# Objects

(Demo1)

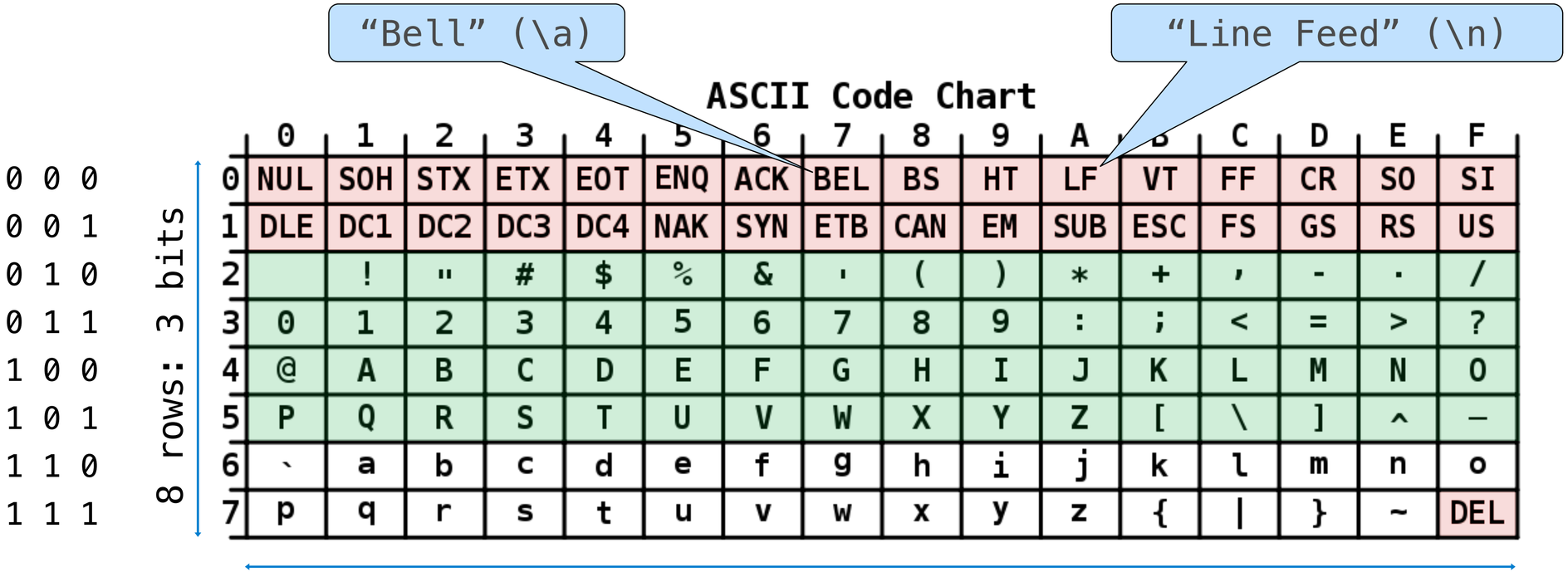
# Objects

- Objects represent information
- They consist of data and behavior, bundled together to create abstractions
- Objects can represent things, but also properties, interactions, & processes
- A type of object is called a class; **classes** are first-class values in Python
- Object-oriented programming:
  - A metaphor for organizing large programs
  - Special syntax that can improve the composition of programs
- In Python, every value is an object
  - All **objects** have **attributes**
  - A lot of data manipulation happens through object **methods**
  - Functions do one thing; objects do many related things

# Example: Strings

# Representing Strings: the ASCII Standard

American Standard Code for Information Interchange




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 0 0 | 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI |
| 0 0 1 | 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 0 1 0 | 2 | | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 0 1 1 | 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 1 0 0 | 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 1 0 1 | 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 1 1 0 | 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 1 1 1 | 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | DEL |

- Layout was chosen to support sorting by character code
- Rows indexed 2–5 are a useful 6-bit (64 element) subset
- Control characters were designed for transmission

