

61A Lecture 11

Friday, September 23

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

Different
return value!

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

Second withdrawal
of the same amount

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

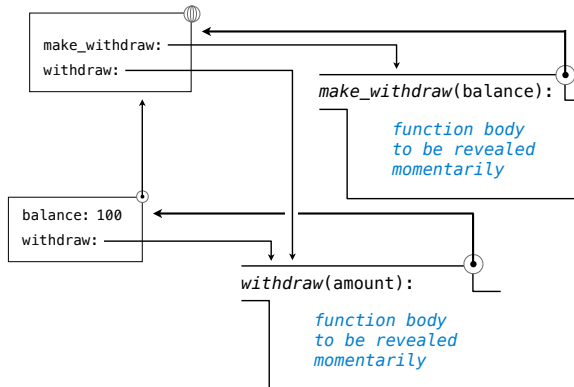
```
>>> withdraw(15)  
35
```

Where's this
balance stored?

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

Within the
function!

Persistent Local State



Local State via Non-Local Assignment

```
def make_withdraw(balance):
```

```
    """Return a withdraw function with a starting balance."""
```

```
    def withdraw(amount):
```

```
        nonlocal balance
```

Declare the name
"balance" nonlocal

```
        if amount > balance:
```

```
            return 'Insufficient funds'
```

```
        balance = balance - amount
```

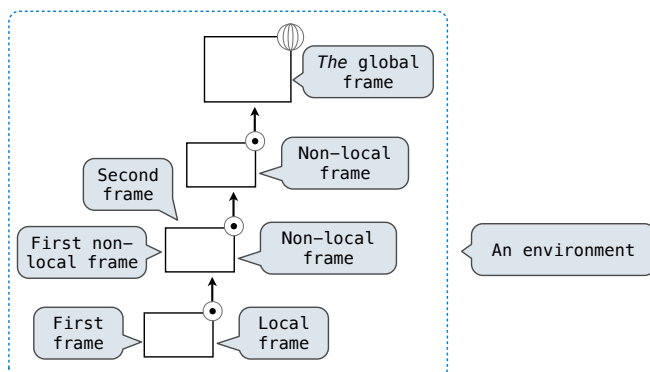
Re-bind the
existing balance
name above

```
        return balance
```

```
    return withdraw
```

Demo

Local, Non-Local, and Global Frames



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.

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/>

The Many Meanings of Assignment Statements

Status

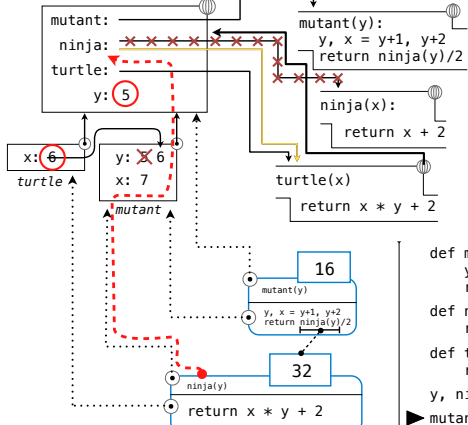
- No nonlocal statement
- "x" is **not** bound locally
- No nonlocal statement
- "x" is **bound** locally
- nonlocal x
- "x" is **bound** locally
- nonlocal x
- "x" is **not** bound in a non-local frame
- nonlocal x
- "x" is **bound** in a non-local frame

x = 2

Effect

- Create a new binding from name "x" to object 2 in the first frame of the current environment.
- Re-bind name "x" to object 2 in the first frame of the current env.
- SyntaxError: name 'x' is parameter and nonlocal
- SyntaxError: no binding for nonlocal 'x' found
- Re-binds "x" to 2 in the first non-local frame of the current environment in which that name is already bound.

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 32
- Return half the result: 16

