

## Mutable Functions

---

## Announcements

## Mutable Functions

## 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
```

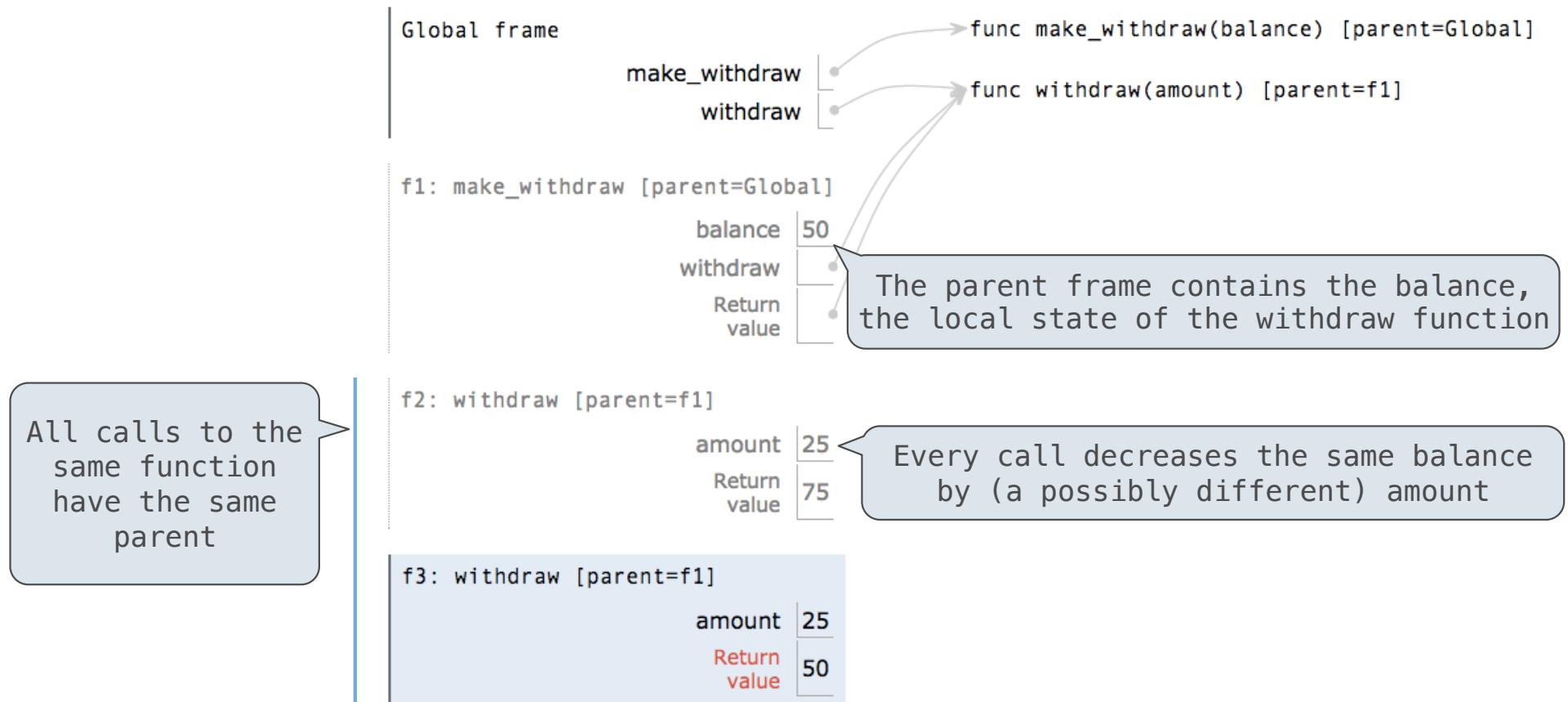
Where's this balance  
stored?

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

Within the parent frame  
of the function!

A function has a body and  
a parent environment

## Persistent Local State Using Environments



## Reminder: Local Assignment

```
def percent_difference(x, y):
    difference = abs(x-y)
    return 100 * difference / x
diff = percent_difference(40, 50)
```

Assignment binds name(s) to value(s) in the first frame of the current environment

Global frame

percent\_difference

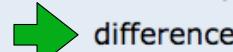
func percent\_difference(x, y) [parent=Global]

f1: percent\_difference [parent=Global]

x 40

y 50

difference 10



### 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