(Demo2)

https://en.wikipedia.org/wiki/List_of_Unicode_characters

# Representing Strings: the Unicode Standard

- 137K characters
- 146 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order
- A canonical name for every character


http://ian-albert.com/unicode_chart/unichart-chinese.jpg

LATIN CAPITAL LETTER A

BASKETBALL AND HOOP

EIGHTH NOTE

(Demo3)

# Mutation Operations

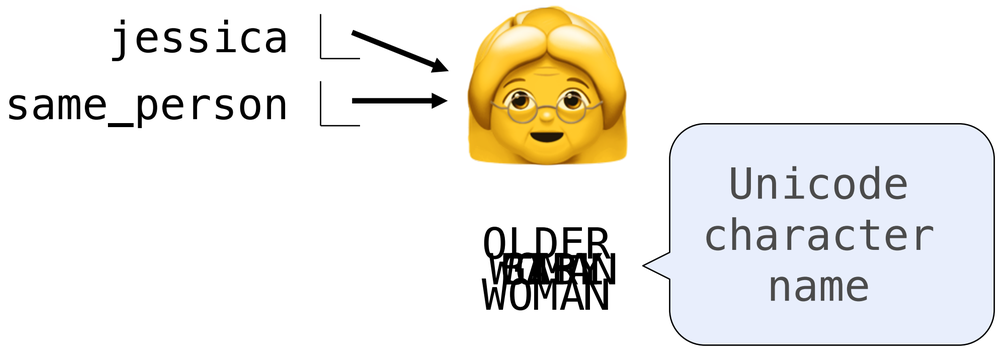
https://en.wikipedia.org/wiki/Playing_card_suit

# Some Objects Can Change

[Demo4]

First example in the course of an object changing state

The same object can change in value throughout the course of computation




All names that refer to the same object are affected by a mutation

Only objects of *mutable* types can change: lists & dictionaries

{Demo5}

# Mutation Can Happen Within a Function Call

A function can change the value of any object in its scope

```
>>> four = [1, 2, 3, 4]
>>> len(four)
4
>>> mystery(four)
>>> len(four)
2
>>> four = [1, 2, 3, 4]
>>> len(four)
4
>>> another_mystery() # No arguments!
>>> len(four)
2
```

```
def mystery(s):
    s.pop()
    s.pop()
```

*or*

```
def mystery(s):
    s[2:] = []
```

```
def another_mystery():
    four.pop()
    four.pop()
```



pythontutor.com/composingprograms.html#code=def%20mystery%28s%29%3A%0A%20%20%20%20s.pop%28%29%0A%20%20%20%20s.pop%28%29%0A%0Afour%20%3D%20[1,%202,%203,%204]%0Amystery%28four%29&mode=display&origin=composingprograms.js&cumulative=true&py=3&rawInputLstJSON=[]&curInstr=0

# Tuples

(Demo6)

# Tuples are Immutable Sequences

Immutable values are protected from mutation

```
>>> turtle = (1, 2, 3)
>>> ooze()
>>> turtle
(1, 2, 3)
```

Next lecture: ooze can change turtle's binding

```
>>> turtle = [1, 2, 3]
>>> ooze()
>>> turtle
['Anything could be inside!']
```

The value of an expression can change because of changes in names or objects

**Name change:**

```
>>> x = 2
>>> x + x
4
>>> x = 3
>>> x + x
6
```

**Object mutation:**

```
>>> x = [1, 2]
>>> x + x
[1, 2, 1, 2]
>>> x.append(3)
>>> x + x
[1, 2, 3, 1, 2, 3]
```

An immutable sequence may still change if it *contains* a mutable value as an element

```
>>> s = ([1, 2], 3)
>>> s[0] = 4
ERROR
```

```
>>> s = ([1, 2], 3)
>>> s[0][0] = 4
>>> s
([4, 2], 3)
```