Intrinsic function name

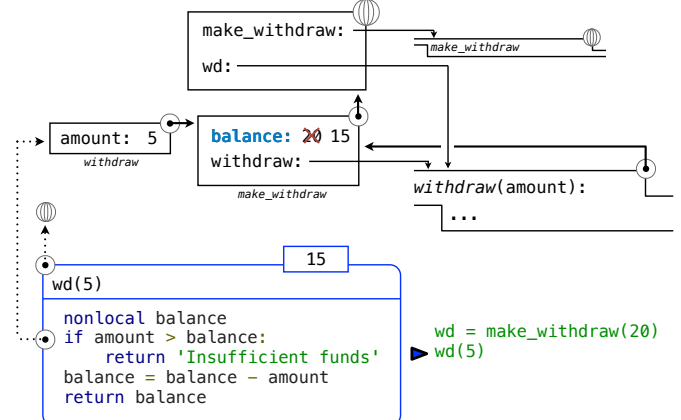
```
def mutant(y):
    y, x = y+1, y+2
    return ninja(y)/2

def ninja(x):
    return x + 2

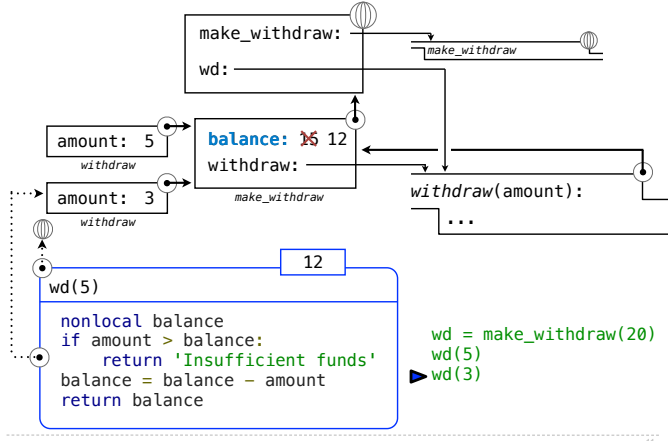
def turtle(x):
    return x * y + 2

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

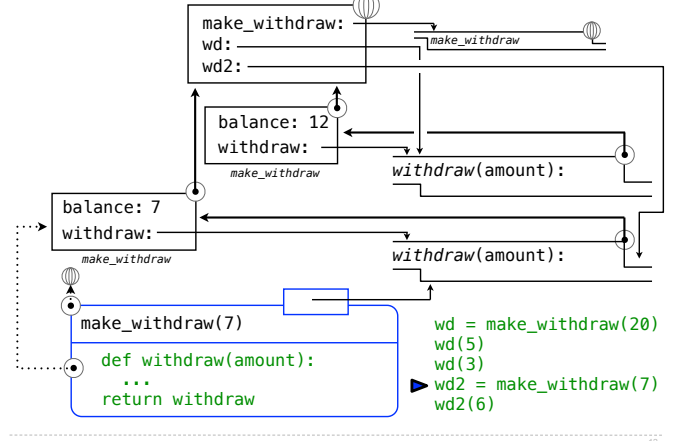
Environment Diagram of Withdraw



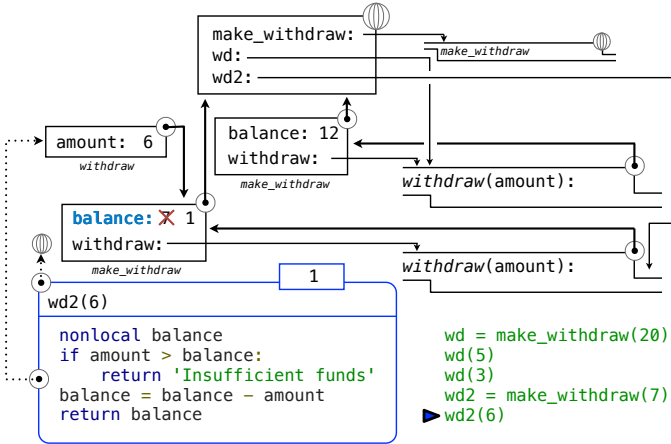
Calling a Withdraw Function Twice



Creating Two Different Withdraw Functions



Creating Two Different Withdraw Functions



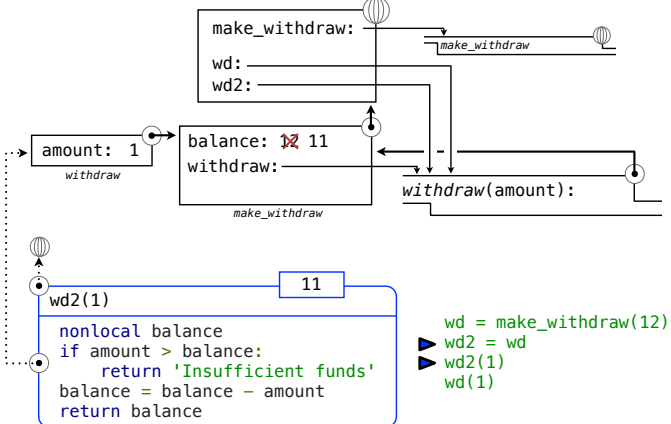
The Benefit of Non-Local Assignment

- 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**.

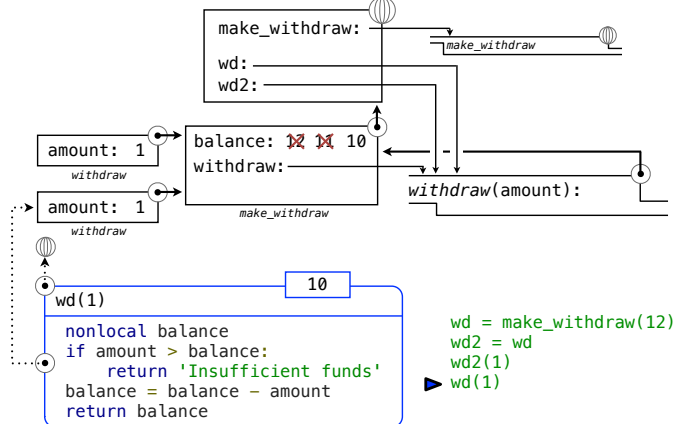
| |
|----------------|
| John's Account |
| \$10 |

| |
|------------------|
| Steven's Account |
| \$1,000,000 |

Multiple References to a Single Withdraw Function



Multiple References to a Single Withdraw Function



Sameness and Change

- So long as we **never modify** data objects, we can regard a compound object to be precisely the **totality of its pieces**.
- A **rational number** is determined by 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**.

| |
|----------------|
| John's Account |
| \$10 |

| |
|------------------|
| Steven's Account |
| \$10 |

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.

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

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

`mul(26, 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**.