61A Lecture 13

Wednesday, September 26

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)
```

Argument:
amount to withdraw

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

A Function with Behavior That Varies Over Time

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

Return value:
remaining balance

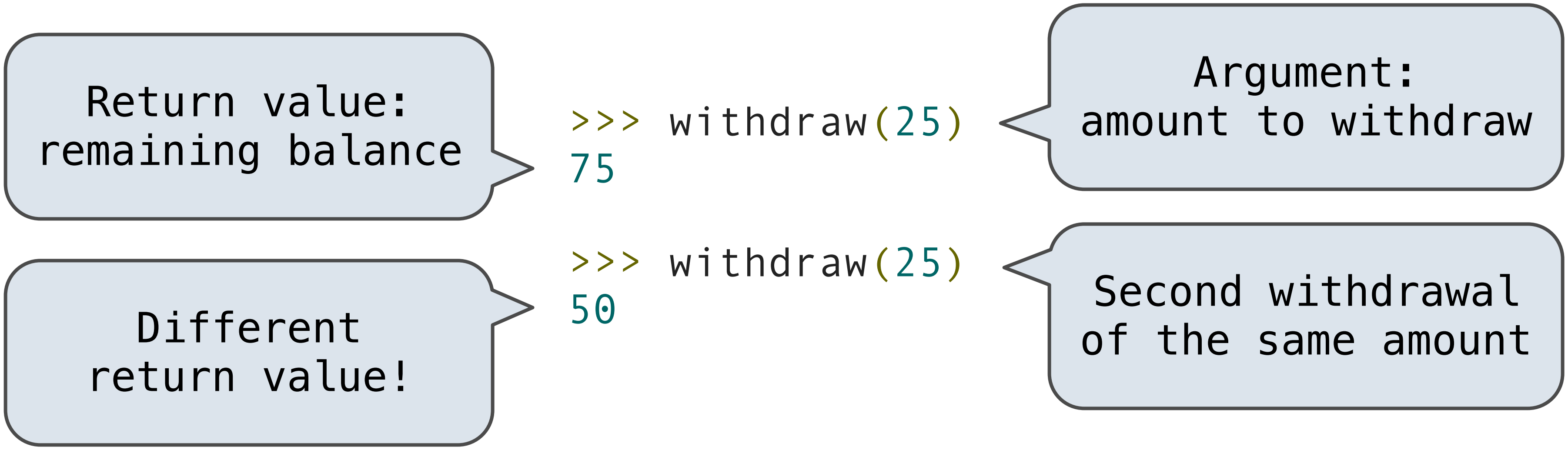
```
>>> withdraw(25)
75

>>> withdraw(25)
```

Argument:
amount to withdraw

Second withdrawal
of the same amount

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

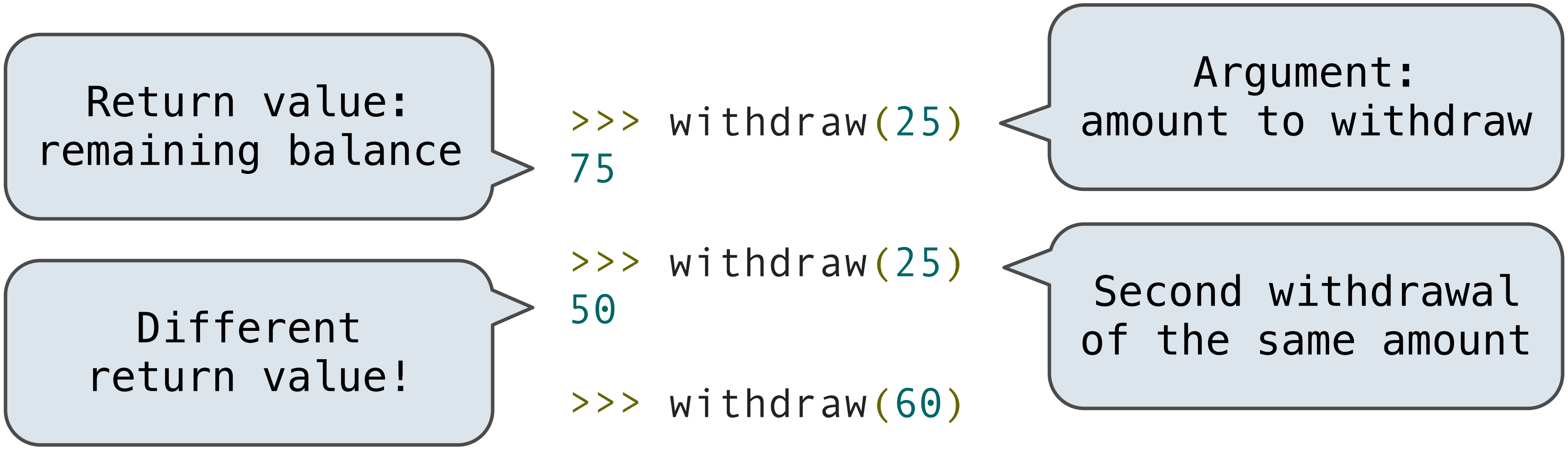
Different return value!

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

Second withdrawal of the same amount

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)
```

Return value: remaining balance

Different return value!

Argument: amount to withdraw

Second withdrawal of the same amount

A Function with Behavior That Varies Over Time

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

Return value:
remaining balance

Argument:
amount to withdraw

```
>>> withdraw(25)
75

>>> withdraw(25)
50

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

Different
return value!

Second withdrawal
of the same amount

A Function with Behavior That Varies Over Time

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

Return value: remaining balance

Different return value!