# Mutation

# Sameness and Change

- As long as we never modify objects, a compound object is just 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
- A compound data object has an "identity" in addition to the pieces of which it is composed
- A list is still "the same" list even if we change its contents
- Conversely, we could have two lists that happen to have the same contents, but are different

```
>>> a = [10]
>>> b = a
>>> a == b
True
>>> a.append(20)
>>> a
[10, 20]
>>> b
[10, 20]
>>> a == b
True
```

```
>>> a = [10]
>>> b = [10]
>>> a == b
True
>>> b.append(20)
>>> a
[10]
>>> b
[10, 20]
>>> a == b
False
```

# Identity Operators

**Identity**

`<exp0> is <exp1>`

evaluates to True if both <exp0> and <exp1> evaluate to the same object

**Equality**

`<exp0> == <exp1>`

evaluates to True if both <exp0> and <exp1> evaluate to equal values

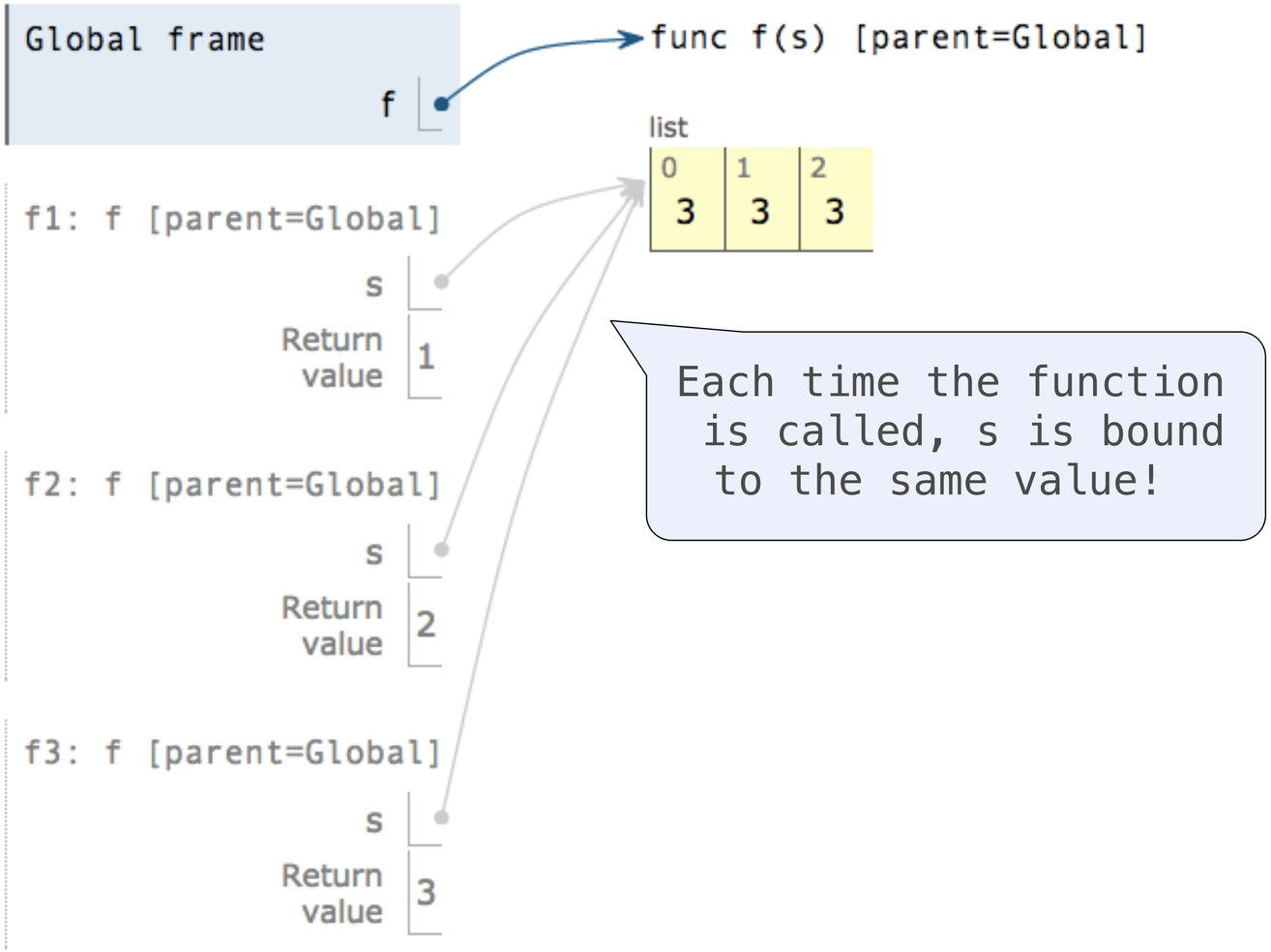
**Identical objects are always equal values**

