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
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

```python
>>> withdraw(25)
75
```
A Function with Behavior That Varies Over Time

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

```python
>>> 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

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

Return value: remaining balance

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)
50
Let's model a bank account that has a balance of $100.

Return value: remaining balance

>>> withdraw(25)
75

>>> withdraw(25)
50

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
  - >>> withdraw(25)
    - 50

Argument:
- amount to withdraw

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

Different return value!

>>> withdraw(25)
75

>>> withdraw(25)
50

Second withdrawal of the same amount

>>> withdraw(60)
'Insufficient funds'
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) 50
- >>> withdraw(60) 'Insufficient funds'
- >>> withdraw(15) 35

Argument: amount to withdraw

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

Second withdrawal of the same amount

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35

Where's this balance stored?

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)
50

Different return value!

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35

Argument: amount to withdraw

Second withdrawal of the same amount

Where's this balance stored?

>>> withdraw = make_withdraw(100)
Let's model a bank account that has a balance of $100

Return value: remaining balance

Different return value!

Argument: amount to withdraw

Second withdrawal of the same amount

Within the parent frame of the function!
A Function with Behavior That Varies Over Time

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

```python
>>> withdraw(25)
75
```
```
>>> withdraw(25)
50
```
```
>>> withdraw(60)
'Insufficient funds'
```
```
>>> withdraw(15)
35
```

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

Different return value!

Where's this balance stored?

Within the parent frame of the function!

A function has a body and a parent environment
Persistent Local State Using Environments

Global frame

- `make_withdraw` [parent=Global]
  - `withdraw`

f1: `make_withdraw` [parent=Global]
- `balance` 50
- `withdraw`
- Return value

f2: `withdraw` [parent=f1]
- `amount` 25
- Return value 75

f3: `withdraw` [parent=f1]
- `amount` 25
- Return value 50

```python
def make_withdraw(balance):
    def withdraw(amount):
        balance -= amount
        return balance
    return withdraw

withdraw = make_withdraw(60)
withdraw(15)
```
Persistent Local State Using Environments

Global frame

make_withdraw

withdraw

func make_withdraw(balance) [parent=Global]

func withdraw(amount) [parent=f1]

f1: make_withdraw [parent=Global]

balance

withdraw

Return value

50

f2: withdraw [parent=f1]

amount

Return value

25

75

f3: withdraw [parent=f1]

amount

Return value

25

50

The parent frame contains the balance, the local state of the withdraw function.
Persistent Local State Using Environments

The parent frame contains the balance, the local state of the withdraw function.

Every call decreases the same balance by (a possibly different) amount.
Persistent Local State Using Environments

The parent frame contains the balance, the local state of the withdraw function.

All calls to the same function have the same parent.

Every call decreases the same balance by (a possibly different) amount.
Reminder: Local Assignment

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

diff = percent_difference(40, 50)
```

Global frame

```plaintext
func percent_difference(x, y) [parent=Global]

f1: percent_difference [parent=Global]

<table>
<thead>
<tr>
<th>x</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>50</td>
</tr>
<tr>
<td>difference</td>
<td>10</td>
</tr>
</tbody>
</table>
```
Reminder: Local Assignment

```python
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

f1: percent_difference [parent=Global]
    x 40
    y 50
    difference 10
```
Reminder: Local Assignment

```python
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.
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`

f1: percent_difference [parent=Global]

```
x  40
y  50
```

difference 10

Execution rule for assignment statements:
Reminder: Local Assignment

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

Non-Local Assignment & Persistent Local State
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

```python
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

```python
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."""
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
```

```python
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."""
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
```
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

```python
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