```
>>> withdraw(25)
75

>>> withdraw(25)
50

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
```

Argument: amount to withdraw

Second withdrawal of the same amount

A Function with Behavior That Varies Over Time

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

Return value:
remaining balance

Argument:
amount to withdraw

```
>>> withdraw(25)
75

>>> withdraw(25)
50

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35
```

Different
return value!

Second withdrawal
of the same amount

A Function with Behavior That Varies Over Time

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

Return value:
remaining balance

Argument:
amount to withdraw

```
>>> withdraw(25)
75

>>> withdraw(25)
50

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35
```

Different
return value!

Second withdrawal
of the same amount

Where's this
balance stored?

A Function with Behavior That Varies Over Time

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

Return value: remaining balance

Argument: amount to withdraw

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

Different return value!

Second withdrawal of the same amount

```
>>> withdraw(25)
50

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35
```

Where's this balance stored?

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

A Function with Behavior That Varies Over Time

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

Return value: remaining balance

Argument: amount to withdraw

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

Different return value!

Second withdrawal of the same amount

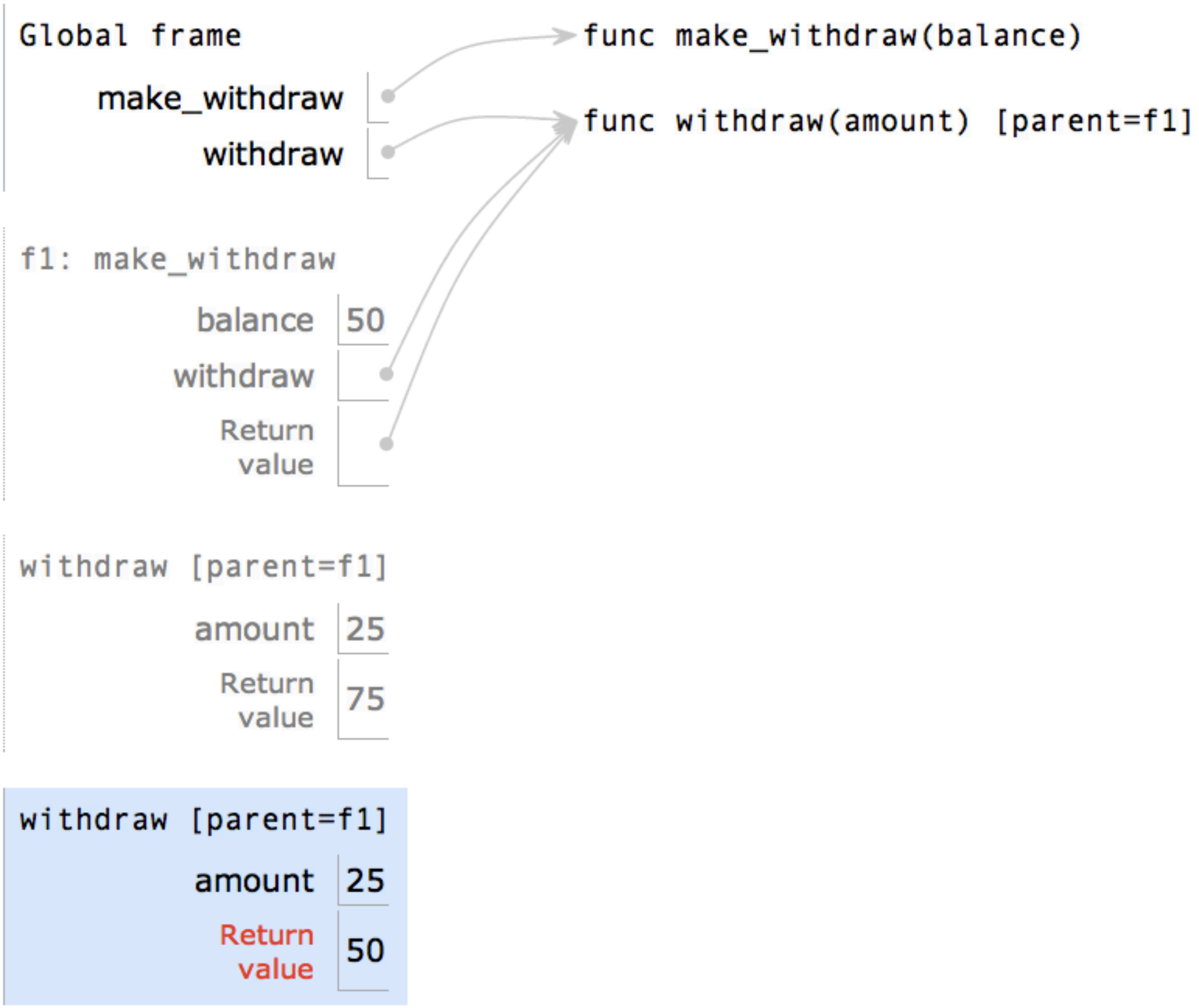
```
>>> withdraw(25)
50

