A Function with Behavior That Varies Over Time

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

Return value: remaining balance

>>> withdraw(25)
75

Argument: amount to withdraw

>>> withdraw(25)
50

Second withdrawal of the same amount

>>> withdraw(60)
'Insufficient funds'

Different return value!

>>> withdraw(15)
35

Where's this balance stored?

>>> withdraw = make_withdraw(100)

Within the function!
Persistent Local State

```python
make_withdraw: withdraw:

balance: 100
withdraw:

make_withdraw

make_withdraw(balance):

function body to be revealed momentarily

withdraw(amount):

function body to be revealed momentarily
```
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
Local, Non-Local, and Global Frames

- First frame
- First non-local frame
- Second frame
- Second (and last) non-local frame
- Non-local frame
- Non-local frame
- Local frame
- The global frame
- An environment
The Effect of Nonlocal Statements

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

**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#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</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 not 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>• No nonlocal statement</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>
<tr>
<td>• nonlocal x</td>
<td></td>
</tr>
<tr>
<td>• &quot;x&quot; is bound locally</td>
<td></td>
</tr>
<tr>
<td>• nonlocal x</td>
<td></td>
</tr>
<tr>
<td>• &quot;x&quot; is bound in a non-local frame</td>
<td></td>
</tr>
</tbody>
</table>
Assignment Review: Teenage Mutant Ninja Turtles

def mutant(y):
    y, x = y+1, y+2
    return ninja(y)/2

def ninja(x):
    return x + 2

def turtle(x):
    return x * y + 2

y, ninja = 5, turtle
mutant(y)
Assignment Review: Teenage Mutant Ninja Turtles

• Bind mutant, ninja, and turtle to their respective functions
• Simultaneously: bind y to 5 and ninja to the turtle function
• Apply the mutant function to 5
  • In the first frame, bind y to 6 and x to 7
  • Look up ninja, which is bound to the turtle function
  • Look up y, which is bound to 6
  • Apply the turtle function to 6
    • Look up x, which is bound to 6 in the local frame
    • Look up y, which is bound to 5 in the global frame
    • Return 6 * 5 + 2 = 32
    • Return 32 / 2 = 16.0

```python
def mutant(y):
    y, x = y+1, y+2
    return ninja(y)/2

def ninja(x):
    return x + 2

def turtle(x):
    return x * y + 2

y, ninja = 5, turtle
mutant(y)
```
make_withdraw:

wd:

withdraw:

balance:

amount:

if amount > balance:
    return 'Insufficient funds'

balance = balance - amount
return balance
Calling a Withdraw Function Twice

```
wd = make_withdraw(20)
wd(5)
wd(3)
```

```
amount: 5
withdraw
```

```
balance: 12
withdraw:
```

```
amount: 3
withdraw
```

```
nonlocal balance
if amount > balance:
    return 'Insufficient funds'
balance = balance - amount
return balance
```

```
w = make_withdraw(20)
w(5)
w(3)
```
Creating Two Different Withdraw Functions

```python
def withdraw(amount):
    # ... (omitted code)
    return withdraw

wd = make_withdraw(20)
wd(5)
wd(3)
wd2 = make_withdraw(7)
wd2(6)
```

Friday, September 23, 2011
Creating Two Different Withdraw Functions

```python
wd = make_withdraw(20)
wd(5)
wd(3)
wd2 = make_withdraw(7)
wd2(6)

nonlocal balance
if amount > balance:
    return 'Insufficient funds'
balance = balance - amount
return balance
```

Friday, September 23, 2011
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**.

<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>
Multiple References to a Single Withdraw Function

```python
wd = make_withdraw(12)
w2d = wd
w2d(1)
wd(1)

balance = 11
wd: withdraw: 11
非局部平衡
if amount > balance:
    return 'Insufficient funds'
balance = balance - amount
return balance
```

Friday, September 23, 2011
Multiple References to a Single Withdraw Function

```python
withdraw(amount):
    make_withdraw:
        wd:
        wd2:

amount: 1
withdraw

balance: 10
withdraw:

amount: 1
withdraw

nonlocal balance
if amount > balance:
    return 'Insufficient funds'
balance = balance - amount
return balance

wd = make_withdraw(12)
wdf2 = wd
wdf2(1)
wdf(1)
```
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

• An expression is **referentially transparent** if its value does not change when we substitute one of its subexpression with the value of that subexpression.

\[
\text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5))
\]

\[
\text{mul}(\text{add}(2, 24), \text{add}(3, 5))
\]

\[
\text{mul}(26, \text{add}(3, 5))
\]

• Re-binding operations violate the condition of referential transparency because they do more than return a value; **they change the environment**.

• Two separately defined functions are not the same, because changes to one may not be reflected in the other.