61A Lecture 13

Monday, February 23

Announcements

Announcements

- Homework 4 due Monday 2/23 @ 11:59pm (small)

Announcements

- Homework 4 due Monday 2/23 @ 11:59pm (small)
- Project 2 due Thursday 2/26 @ 11:59pm (BIG!)

Announcements

- Homework 4 due Monday 2/23 @ 11:59pm (small)
- Project 2 due Thursday 2/26 @ 11:59pm (BIG!)
  - Office hours on Monday 2/23 3pm–5pm are relocated to 310 Soda

Announcements

- Homework 4 due Monday 2/23 @ 11:59pm (small)
- Project 2 due Thursday 2/26 @ 11:59pm (BIG!)
  - Office hours on Monday 2/23 3pm–5pm are relocated to 310 Soda
  - Project party on Tuesday 2/24 5pm–6:30pm in 2050 VLSB

Announcements

- Homework 4 due Monday 2/23 @ 11:59pm (small)
- Project 2 due Thursday 2/26 @ 11:59pm (BIG!)
  - Office hours on Monday 2/23 3pm–5pm are relocated to 310 Soda
  - Project party on Tuesday 2/24 5pm–6:30pm in 2050 VLSB
  - Extra office hours on Wednesday 2/25 4pm–6pm in Bechtel (Garbarini Lounge)

Announcements

- Homework 4 due Monday 2/23 @ 11:59pm (small)
- Project 2 due Thursday 2/26 @ 11:59pm (BIG!)
  - Office hours on Monday 2/23 3pm–5pm are relocated to 310 Soda
  - Project party on Tuesday 2/24 5pm–6:30pm in 2050 VLSB
  - Extra office hours on Wednesday 2/25 4pm–6pm in Bechtel (Garbarini Lounge)
  - Bonus point for early submission by Wednesday 2/25 @ 11:59pm!

Announcements

- Homework 4 due Monday 2/23 @ 11:59pm (small)
- Project 2 due Thursday 2/26 @ 11:59pm (BIG!)
  - Office hours on Monday 2/23 3pm–5pm are relocated to 310 Soda
  - Project party on Tuesday 2/24 5pm–6:30pm in 2050 VLSB
  - Extra office hours on Wednesday 2/25 4pm–6pm in Bechtel (Garbarini Lounge)
  - Bonus point for early submission by Wednesday 2/25 @ 11:59pm!
  - Thursday office hours will be held in...

Mutable Functions

A Function with Behavior That Varies Over Time

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

A Function with Behavior That Varies Over Time

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

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

A Function with Behavior That Varies Over Time

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

```
>>> withdraw(25)
75
```

A Function with Behavior That Varies Over Time

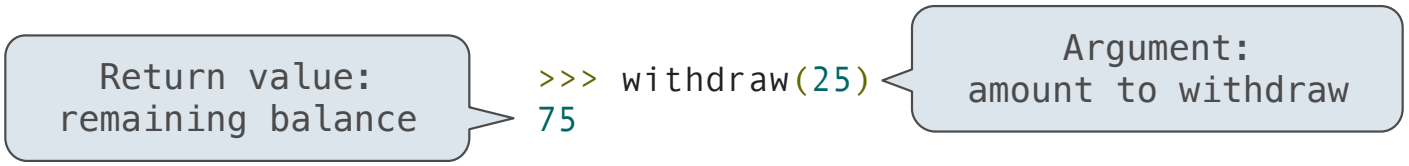
Let's model a bank account that has a balance of $100

```
>>> withdraw(25)
75
```

Argument:
amount to withdraw

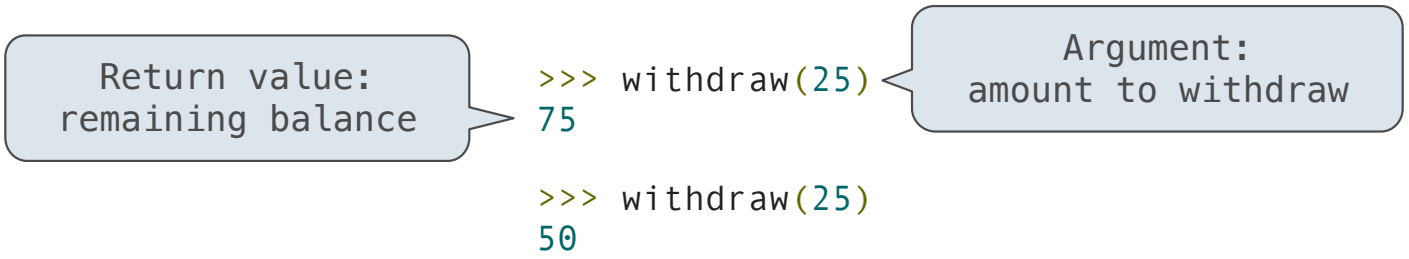
A Function with Behavior That Varies Over Time

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



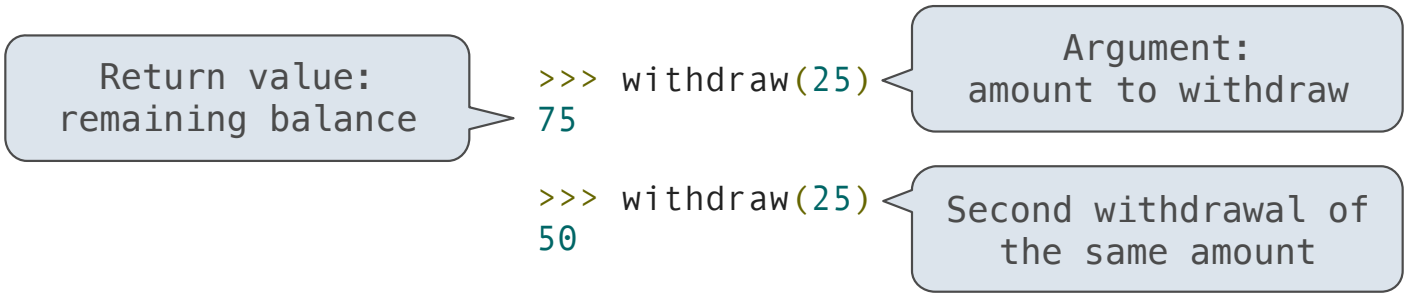
A Function with Behavior That Varies Over Time

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



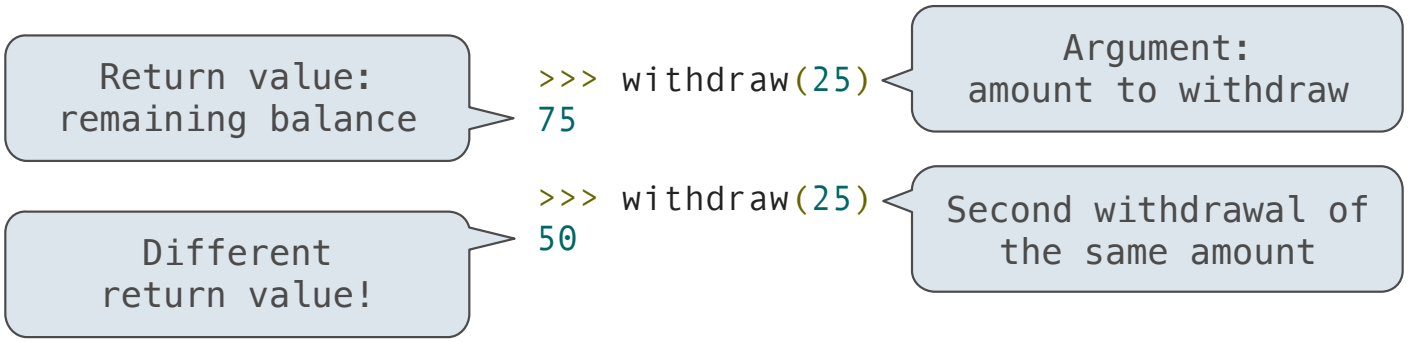
A Function with Behavior That Varies Over Time