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35
```

Where's this balance stored?

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

Within the function!

Persistent Local State



Global frame

make_withdraw

withdraw

func make_withdraw(balance)

func withdraw(amount) [parent=f1]

f1: make_withdraw

balance 50

withdraw

Return value

withdraw [parent=f1]

amount 25

Return value 75

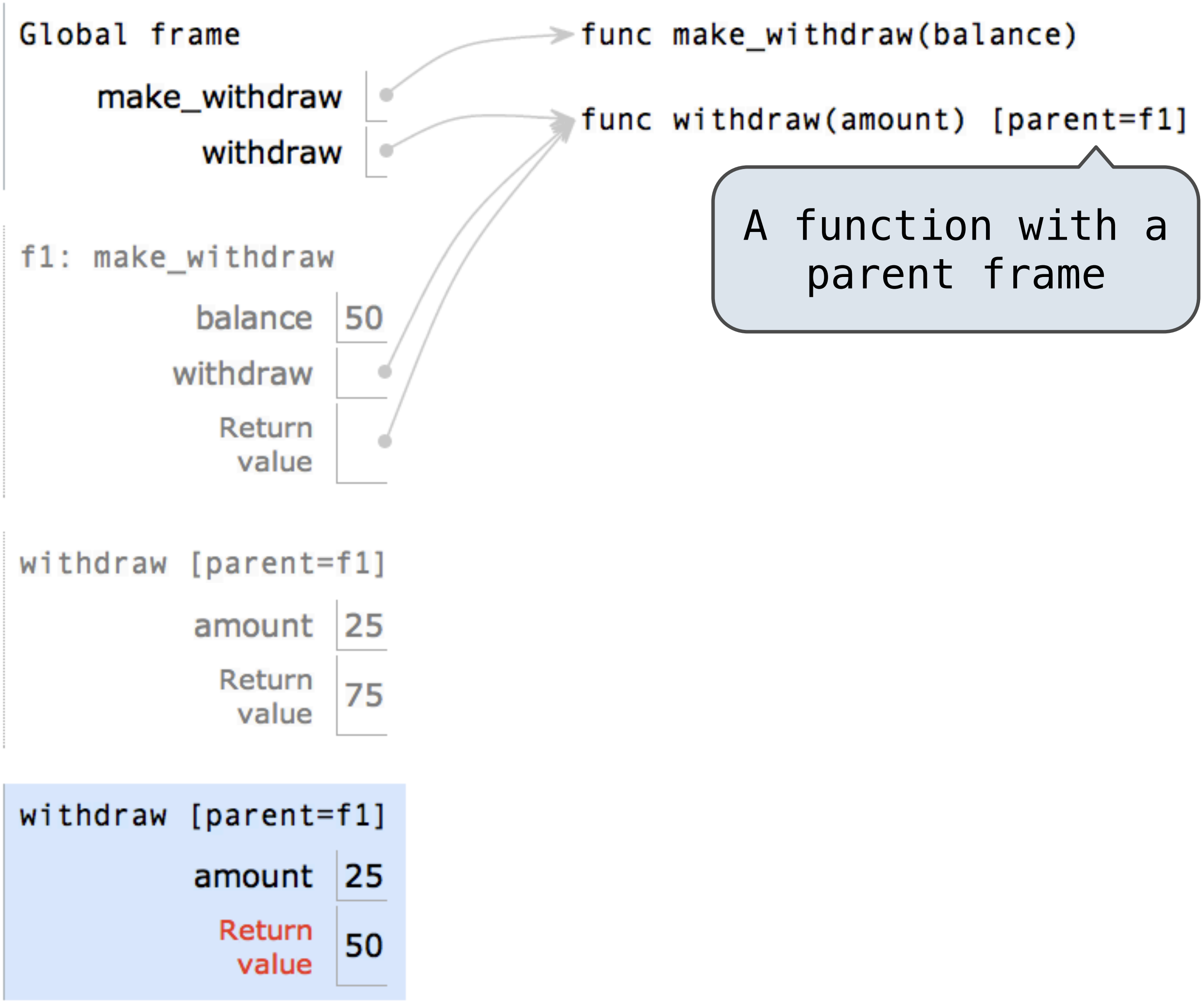
withdraw [parent=f1]

amount 25

Return value 50

http://goo.gl/StRZP

Persistent Local State



Global frame

make_withdraw

withdraw

func make_withdraw(balance)

func withdraw(amount) [parent=f1]

f1: make_withdraw

balance 50

withdraw

Return value

withdraw [parent=f1]

amount 25

Return value 75

withdraw [parent=f1]

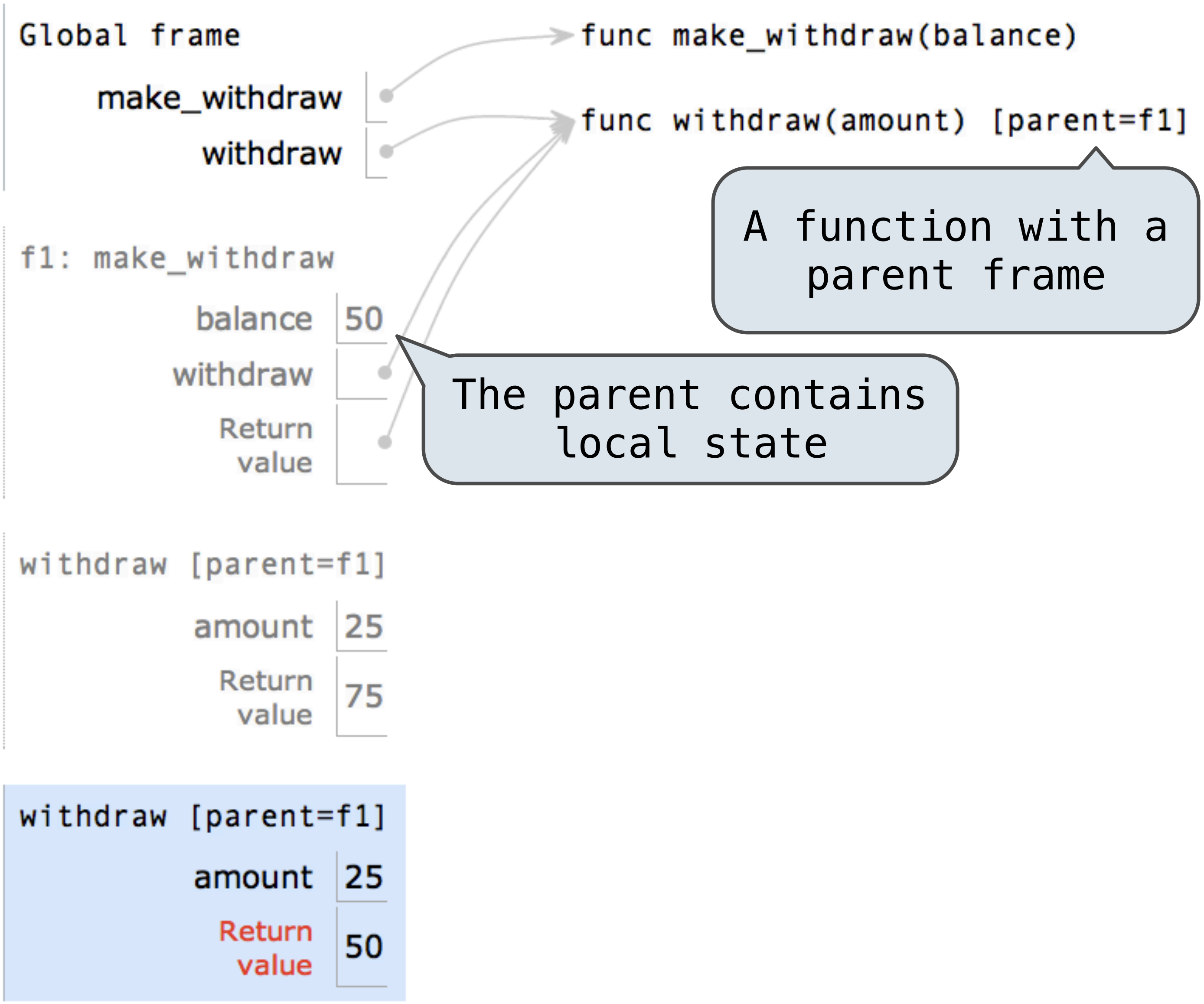
amount 25

Return value 50

A function with a parent frame

http://goo.gl/StRZP

Persistent Local State



Global frame

make_withdraw