```python
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

```python
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.
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
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)
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.

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.

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.

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.

---

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

\[ x = 2 \]

<table>
<thead>
<tr>
<th>Status</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Many Meanings of Assignment Statements

\[
x = 2
\]

<table>
<thead>
<tr>
<th>Status</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>• No nonlocal statement</td>
<td></td>
</tr>
<tr>
<td>• &quot;x&quot; is not bound locally</td>
<td></td>
</tr>
</tbody>
</table>
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></td>
</tr>
</tbody>
</table>

$x = 2$
### 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></td>
</tr>
<tr>
<td>• No nonlocal statement</td>
<td></td>
</tr>
<tr>
<td>• &quot;x&quot; is bound locally</td>
<td></td>
</tr>
</tbody>
</table>

The picture represents an assignment statement `x = 2`.
### The Many Meanings of Assignment Statements

<table>
<thead>
<tr>
<th>Status</th>
<th>Effect</th>
</tr>
</thead>
</table>
| • 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

<table>
<thead>
<tr>
<th>Status</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>• No nonlocal statement • &quot;x&quot; is not bound locally</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>• No nonlocal statement • &quot;x&quot; is bound locally</td>
<td>Re-bind name &quot;x&quot; to object 2 in the first frame of the current environment</td>
</tr>
<tr>
<td>• nonlocal x • &quot;x&quot; is bound in a non-local frame</td>
<td></td>
</tr>
</tbody>
</table>
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 environment</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 which &quot;x&quot; is bound</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>
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></td>
</tr>
<tr>
<td>• No nonlocal statement</td>
<td>Re-bind name &quot;x&quot; to object 2 in the first frame of the current environment</td>
</tr>
<tr>
<td>&quot;x&quot; is bound locally</td>
<td></td>
</tr>
<tr>
<td>• nonlocal x</td>
<td>Re-bind &quot;x&quot; to 2 in the first non-local frame of the current environment in which &quot;x&quot; is bound</td>
</tr>
<tr>
<td>&quot;x&quot; is bound in a non-local frame</td>
<td></td>
</tr>
<tr>
<td>• nonlocal x</td>
<td></td>
</tr>
<tr>
<td>&quot;x&quot; is not bound in a non-local frame</td>
<td></td>
</tr>
</tbody>
</table>
### 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></td>
</tr>
<tr>
<td>• No nonlocal statement</td>
<td>Re-bind name &quot;x&quot; to object 2 in the first frame of the current environment</td>
</tr>
<tr>
<td>• &quot;x&quot; is bound locally</td>
<td></td>
</tr>
<tr>
<td>• nonlocal x</td>
<td>Re-bind &quot;x&quot; to 2 in the first non-local frame of the current environment in which &quot;x&quot; is bound</td>
</tr>
<tr>
<td>• &quot;x&quot; is bound in a non-local frame</td>
<td></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></td>
</tr>
</tbody>
</table>
## The Many Meanings of Assignment Statements

<table>
<thead>
<tr>
<th>Status</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>• No nonlocal statement • &quot;x&quot; is not bound locally</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>• No nonlocal statement • &quot;x&quot; is bound locally</td>
<td>Re-bind name &quot;x&quot; to object 2 in the first frame of the current environment</td>
</tr>
<tr>
<td>• nonlocal x • &quot;x&quot; is bound in a non-local frame</td>
<td>Re-bind &quot;x&quot; to 2 in the first non-local frame of the current environment in which &quot;x&quot; is bound</td>
</tr>
<tr>
<td>• nonlocal x • &quot;x&quot; is not bound in a non-local frame</td>
<td>SyntaxError: no binding for nonlocal 'x' found</td>
</tr>
<tr>
<td>• nonlocal x • &quot;x&quot; is bound in a non-local frame • &quot;x&quot; also bound locally</td>
<td></td>
</tr>
</tbody>
</table>
### The Many Meanings of Assignment Statements

<table>
<thead>
<tr>
<th>Status</th>
<th>Effect</th>
</tr>
</thead>
</table>
| • 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.

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

wd = make_withdraw(20)
w(5)
```
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
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. Within the body of a function, all instances of a name must refer to the same frame.

```python
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)
```

UnboundLocalError: local variable 'balance' referenced before assignment
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.

```python
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 \textit{without} a nonlocal statement.

```python
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.

```python
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 \textit{without} a nonlocal statement.

```python
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.

goo.gl/y4TyFZ
Mutable Values & Persistent Local State

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

```python
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)
```

Name bound outside of withdraw def

Element assignment changes a list

Name-value binding cannot change because there is no nonlocal statement
Multiple Mutable Functions

(Demo)
Referential Transparency, Lost

```python
def f(x):
    x = 4
    def g(y):
        return x + y + z
    return h
return g
a = f(1)
b = a(2)
total = b(3) + b(4)
```
Referential Transparency, Lost

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

```python
mul(add(2, mul(4, 6)), add(3, 5))
```
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))
```
Expressions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.

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

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

\[
\text{mul}(26, \text{add}(3, 5))
\]
Expressions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.

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

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

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

Mutation operations violate the condition of referential transparency because they do more than just return a value; they change the environment.
Referential Transparency, Lost

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

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

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

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

- Mutation operations violate the condition of referential transparency because they do more than just return a value; **they change the environment**.
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**.
Environment Diagrams
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)
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)
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)
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)
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)
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)
Go Bears!

```python
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)
```

Go Bears!

```python
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)
```

```
Return Value
```

Global frame

```
Global frame
oski
```

1. `func oski(bear)[parent=G]`
2. `func abs(...)[parent=G]`
3. `func cal(berk)[parent=f1]`

```
f1: oski [parent=G]
    bear
    cal
    Return Value
```

```
f2: cal [parent=f1]
    Return Value
```

```
Return Value
```

```
Return Value
```

```
Return Value
```

```
Return Value
```
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)
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)
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)
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)
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)
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)
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)
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)
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)
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)
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)
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)
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)
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)
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)
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)
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)
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)
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)
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)
Go Bears!

```python
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)
```

Diagram:

- **Global frame**
  - `oski`

- **Frame f1**
  - `oski`
    - `bear`
    - `cal`
    - Return Value
  - **Frame f2**
    - `cal`
      - `berk` 2
      - Return Value
    - **Frame f3**
      - `berk` 2
      - Return Value
    - **Frame f4**
      - `ley` 2
      - Return Value 0

- `func oski(bear)[parent=G]`
- `func cal(berk)[parent=f1]`
- `func λ(ley)[parent=f2]`
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)
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)
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)
Go Bears!

```python
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)
```
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)