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



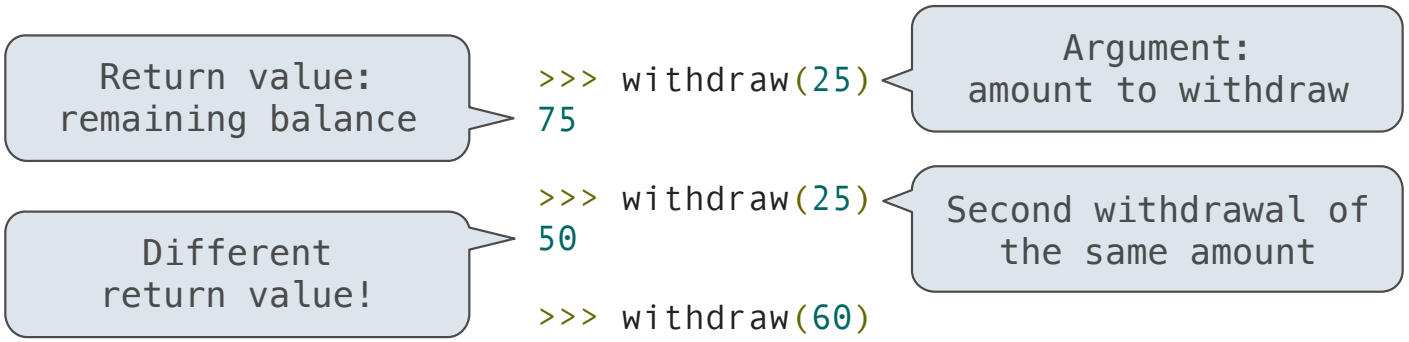
A Function with Behavior That Varies Over Time

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



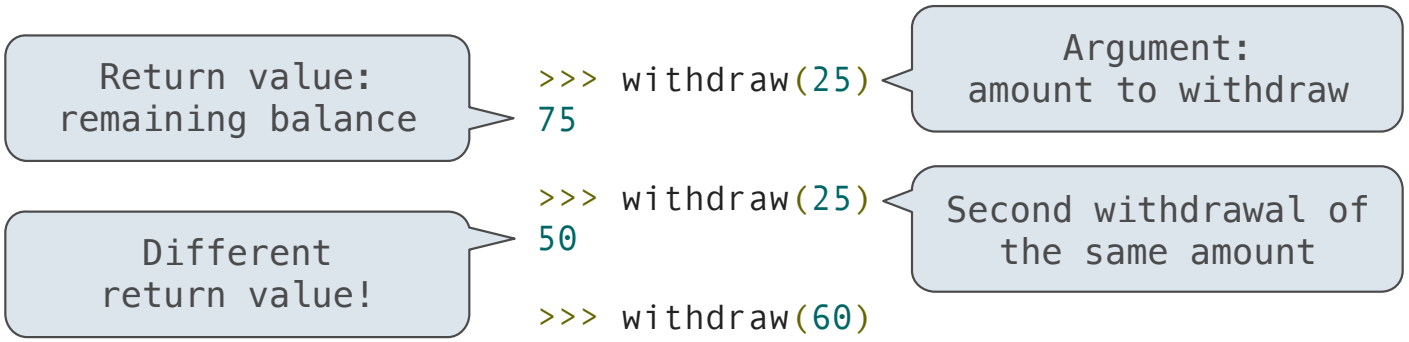
A Function with Behavior That Varies Over Time

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



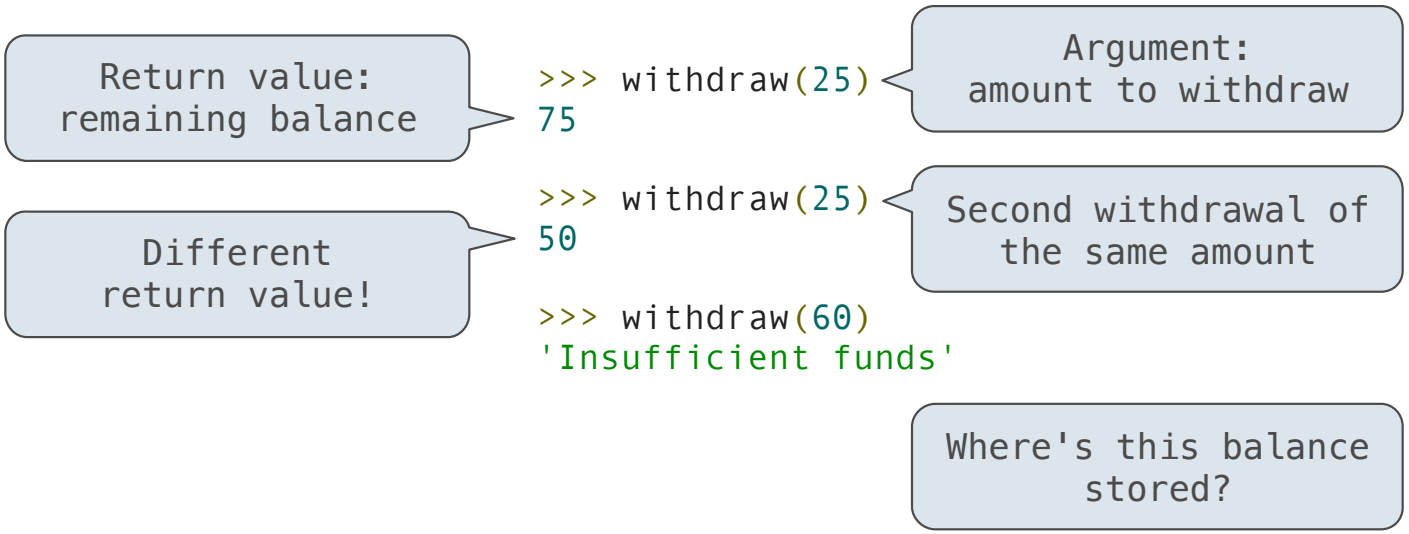
A Function with Behavior That Varies Over Time