withdraw

func make_withdraw(balance)

func withdraw(amount) [parent=f1]

f1: make_withdraw

balance	50
withdraw	
Return value	

withdraw [parent=f1]

amount	25
Return value	75

withdraw [parent=f1]

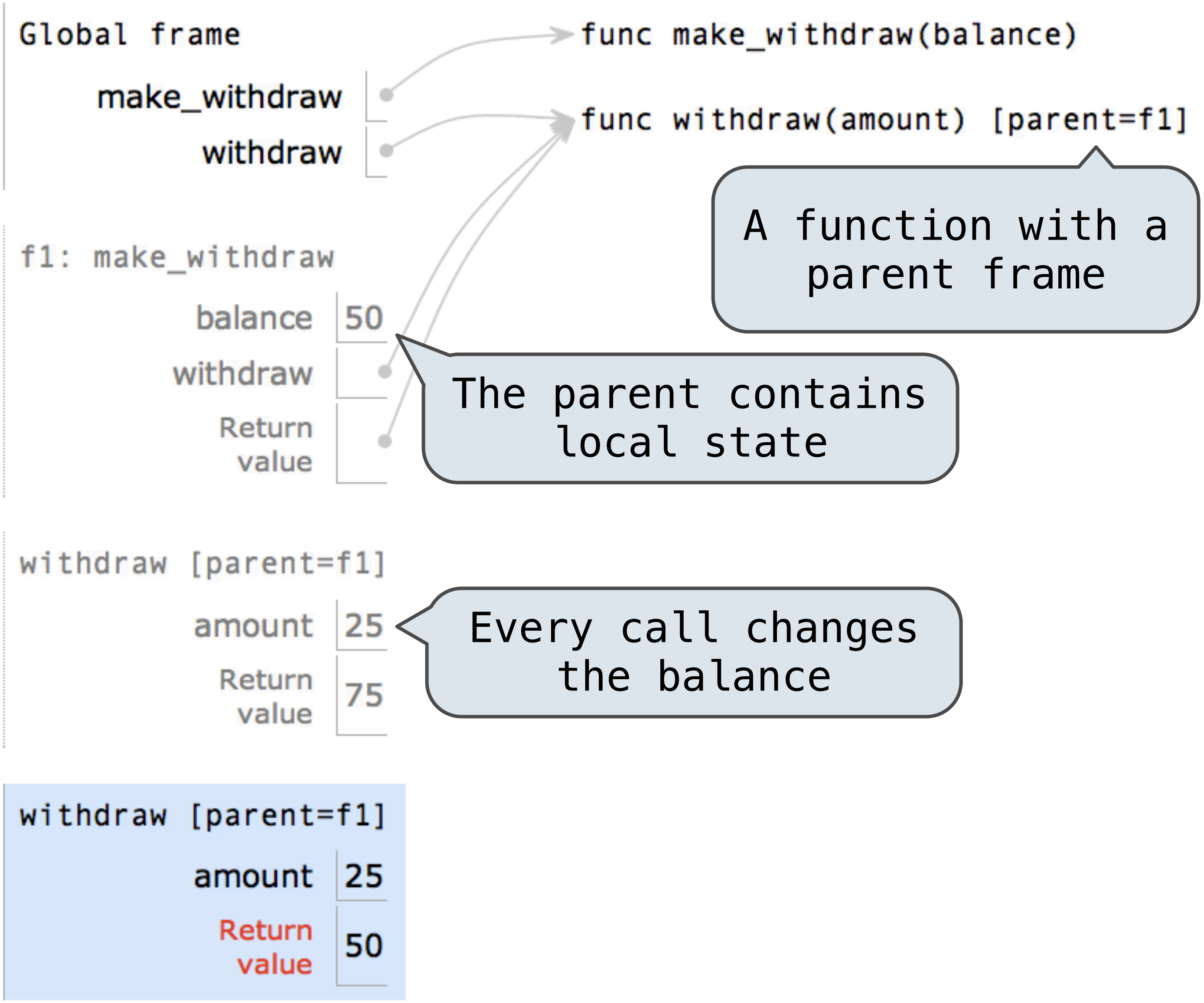
amount	25
Return value	50

A function with a parent frame

The parent contains local state

http://goo.gl/StRZP

Persistent Local State



A function with a parent frame

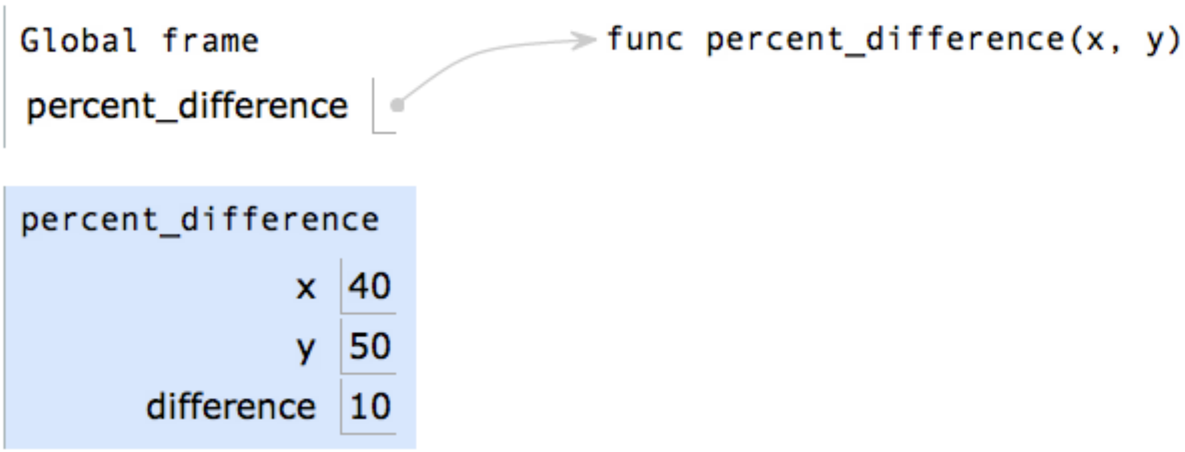
The parent contains local state

Every call changes the balance

http://goo.gl/StRZP

Reminder: Local Assignment

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



Global frame

percent_difference

func percent_difference(x, y)

percent_difference

x 40

y 50

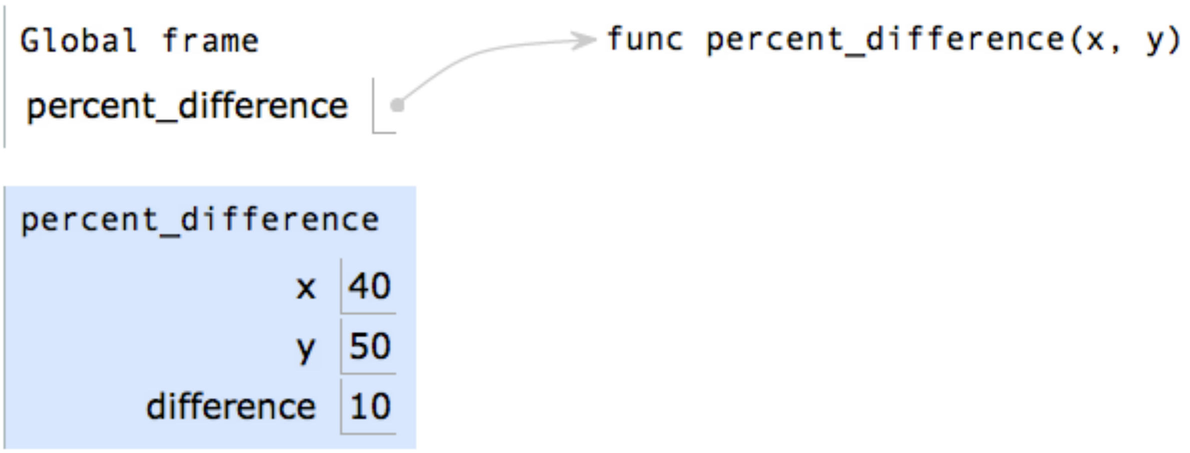
difference 10

Example: http://goo.gl/wcF71

Reminder: Local Assignment

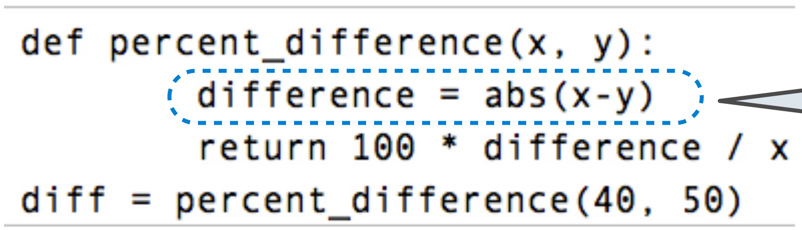
```
def percent_difference(x, y):
        difference = abs(x-y)