```
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
```

nonlocal balance      Declare the name "balance" nonlocal at the top of the body of the function in which it is re-assigned

if amount > balance:      balance assigned in make\_withdraw frame instead of withdraw frame

balance = balance - amount      Re-bind balance in the first non-local frame in which it was bound previously

(Demo)

## Non-Local Assignment

## The Effect of Nonlocal Statements

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.

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**.

Current frame

[http://docs.python.org/release/3.1.3/reference/simple\\_stmts.html#the-nonlocal-statement](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

x = 2

### Status

### Effect

- No nonlocal statement
- "x" **is not** bound locally

Create a new binding from name "x" to object 2 in the first frame of the current environment

- 
- No nonlocal statement
  - "x" **is** bound locally

Re-bind name "x" to object 2 in the first frame of the current environment

- 
- nonlocal x
  - "x" **is** bound in a non-local frame

Re-bind "x" to 2 in the first non-local frame of the current environment in which "x" is bound

- 
- nonlocal x
  - "x" **is not** bound in a non-local frame

SyntaxError: no binding for nonlocal 'x' found

- 
- nonlocal x
  - "x" **is** bound in a non-local frame
  - "x" also bound locally

SyntaxError: name 'x' is parameter and nonlocal

# 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.

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

## Local assignment

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

**UnboundLocalError:** local variable 'balance' referenced before assignment

## Python Particulars

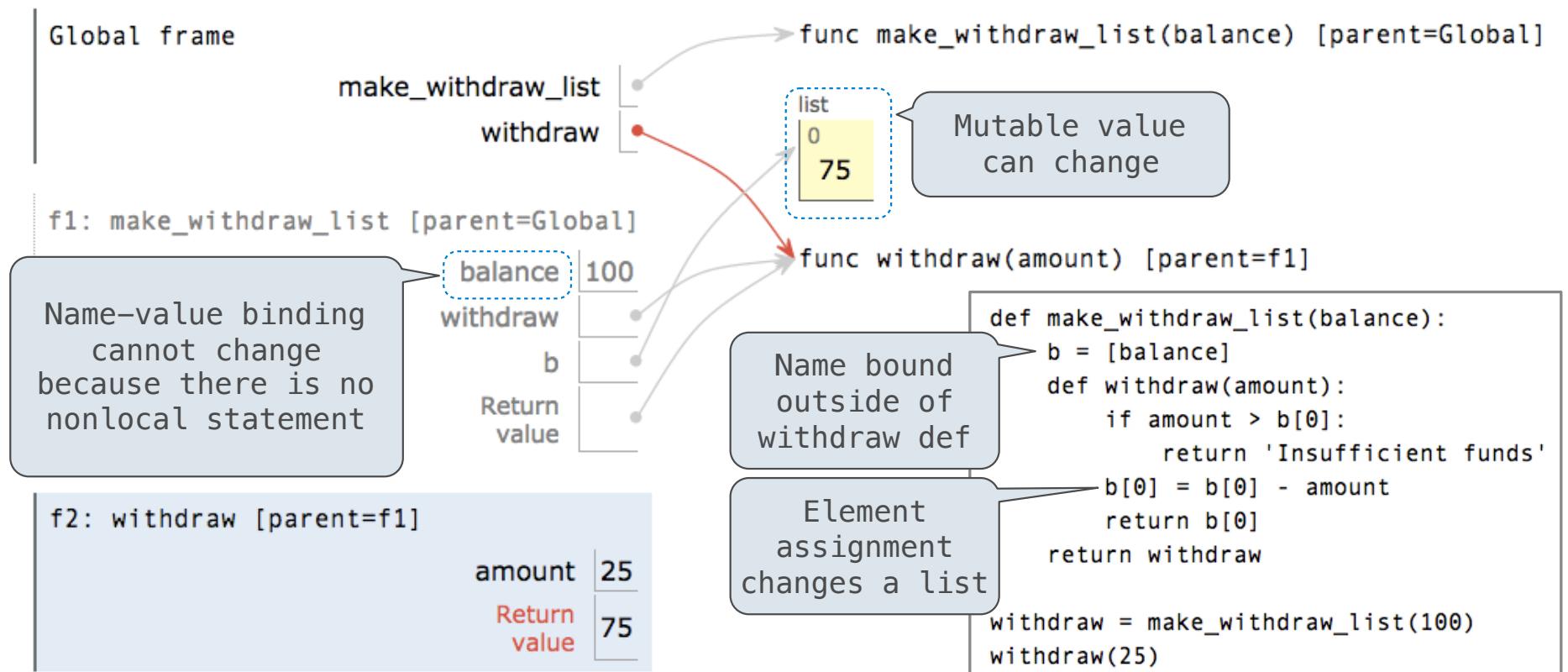