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



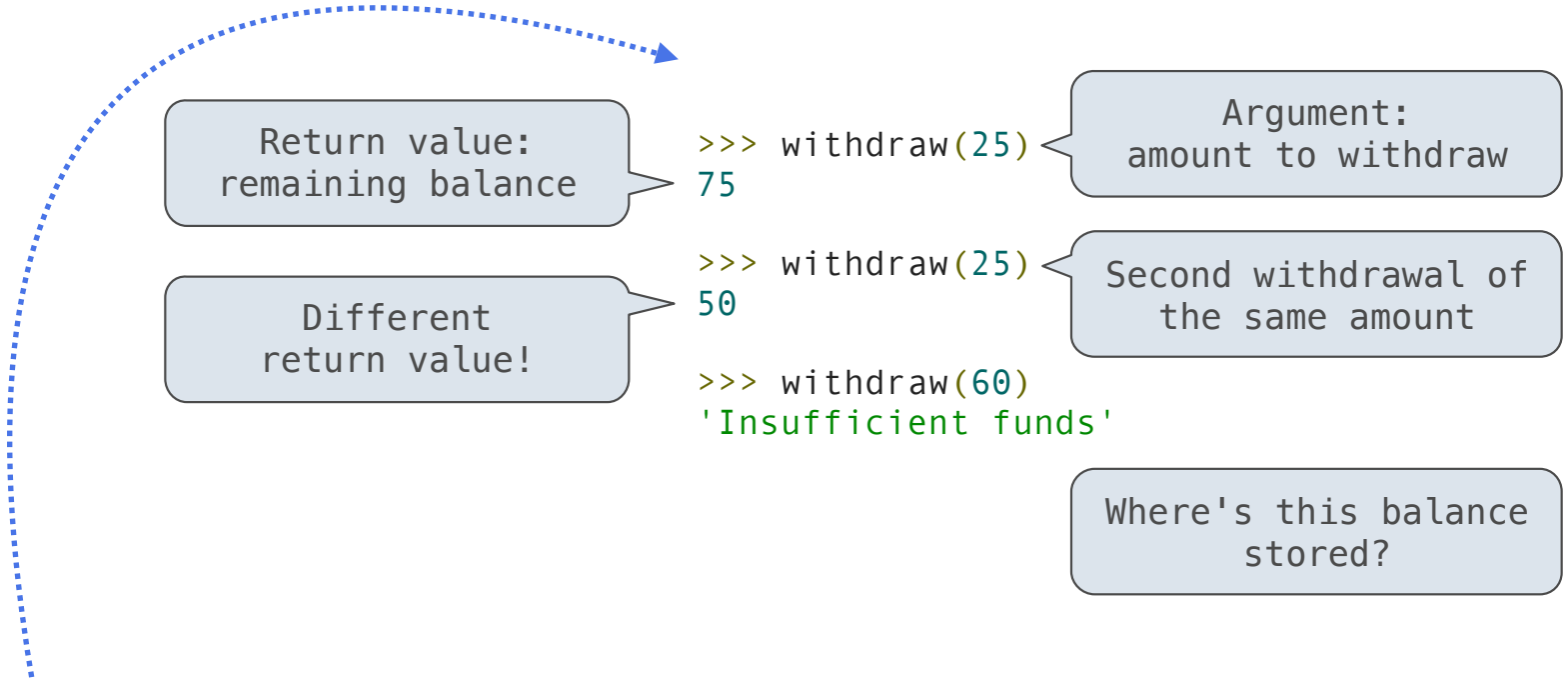
A Function with Behavior That Varies Over Time

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

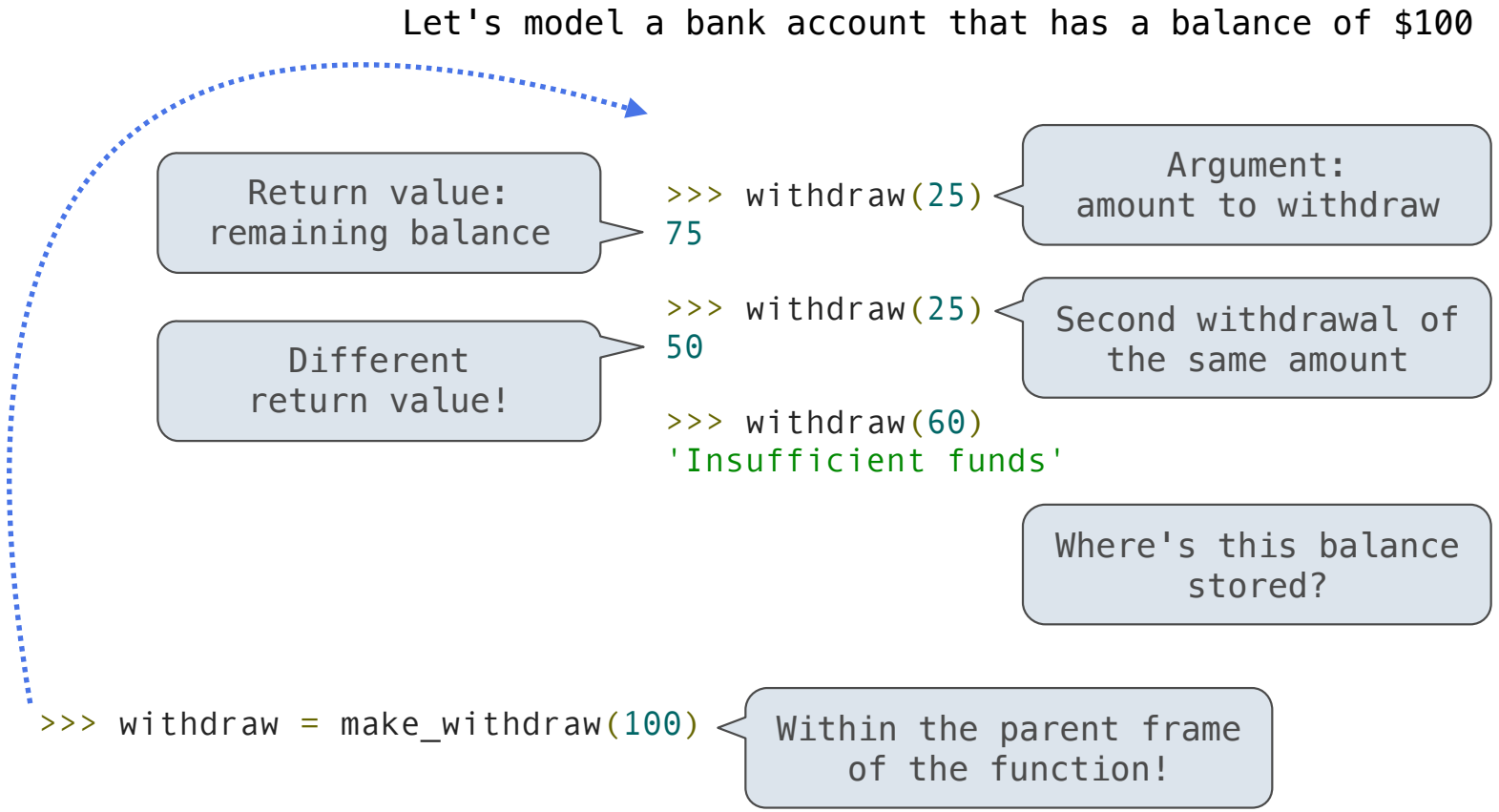


A Function with Behavior That Varies Over Time

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



A Function with Behavior That Varies Over Time



A Function with Behavior That Varies Over Time

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

```
>>> withdraw(25)
75
>>> withdraw(25)
50
>>> withdraw(60)
'Insufficient funds'
```

Return value: remaining balance

Argument: amount to withdraw

Different return value!