        return 100 * difference / x
diff = percent_difference(40, 50)
```

Assignment binds names to values in the current local frame



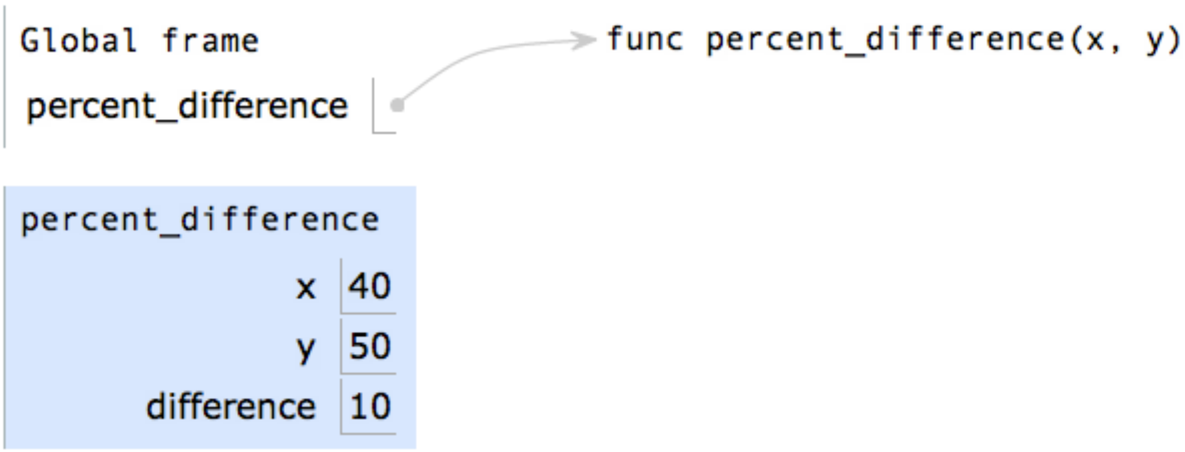
Example: http://goo.gl/wcF71

Reminder: Local Assignment



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

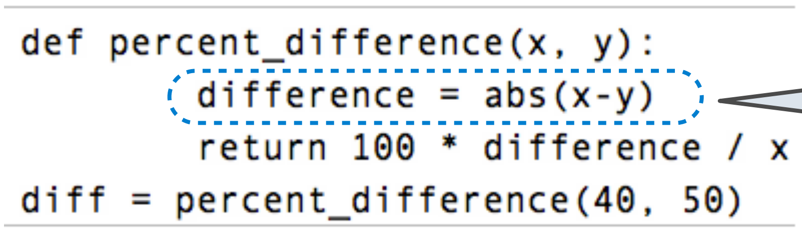
Assignment binds names to values in the current local frame



Execution rule for assignment statements:

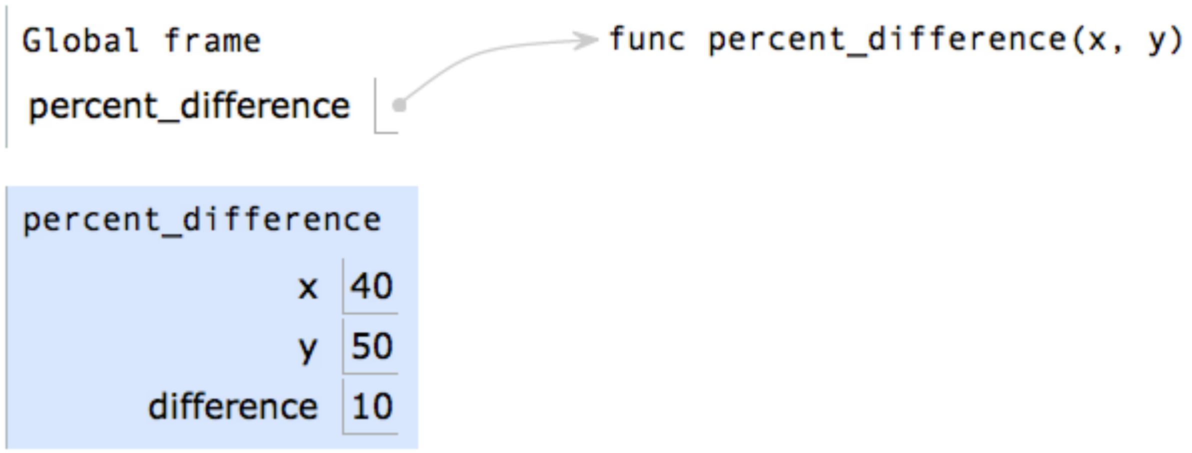
Example: http://goo.gl/wcF71

Reminder: Local Assignment



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

Assignment binds names to values in the current local frame



Global frame

percent_difference → func percent_difference(x, y)

percent_difference

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 the resulting values in the **first frame** of the current environment.

Example: http://goo.gl/wcF71

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
```