(Demo)

# Mutable Default Arguments are Dangerous

A default argument value is part of a function value, not generated by a call

```
>>> def f(s=[]):
...     s.append(3)
...     return len(s)
...
>>> f()
1
>>> f()
2
>>> f()
3
```



# Mutable Functions

# Functions with behavior that changes over time

```
def square(x):
    return x * x


>>> square(5)
25
>>> square(5)
25
>>> square(5)
25
```

Returns the same value when called with the same input

```
def f(x):
   ...


>>> f(5)
25
>>> f(5)
26
>>> f(5)
27
```

Return value is different when called with the same input

# Example - Withdraw

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

```
>>> withdraw = make_withdraw(100)
>>> withdraw(25)
75
>>> withdraw(25)
50
>>> withdraw(60)
'Insufficient fund
>>> withdraw(15)
35
```

Argument: amount to withdraw

Return value: remaining balance

Second withdrawal of the same amount

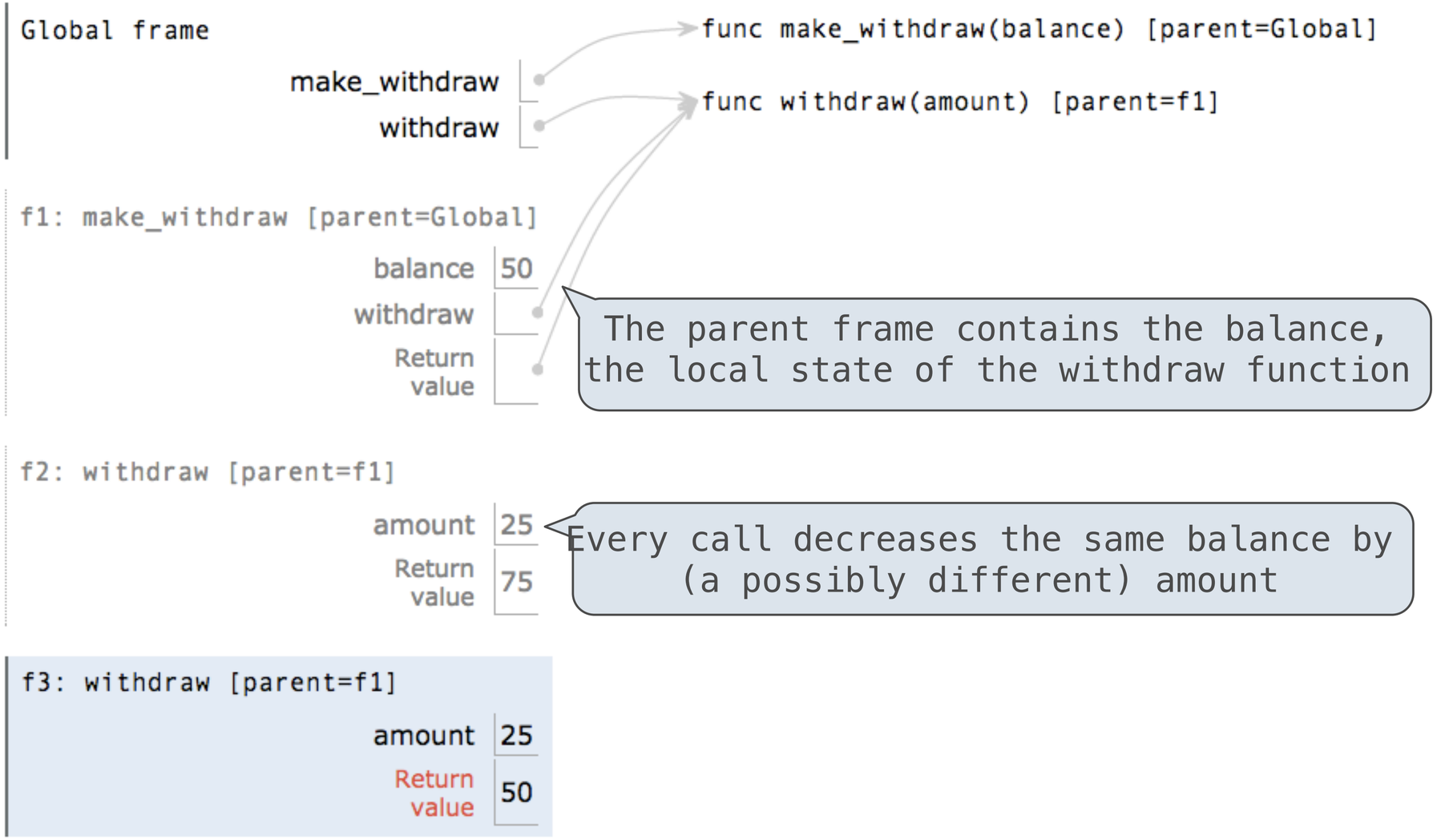
Different return value!

