numerator and denominator.
- This view is no longer valid in the presence of change.
- Now, a compound data object has an “identity” that is something more than the pieces of which it is composed.
- A bank account is still “the same” bank account even if we change the balance by making a withdrawal.
- Conversely, we could have two bank accounts that happen to have the same balance, but are different objects.

<table>
<thead>
<tr>
<th>John’s Account</th>
<th>Steven’s Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10</td>
<td>$10</td>
</tr>
</tbody>
</table>

Referential Transparency, Lost

- Expressions are referentially transparent if substituting an expression with its value does not change the meaning of a program.
- Re-binding operations violate the condition of referential transparency because they let us define functions that do more than just return a value; we can change the environment, causing values to mutate.

```
mul(add(2, mul(4, 6)), add(3, 5))
```

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