Second withdrawal of the same amount

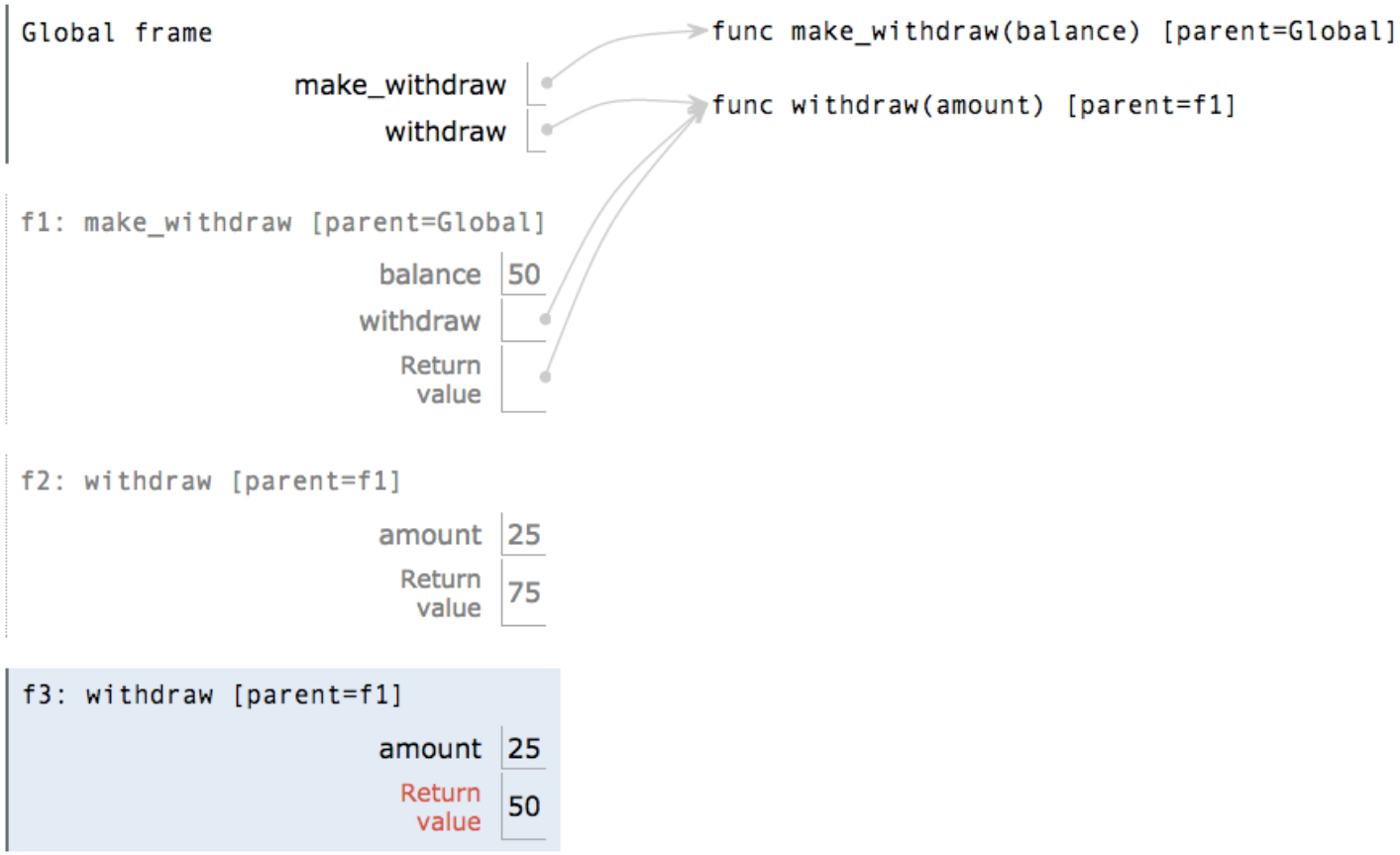
Where's this balance stored?

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

Within the parent frame of the function!

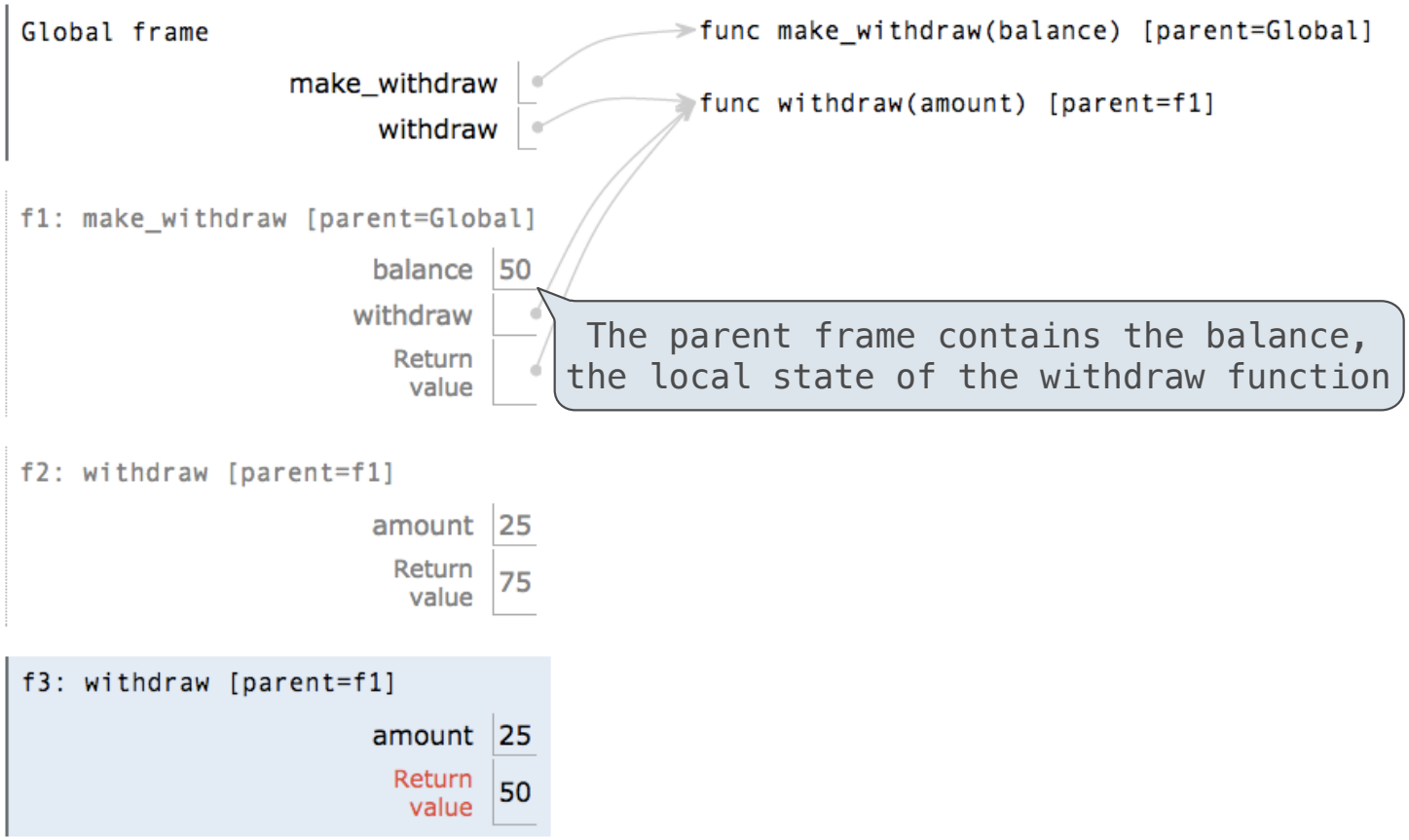
A function has a body and a parent environment