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



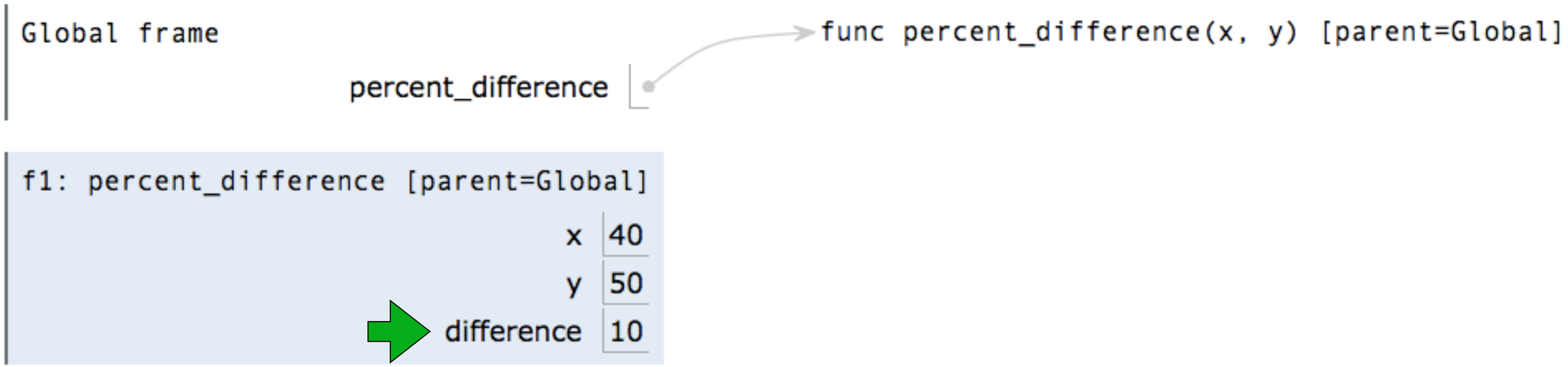



pythontutor.com/composingprograms.html#code=def%20make_withdraw%28balance%29%3A%0A%20%20%20%20def%20withdraw%28amount%29%3A%0A%20%20%20%20%20%20%20%20nonlocal%20balance%0A%20%20%20%20%20%20%20%20if%20amount%20>%20balance%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20return%20'Insufficient%20funds'%0A%20%20%20%20%20%20%20%20balance%20%3D%20balance%20-%20amount%0A%20%20%20%20