```
MacVim  File  Edit  Tools  Syntax  Buffers  Window  Help
lec — Python — 52x28
~/lec$ python3 -i ex.py
>>> w = make_withdraw(100)
>>> w
<function make_withdraw.<locals>.withdraw at 0x10283e378>
>>> w(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "ex.py", line 4, in withdraw
      if amount > balance:
UnboundLocalError: local variable 'balance' referenced before assignment
>>> ^D
~/lec$ python3 -i ex.py
>>> w = make_withdraw(100)
>>> w(10)
100
>>> w(10)
100
```

```
def make_withdraw(balance):
    def withdraw(amount):
        # nonlocal balance
        line 4 if amount > balance:
            return 'Insufficient funds'
        line 6 # balance = balance - amount
            return balance
        return withdraw
    ~
    ~
    ~
    ~ Note:
    ~ Though python interpreter says error
    ~ in line 4, it is the assignment in line 6 that
    ~ goes wrong: non-local lookup (line 4) and
    ~ local assignment (line 6) should not exist
    ~ at the same time.
After removing line 6, the function will have
no error; it just doesn't work properly.
```

## Mutable Values & Persistent Local State

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



## 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.**

Chrome File Edit View History Bookmarks Window Help

Online Python Tutor - Visu X

pythontutor.com/composingprograms.html#mode=display

The screenshot shows a browser window for the Online Python Tutor. On the left, a code editor displays the following Python code:

```
1 def f(x):
2     x = 4
3     def g(y):
4         def h(z):
5             nonlocal x
6             x = x + 1
7             return x + y + z
8         return h
9     return g
10 a = f(1)
11 b = a(2)
12 total = b(3) + b(4)
```

The line `12 total = b(3) + b(4)` is highlighted with a red arrow, indicating it is the next line to execute. Below the code editor are navigation buttons: << First, < Back, Program terminated, Forward >, and Last >>. A legend at the bottom left indicates: a green arrow for the line just executed, and a red arrow for the next line to execute.

On the right, a call stack visualization shows the state of three frames:

- Global frame:** Contains `f`, `a`, `b`, and `total 22`. Arrows point from `f`, `a`, and `b` to their respective child frames.
- f1: f [parent=Global]:** Contains `x 6`, `g`, and `Return value`.
- f2: g [parent=f1]:** Contains `y 2`, `h`, and `Return value`.
- f3: h [parent=f2]:** Contains `z 3`, `Return value 10`, and a partially visible frame below it containing `z 4` and `Return value 12`.

A video camera icon is located in the bottom right corner of the visualization area.

**total of 22 so there's an interesting**

Chrome File Edit View History Bookmarks Window Help

Online Python Tutor - Visu X

pythontutor.com/composingprograms.html#mode=display

```

1 def f(x):
2     x = 4
3     def g(y):
4         def h(z):
5             nonlocal x
6             x = x + 1
7             return x + y + z
8         return h
9     return g
10 a = f(1)
11 b = a(2)
12 total = 10 + b(4)

```

[Edit code](#)

<< First < Back Program terminated Forward > Last >>

line that has just executed  
next line to execute

Frames Objects

Global frame

- f → func f(x) [parent=Global]
- a → func g(y) [parent=f1]
- b → func h(z) [parent=f2]
- total 21

f1: f [parent=Global]

- x 5
- g
- Return value

f2: g [parent=f1]

- y 2
- h
- Return value

f3: h [parent=f2]

- z 4
- Return value 11

**therefore X only got incremented once**

Generate permanent link

pythontutor.com/composingprograms.html#

when  $b(3) \rightarrow 10$ , the meaning of the statement is changed, because  $x = x + 1$  in line 7 is implemented only once, and thus change the result

## Environment Diagrams

## Go Bears!

```

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)

```