Persistent Local State Using Environments



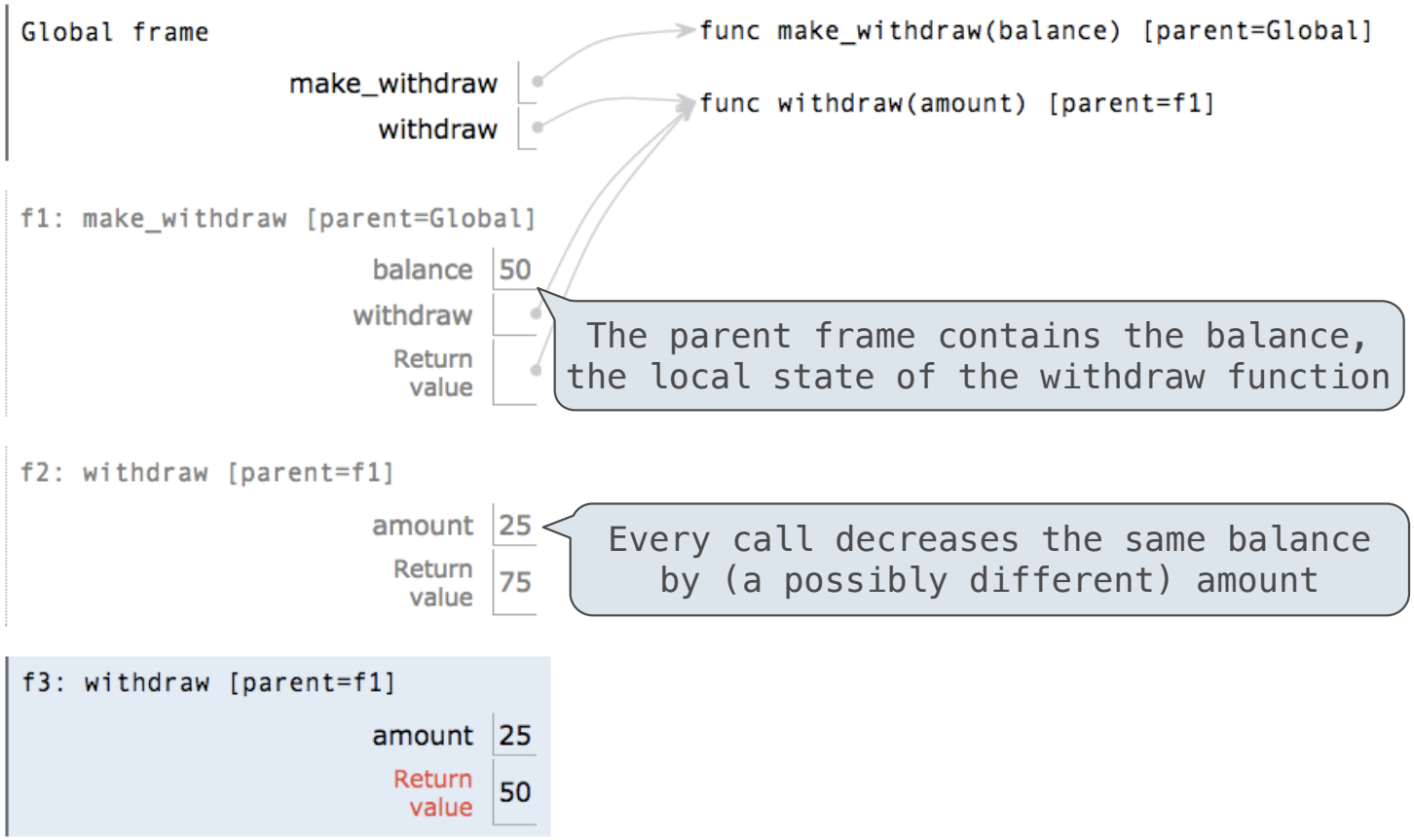
Interactive Diagram

Persistent Local State Using Environments



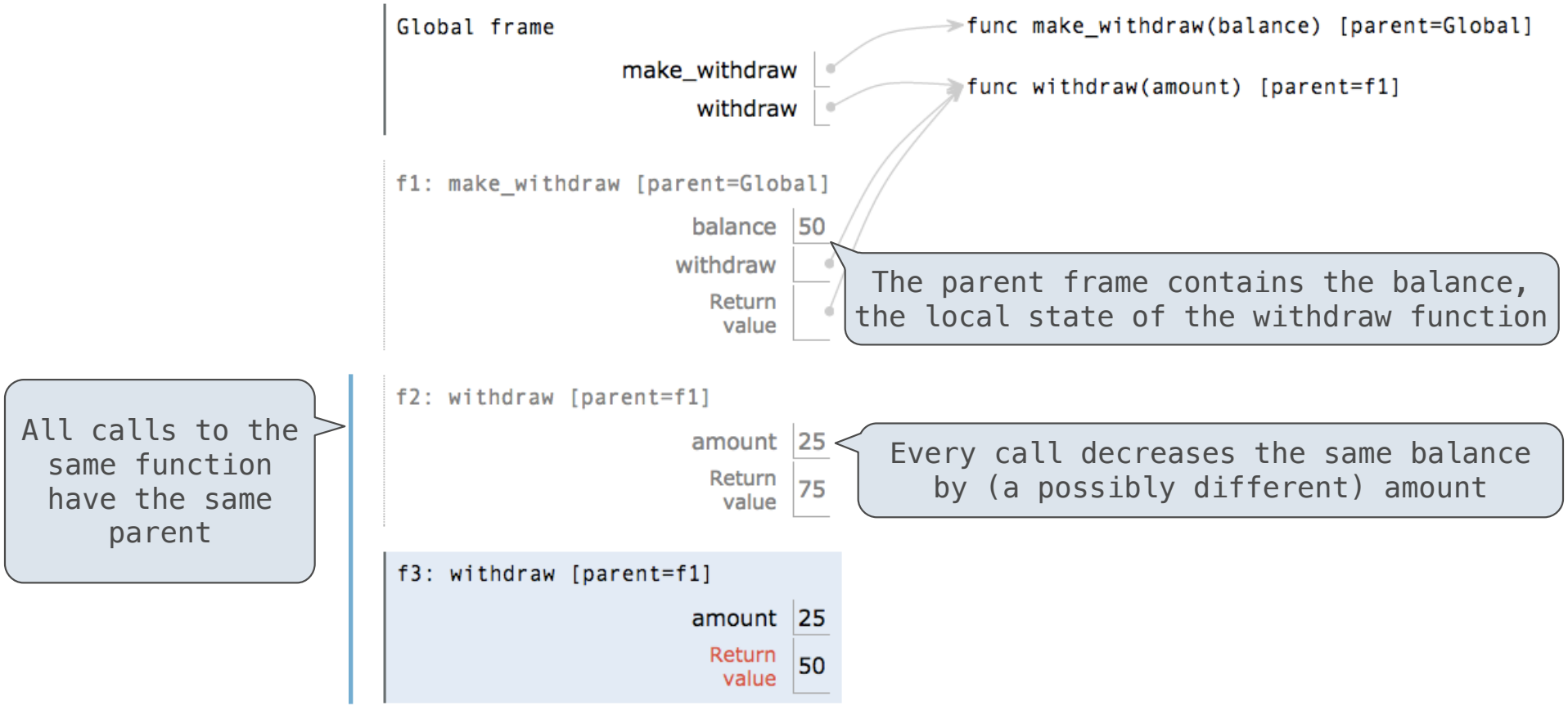
Interactive Diagram

Persistent Local State Using Environments



Interactive Diagram

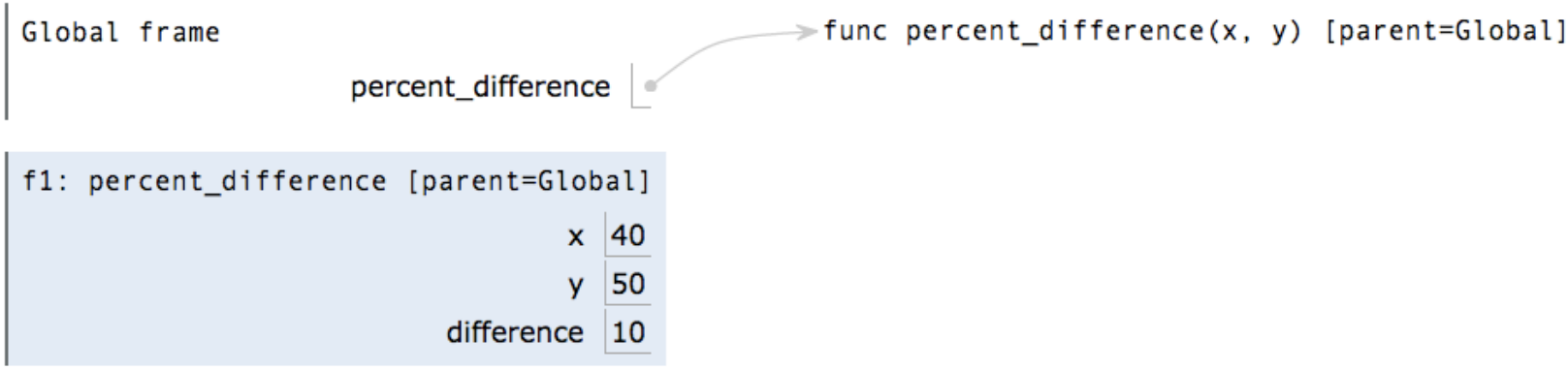
Persistent Local State Using Environments