# Reminder: Local Assignment

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

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




**Execution rule for assignment statements:**

1. Evaluate all expressions right of =, from left to right

2. Bind the names on the left to the resulting values in the **current frame**

pythontutor.com/composingprograms.html#code=def%20percent_difference%28x,%20y%29%3A%0A%20%20%20%20difference%20%3D%20abs%28x-y%29%0A%20%20%20%20return%20100%20*%20difference%20/%20x%0Adiff%20%3D%20percent_difference%2840,%2050%29&mode=display&origin=composingprograms.js&cumulative=true&py=3&rawInputLstJSON=[]&curInstr=4

# Non-Local Assignment & Persistent Local State

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

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

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

(Demo)

# Non-Local Assignment

# The Effect of Nonlocal Statements

```
nonlocal <name>, <name>, ...
```

**Effect:** Future assignments to that name change its pre-existing binding in the **first non-local frame** of the current environment in which that name is bound.

Python Docs: an "enclosing scope"

**From the Python 3 language reference:**

Names listed in a nonlocal statement must refer to pre-existing bindings in an enclosing scope.

Names listed in a nonlocal statement must not collide with pre-existing bindings in the local scope.

Current frame

http://docs.python.org/release/3.1.3/reference/simple_stmts.html#the-nonlocal-statement

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

# The Many Meanings of Assignment Statements

`x = 2`

| Status | Effect |
| --- | --- |
| • No nonlocal statement<br>• "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<br>• "x" **is** bound locally | Re-bind name "x" to object 2 in the first frame of the current environment |
| • nonlocal x<br>• "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<br>• "x" **is not** bound in a non-local frame | SyntaxError: no binding for nonlocal 'x' found |
| • nonlocal x<br>• "x" **is** bound in a non-local frame<br>• "x" also bound locally | SyntaxError: name 'x' is parameter and nonlocal |