Declare the name "balance" nonlocal

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:
```

Declare the name "balance" nonlocal

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'
```

Declare the name "balance" nonlocal

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
```

Declare the name "balance" nonlocal

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
```

Declare the name "balance" nonlocal

Re-bind balance where 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
```

Declare the name "balance" nonlocal

Re-bind balance where 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

Re-bind balance where 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

Re-bind balance where it was bound previously

Demo

The Effect of Nonlocal Statements

```
nonlocal <name>
```

The Effect of Nonlocal Statements

```
nonlocal <name>
```

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.

The Effect of Nonlocal Statements

```
nonlocal <name>
```

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"

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"

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:

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.

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.

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

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

```
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 env.

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 env.
• 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 env.
• 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 it 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 env.
• 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 it 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 env.
• 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 it 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 env.
• 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 it 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 env.
• 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 it 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

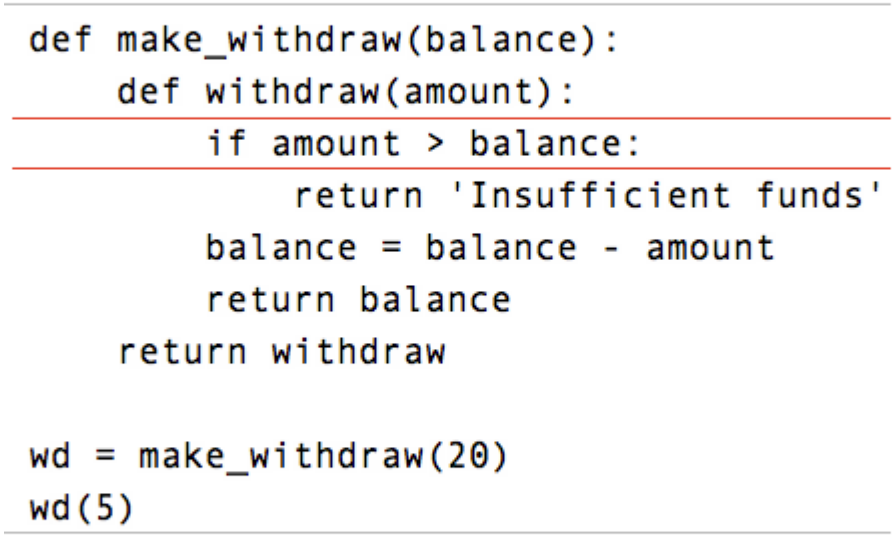
Python pre-computes which frame contains each name before executing the body of a function.

Therefore, 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.

Therefore, within the body of a function, all instances of a name must refer to the same frame.

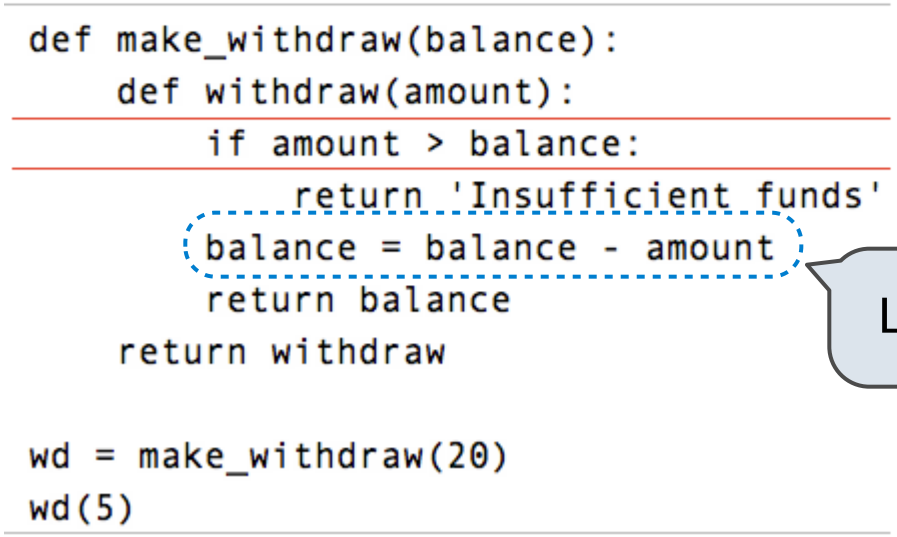


```
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw

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

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.



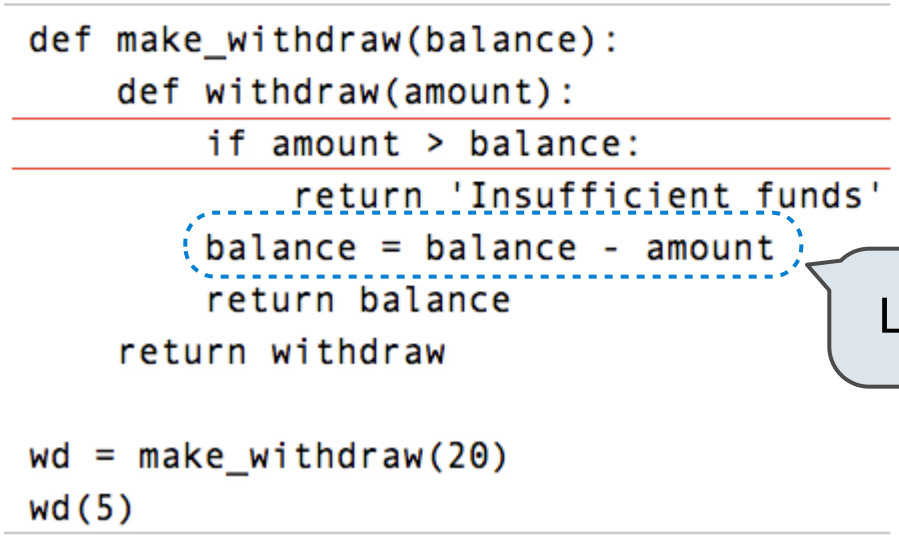
```
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw

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

Local assignment

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.