Interactive Diagram

Reminder: Local Assignment

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

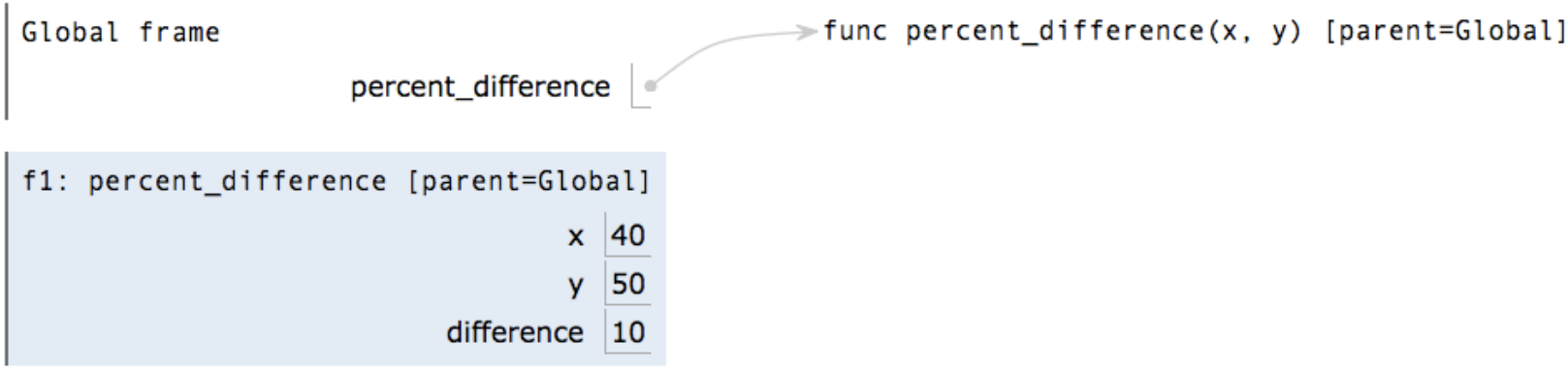


Interactive Diagram

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

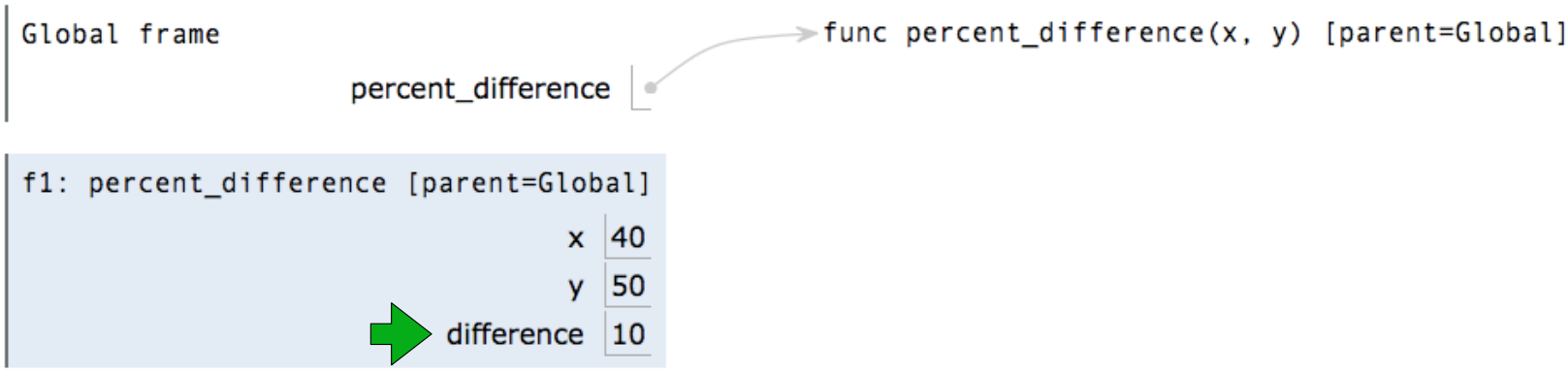


Interactive Diagram

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

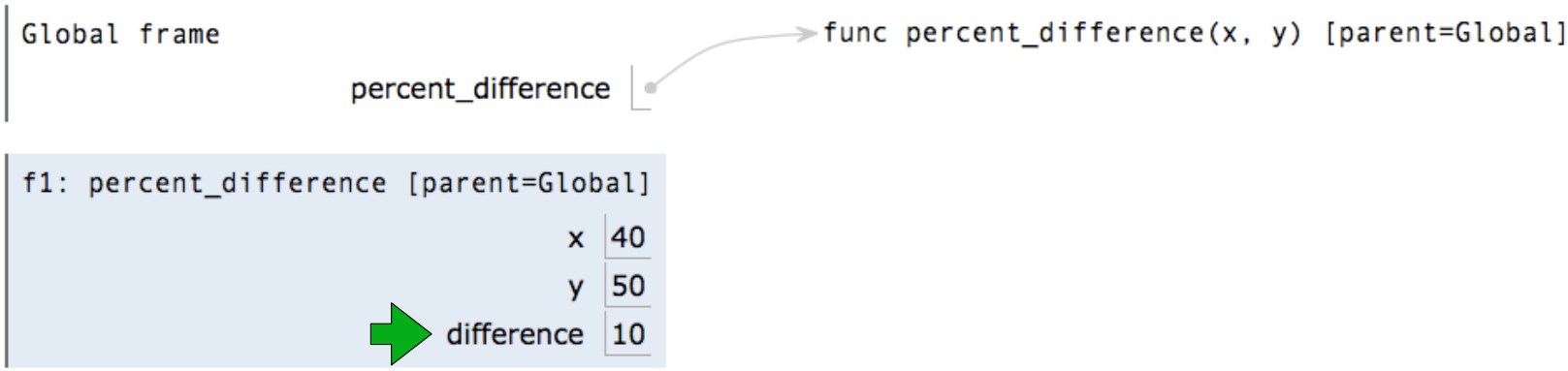


Interactive Diagram

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



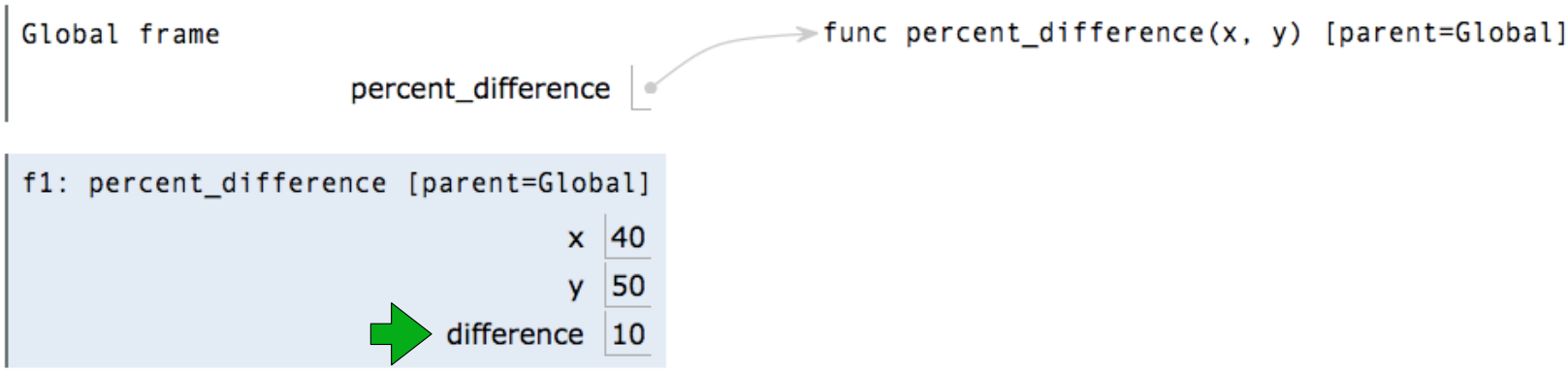
Execution rule for assignment statements:

