A Function with Behavior That Varies Over Time

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

```python
>>> 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
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
Let's model a bank account that has a balance of $100.

Return value: remaining balance

>>> withdraw(25)
75

Different return value!

>>> 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

Different return value!

>>> withdraw(25)
75

>>> withdraw(25)
50

Argument: amount to withdraw

Second withdrawal of the same amount

>>> withdraw(60)
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

>>> withdraw(60)
'Insufficient funds'

Second withdrawal of the same amount
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)
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

Second withdrawal of the same amount

Different return value!

>>> withdraw(25)
75

>>> withdraw(25)
50

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35
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

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

Second withdrawal of the same amount

Different return value!

Where's this balance stored?

>>> withdraw(25)
75

>>> withdraw(25)
50

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35

>>> 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

>>> withdraw(25)
75

>>> withdraw(25)
50

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35

>>> withdraw = make_withdraw(100)

Argument: amount to withdraw

Second withdrawal of the same amount

Where's this balance stored?

Within the function!
Persistent Local State

Global frame

func make_withdraw(balance)

make_withdraw
withdraw

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

A function with a parent frame
Persistent Local State

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
Reminder: Local Assignment

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

diff = percent_difference(40, 50)
```

Example: [http://goo.gl/wcF71](http://goo.gl/wcF71)
Reminder: Local Assignment

```python
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](http://goo.gl/wcF71)
Reminder: Local Assignment

```python
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
```

Local frame
```
percent_difference
  x  40
  y  50
difference  10
```

Execution rule for assignment statements:

Example: [http://goo.gl/wcF71](http://goo.gl/wcF71)
Reminder: Local Assignment

```python
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

<table>
<thead>
<tr>
<th>Global frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>percent_difference</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>percent_difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 40</td>
</tr>
<tr>
<td>y 50</td>
</tr>
<tr>
<td>difference 10</td>
</tr>
</tbody>
</table>

**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](http://goo.gl/wcF71)
Non-Local Assignment & Persistent Local State
Non-Local Assignment & Persistent Local State

def make_withdraw(balance):

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
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:
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  # Declare the name "balance" nonlocal
    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 withdraw

Declare the name "balance" nonlocal
Re-bind balance where it was bound previously
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
```

Declare the name "balance" nonlocal
Re-bind balance where it was bound previously
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
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.
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.

From the Python 3 language reference:
The Effect of Nonlocal Statements

\texttt{nonlocal <name>, <name 2>, ...}

\textbf{Effect}: Future references to that name refer to its pre-existing binding in the \textit{first non-local frame} of the current environment in which that name is bound.

\textit{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

```python
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.

*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.

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

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.

nonlocal <name>, <name 2>, ...

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>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<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

```
x = 2
```

<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</td>
</tr>
<tr>
<td>• &quot;x&quot; is not bound locally</td>
<td>the current environment.</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</td>
</tr>
<tr>
<td>• &quot;x&quot; is not bound locally</td>
<td>the current environment.</td>
</tr>
</tbody>
</table>

<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</td>
</tr>
<tr>
<td>• &quot;x&quot; is bound locally</td>
<td>the current environment.</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</td>
</tr>
<tr>
<td>• &quot;x&quot; is not bound locally</td>
<td>the current environment.</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 env.</td>
</tr>
<tr>
<td>• &quot;x&quot; is 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; <strong>is not</strong> 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 env.</td>
</tr>
<tr>
<td>• &quot;x&quot; <strong>is</strong> bound locally</td>
<td></td>
</tr>
<tr>
<td>• nonlocal x</td>
<td></td>
</tr>
<tr>
<td>• &quot;x&quot; <strong>is bound in a non-local frame</strong></td>
<td></td>
</tr>
</tbody>
</table>
## The Many Meanings of Assignment Statements

### Status

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

### Effect

| Create a new binding from name "x" to object 2 in the first frame of the current environment. |
| Re-bind name "x" to object 2 in the first frame of the current env. |
| 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

<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 env.</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 it is bound.</td>
</tr>
<tr>
<td>• nonlocal x • &quot;x&quot; is not bound in a non-local frame</td>
<td></td>
</tr>
</tbody>
</table>
## The Many Meanings of Assignment Statements

### Status

<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; <strong>is not</strong> 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 env.</td>
</tr>
<tr>
<td>• &quot;x&quot; <strong>is bound locally</strong></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 it is bound.</td>
</tr>
<tr>
<td>• &quot;x&quot; <strong>is bound in a non-local frame</strong></td>
<td></td>
</tr>
<tr>
<td>• no nonlocal x</td>
<td>SyntaxError: no binding for nonlocal 'x' found</td>
</tr>
<tr>
<td>• &quot;x&quot; <strong>is not bound in a non-local frame</strong></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 env.</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 it 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>
<tr>
<td>• nonlocal x</td>
<td></td>
</tr>
<tr>
<td>• &quot;x&quot; is bound in a non-local frame</td>
<td></td>
</tr>
<tr>
<td>• &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 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 |

\[
x = 2
\]
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.

```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.

Therefore, 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)
```

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.

```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.

```
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

def withdraw(amount):
    withdraw = make_withdraw_list(100)
    withdraw(25)
```
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)
```
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**.

<table>
<thead>
<tr>
<th>John's Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>John's Account</th>
<th>Steven's Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10</td>
<td>$1,000,000</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>John's Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>John's Account</th>
<th>Steven's Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10</td>
<td>$10</td>
</tr>
</tbody>
</table>
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.
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))
```
Expressions are \textit{referentially transparent} if substituting an expression with its value does not change the meaning of a program.

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

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

• 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.

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

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

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

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

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

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

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

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

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

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

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

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

\[
\text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5))
\]
\[
\text{mul}(\text{add}(2, 24), \text{add}(3, 5))
\]
\[
\text{mul}(26, \text{add}(3, 5))
\]

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