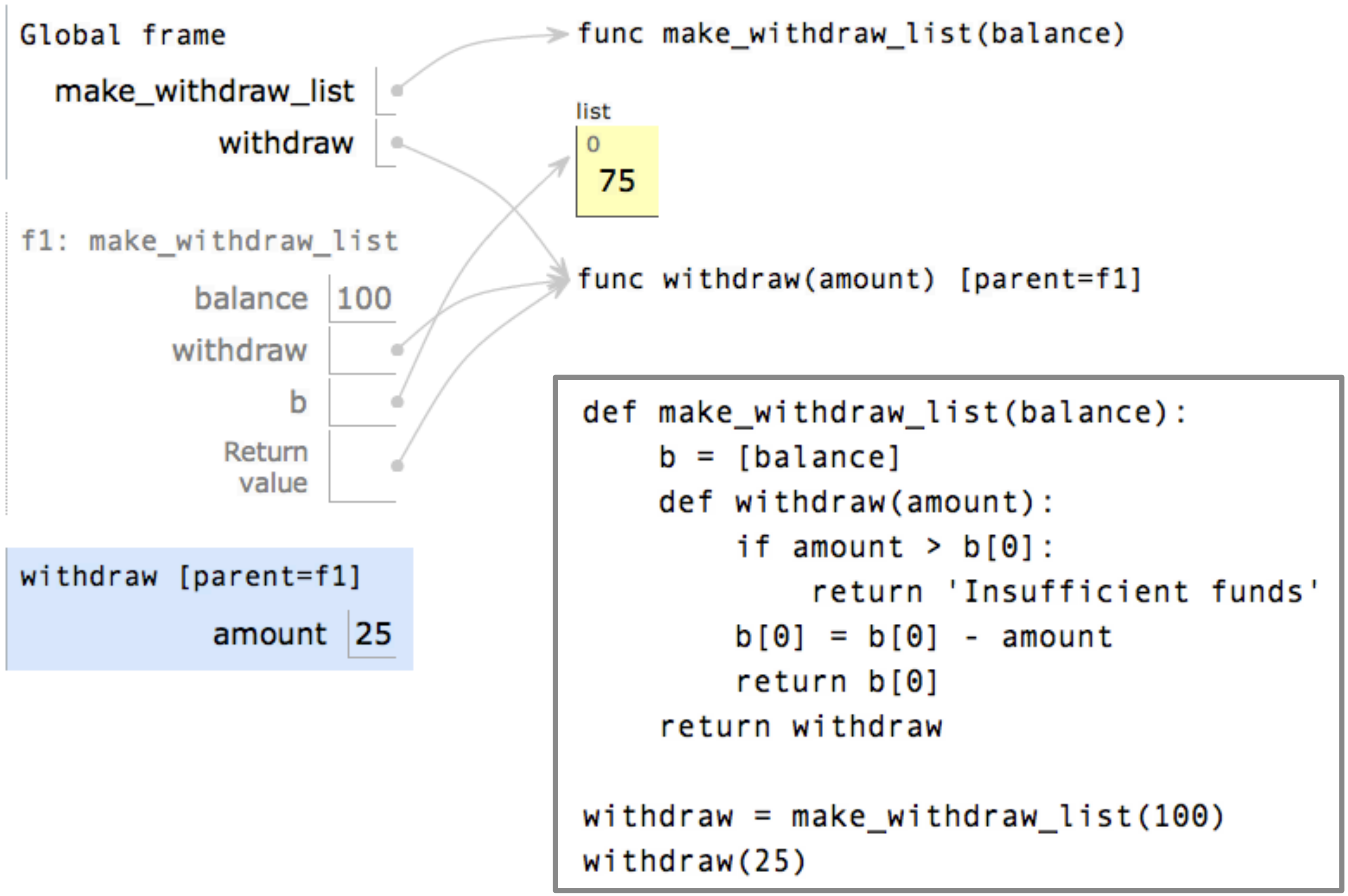
```
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw

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

Local assignment

UnboundLocalError: local variable 'balance' referenced before assignment

Mutable Values & Persistent Local State

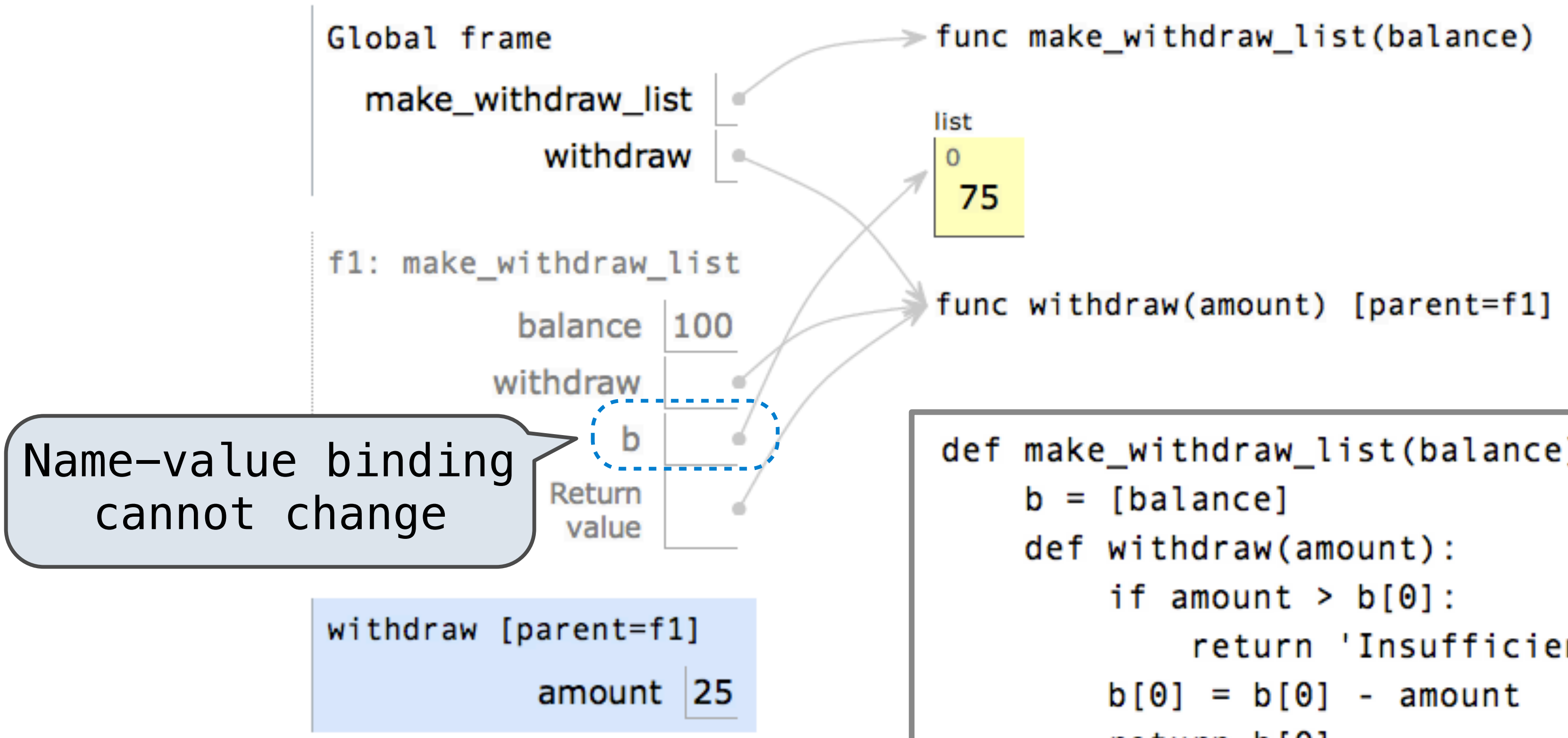
Mutable values can be changed *without* a nonlocal statement.