Interactive Diagram

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



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**

Interactive Diagram

Non-Local Assignment & Persistent Local State

Non-Local Assignment & Persistent Local State

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

Non-Local Assignment & Persistent Local State

```
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."""
```

Non-Local Assignment & Persistent Local State

```
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."""
    def withdraw(amount):
```

Non-Local Assignment & Persistent Local State

```
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."""
    def withdraw(amount):
        nonlocal balance
```

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:
```

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'
```

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
```

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
```

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
```

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
```

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

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
```

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

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

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
```

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

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>
```

The Effect of Nonlocal Statements

```
nonlocal <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.

The Effect of Nonlocal Statements

```
nonlocal <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"

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"

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:

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.

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.

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

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

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://www.python.org/dev/peps/pep-3104/

The Many Meanings of Assignment Statements

```
x = 2
```

The Many Meanings of Assignment Statements

```
x = 2
```

Status **Effect**

The Many Meanings of Assignment Statements

```
x = 2
```

Status **Effect**

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

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

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	

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

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	

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

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	

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

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	

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 Particulars

Python pre-computes which frame contains each name before executing the body of a function.

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 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

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

Interactive Diagram

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

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

Local assignment

Interactive Diagram

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

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

Local assignment

UnboundLocalError: local variable 'balance' referenced before assignment

Interactive Diagram

Mutable Values & Persistent Local State

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

Mutable Values & Persistent Local State

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

```
def make_withdraw_list(balance):
    b = [balance]
    def withdraw(amount):
        if amount > b[0]:
            return 'Insufficient funds'
        b[0] = b[0] - amount
        return b[0]
    return withdraw

withdraw = make_withdraw_list(100)
withdraw(25)
```

Mutable Values & Persistent Local State

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

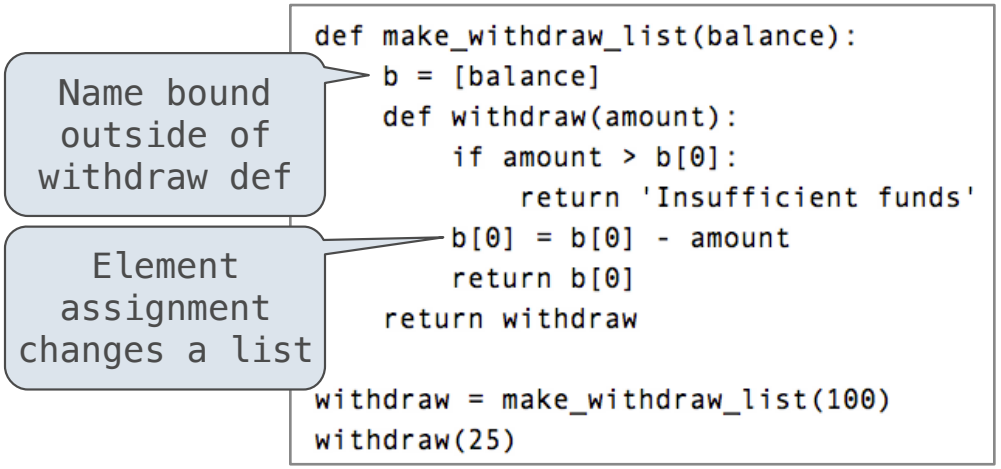
Name bound outside of withdraw def

```
def make_withdraw_list(balance):
    b = [balance]
    def withdraw(amount):
        if amount > b[0]:
            return 'Insufficient funds'
        b[0] = b[0] - amount
        return b[0]
    return withdraw