# Python Particulars

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

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

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

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

Local assignment

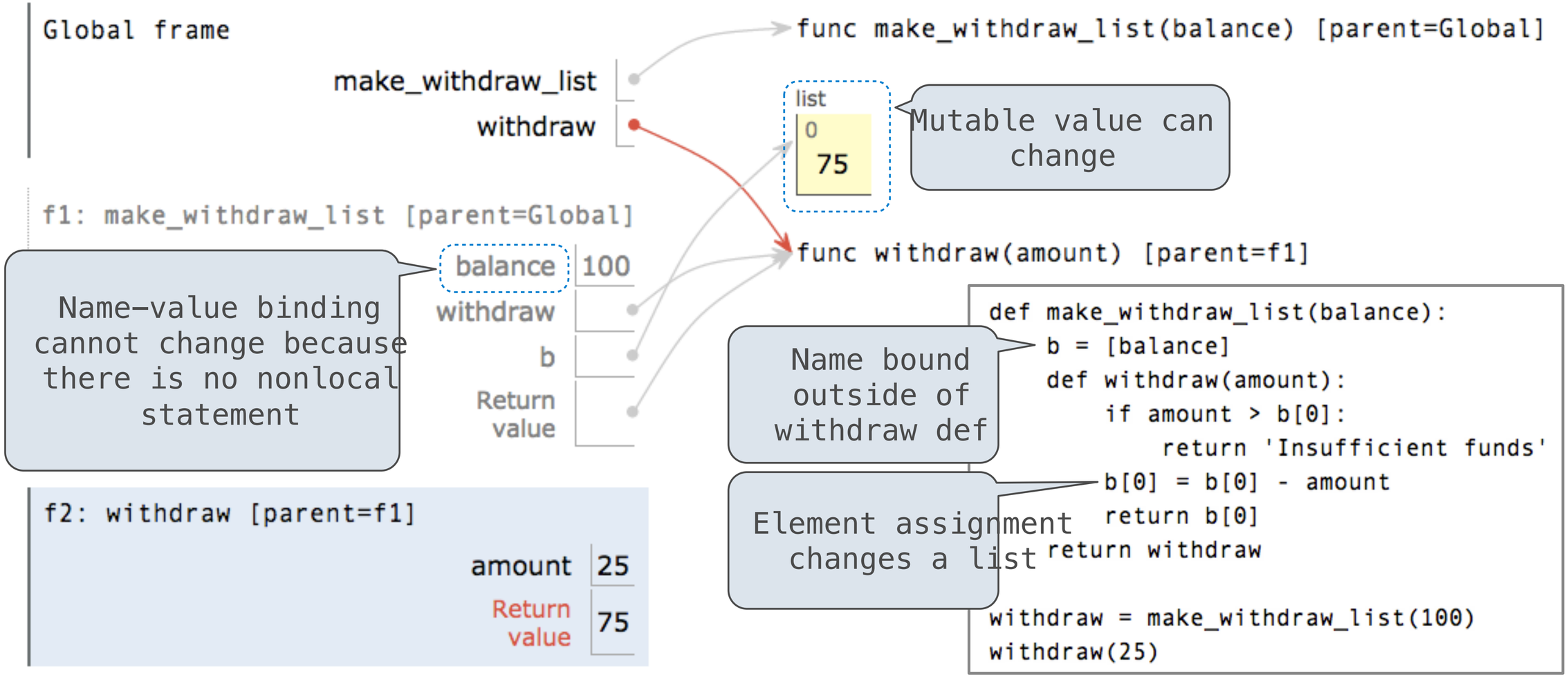
UnboundLocalError: local variable 'balance' referenced before assignment



pythontutor.com/composingprograms.html#code=def%20make_withdraw%28balance%29%3A%0A%20%20%20%20def%20withdraw%28amount%29%3A%0A%20%20%20%20%20%20%20%20if%20amount%20>%20balance%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20return%20'Insufficient%20funds'%0A%20%20%20%20%20%20%20%20balance%20%3D

# Mutable Values & Persistent Local State

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




goo.gl/y4TyFZ

# Multiple Mutable Functions

(Demo)

# Referential Transparency, Lost

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

```
mul(add(2, mul(4, 6)), add(3, 5))

mul(add(2,    24    ), add(3, 5))

mul(       26         , add(3, 5))
```

- Mutation operations violate the condition of referential transparency because they do more than just return a value; **they change the environment.**