```
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.

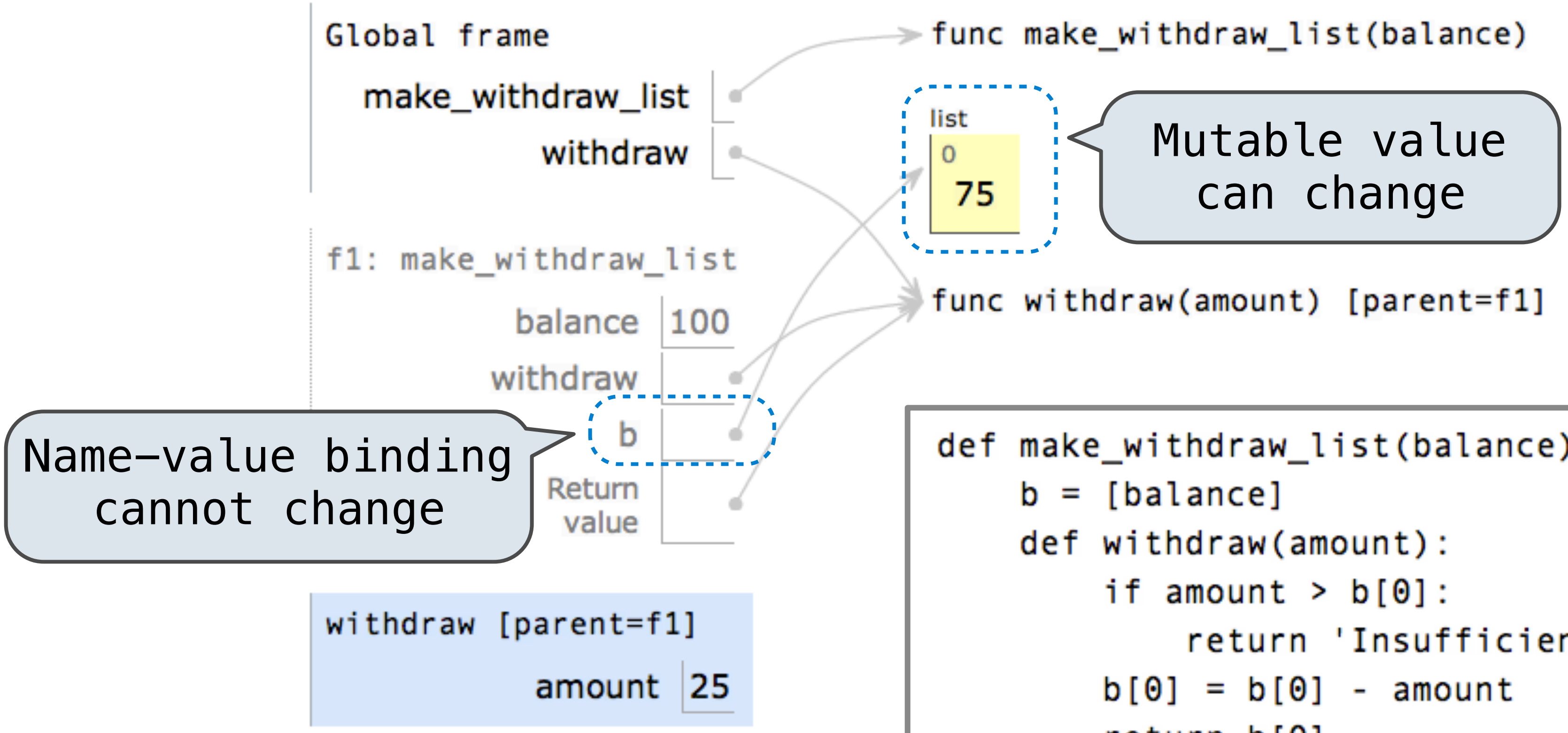


Name-value binding cannot change

```
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 value can change

Name-value binding cannot change

```
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)
```

Creating Two Different Withdraw Functions

Demo

The Benefit of Non-Local Assignment

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 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.**

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.**

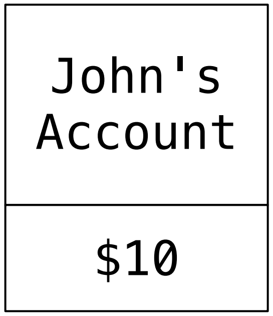
- An abstraction of a bank account that **manages its own internal state.**

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

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

Demo

Sameness and Change

Sameness and Change

- As long as we **never modify** objects, we can regard a compound object to be precisely the **totality of its pieces.**

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.

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.**

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.

Sameness and Change

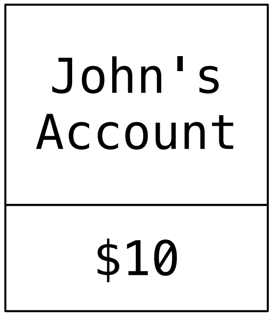
- As long as we **never modify** objects, we can regard a compound object to be precisely the **totality of its pieces.**
- A **rational number** is just its numerator and denominator.
- This view is no longer valid **in the presence of change.**
- Now, a compound data **object has an "identity"** that 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.

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.**

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.**



John's Account

$10

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.**

John's Account

$10

Steven's Account

$10

Referential Transparency, Lost

Referential Transparency, Lost

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

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))
```

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))
```

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))
```

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))
```

- 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.

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))
```

- 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.

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))
```

- 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.

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))
```

- 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.

Demo