withdraw = make_withdraw_list(100)
withdraw(25)
```

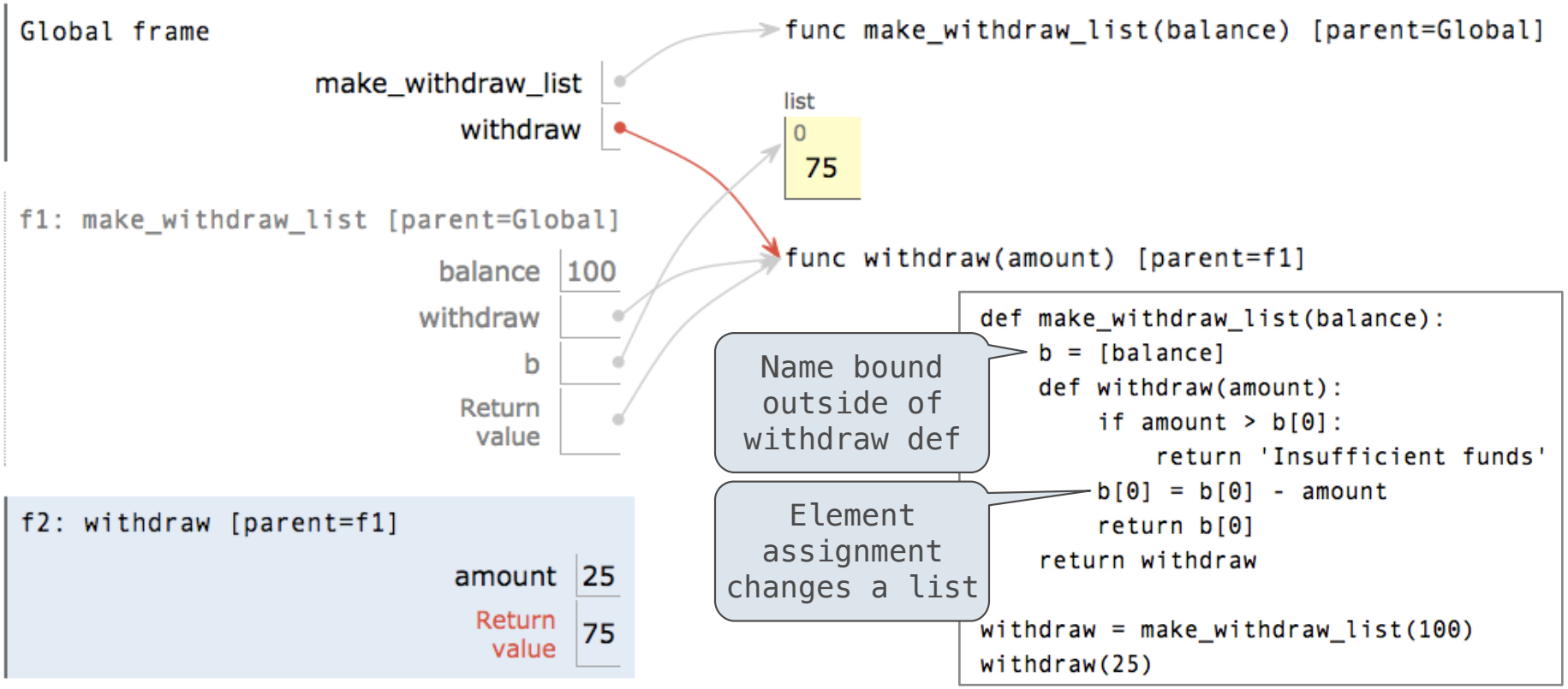
Mutable Values & Persistent Local State

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



Mutable Values & Persistent Local State

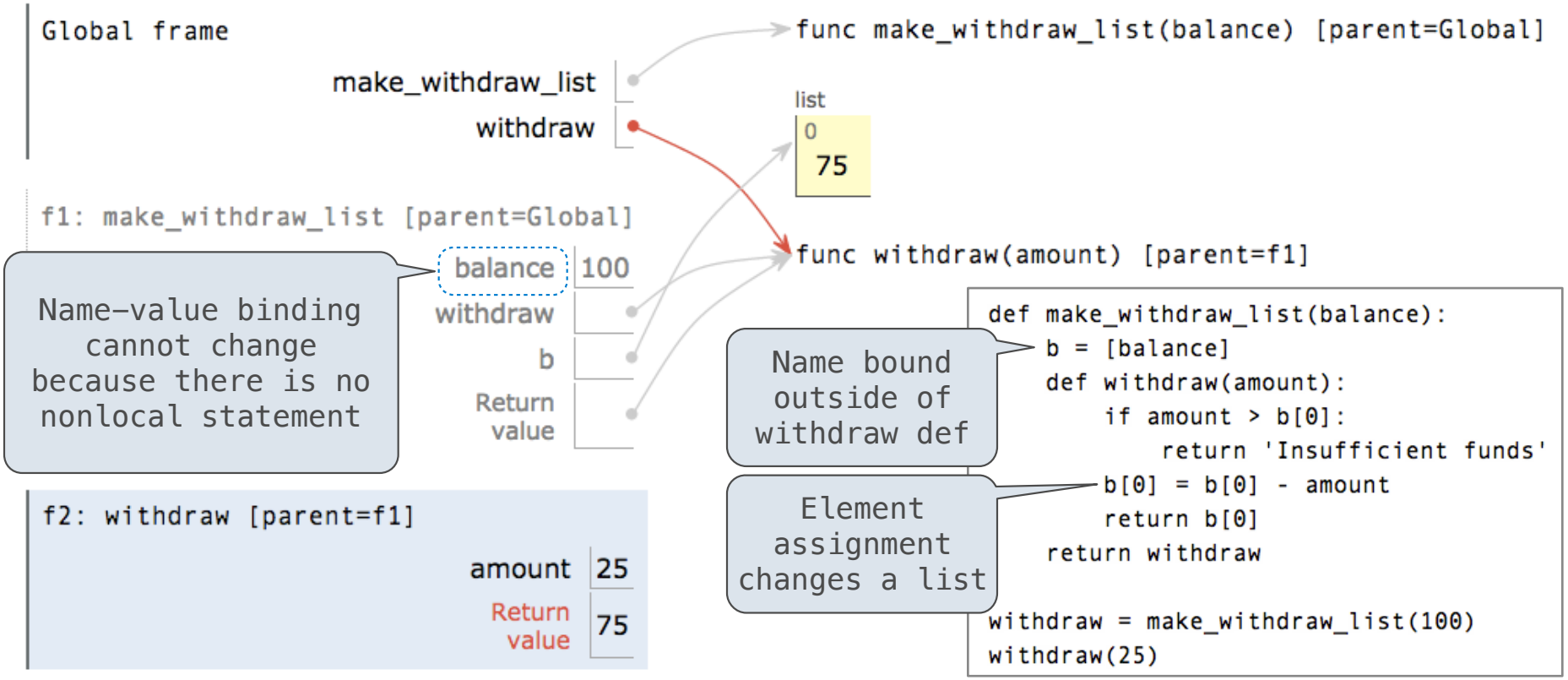
Mutable values can be changed *without* a nonlocal statement.



Interactive Diagram

Mutable Values & Persistent Local State

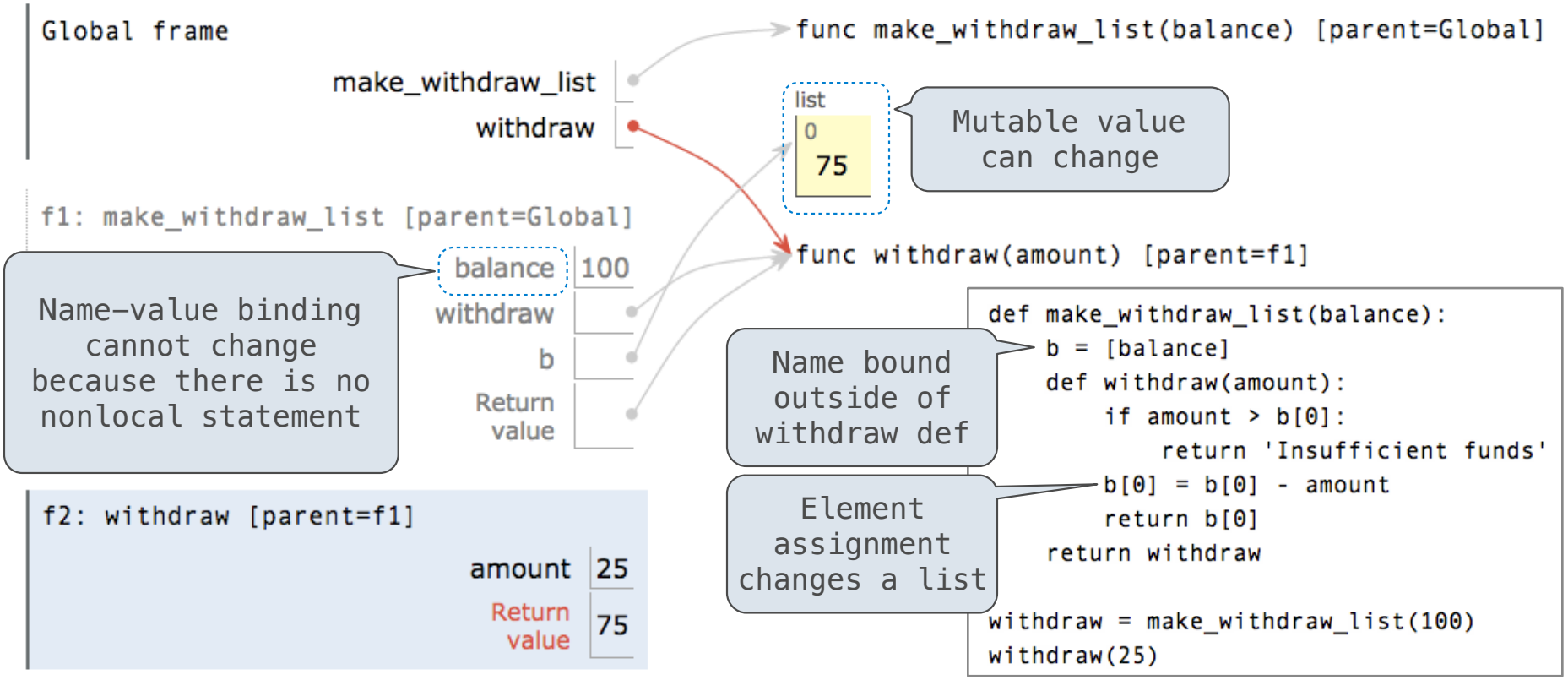
Mutable values can be changed *without* a nonlocal statement.



Interactive Diagram

Mutable Values & Persistent Local State

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



Interactive Diagram

Multiple Mutable Functions

(Demo)

Referential Transparency, Lost

Interactive Diagram

Referential Transparency, Lost

- Expressions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.

Interactive Diagram

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))
```

Interactive Diagram

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))
```

Interactive Diagram

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))
```

Interactive Diagram

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.**

Interactive Diagram

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.**

Interactive Diagram

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.**

Interactive Diagram