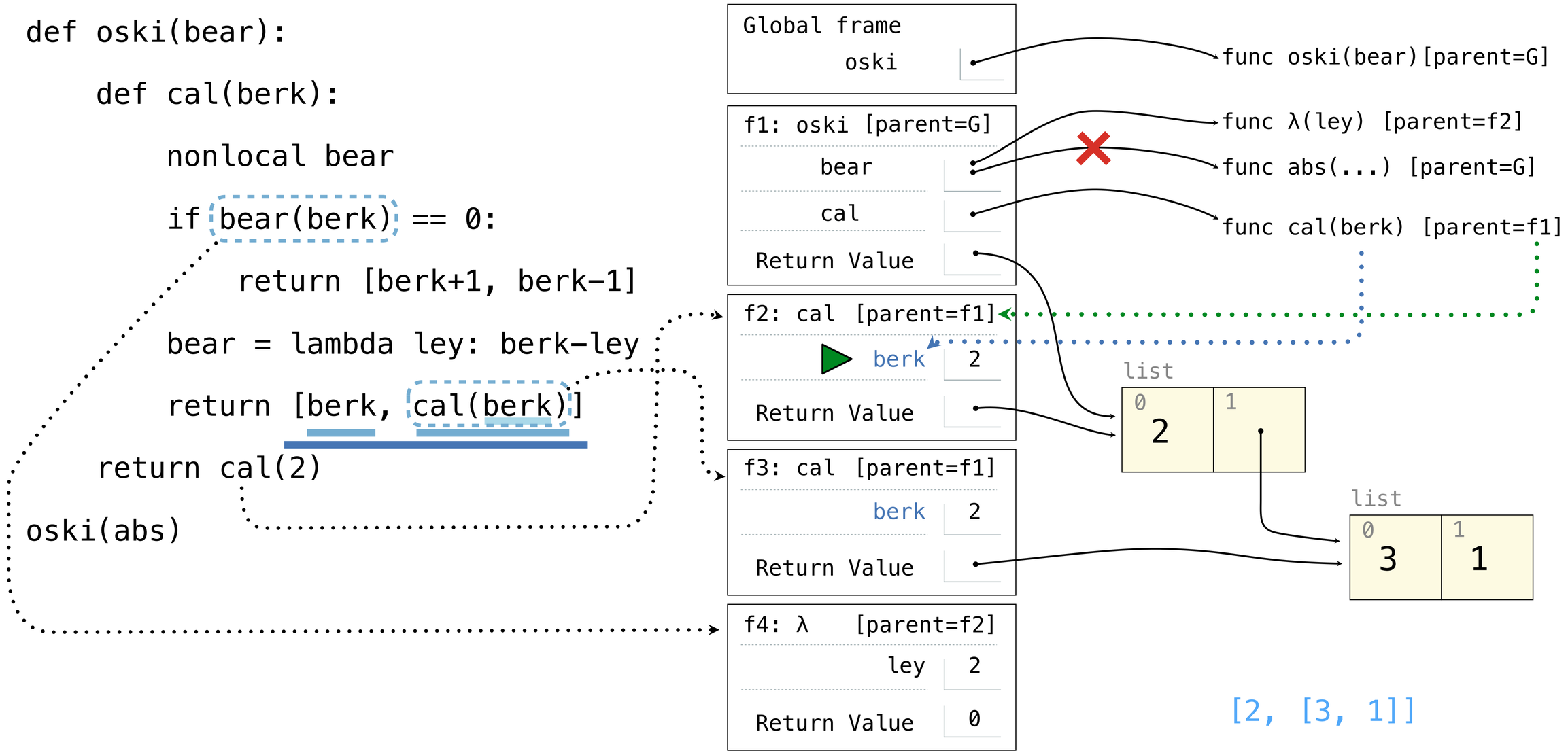


pythontutor.com/composingprograms.html#code=def%20f%28x%29%3A%0A%20%20%20%20x%20%3D%204%0A%20%20%20%20def%20g%28y%29%3A%0A%20%20%20%20%20%20%20%20def%20h%28z%29%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20nonlocal%20x%0A%20%20%20%20%20%20%20%20%20%20%20%20x%20%3D%20x%20%2B%201%0A%20%20%20%20%20%20%20%20%20%20%20%20return%20x%20%2B%20y%20%2B%20z%0A%20%20%20%20%20%20%20%20return%20h%0A%20%20%20%20

# Environment Diagrams

# Go Bears!

```
def oski(bear):
    def cal(berk):
        nonlocal bear
        if bear(berk) == 0:
            return [berk+1, berk-1]
        bear = lambda ley: berk-ley
        return [berk, cal(berk)]
    return cal(2)
oski(abs)
```



# Summary

- Nonlocal allows for functions whose behavior changes over time
- When declaring a variable nonlocal, we move part of the function's local state to its parent
- There are various rules for which variables may be declared nonlocal
- Nonlocal gives us a new type of assignment, where we change the binding in a parent instead