# Mutability

# Announcements

# Objects

(Demo)

# Objects

# Objects

- Objects represent information

# Objects

- Objects represent information

- They consist of data and behavior, bundled together to create abstractions

# Objects

- Objects represent information

- They consist of data and behavior, bundled together to create abstractions

- Objects can represent things, but also properties, interactions, & processes

# 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

# 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:

# 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

# 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

# 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

# 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**

# 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**

# 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

(Demo)

# Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

## ASCII Code Chart

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

6

# Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

### ASCII Code Chart

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

8 rows: 3 bits

# Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

### ASCII Code Chart

| | | | 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 | bits | 2 | | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 0 1 1 | 3 | 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 | rows | 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 1 1 0 | 8 | 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 |

# Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

## ASCII Code Chart

| | | 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 |

8 rows: 3 bits

16 columns: 4 bits

# Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

**ASCII Code Chart**

| | | 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 |

8 rows: 3 bits

16 columns: 4 bits

- Layout was chosen to support sorting by character code

# Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

## ASCII Code Chart

| 3 bits | 8 rows | | 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 |

16 columns: 4 bits

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

# Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

## ASCII Code Chart



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

Row index (8 rows: 3 bits):
0 0 0 → 0
0 0 1 → 1
0 1 0 → 2
0 1 1 → 3
1 0 0 → 4
1 0 1 → 5
1 1 0 → 6
1 1 1 → 7

16 columns: 4 bits

- 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

# Representing Strings: the ASCII Standard

American Standard Code for Information Interchange



**ASCII Code Chart**

"Line feed" (\n)

|       |   | 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 |

8 rows: 3 bits

16 columns: 4 bits

- 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

# Representing Strings: the ASCII Standard

American Standard Code for Information Interchange



ASCII Code Chart

"Bell" (\a)

"Line feed" (\n)

16 columns: 4 bits

- 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

# Representing Strings: the ASCII Standard

American Standard Code for Information Interchange



16 columns: 4 bits

- 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

(Demo)

# Representing Strings: the Unicode Standard

# Representing Strings: the Unicode Standard



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

# Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1



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

# Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1
- 150 scripts (organized)



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

# Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1

- 150 scripts (organized)

- Enumeration of character properties,
  such as case



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

# Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1

- 150 scripts (organized)

- Enumeration of character properties, such as case

- Supports bidirectional display order



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

# Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1

- 150 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

# Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1

- 150 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

# Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1

- 150 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

DIE FACE-6

# Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1

- 150 scripts (organized)

- Enumeration of character properties, such as case

- Supports bidirectional display order
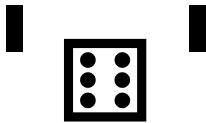
- A canonical name for every character



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

LATIN CAPITAL LETTER A

DIE FACE-6

EIGHTH NOTE

# Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1

- 150 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

DIE FACE-6

EIGHTH NOTE

# Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1

- 150 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

DIE FACE-6

EIGHTH NOTE

# Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1

- 150 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

DIE FACE-6

EIGHTH NOTE

(Demo)

# Mutation Operations

# Some Objects Can Change

[Demo]

# Some Objects Can Change

[Demo]

First example in the course of an object changing state

# Some Objects Can Change

[Demo]

First example in the course of an object changing state

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

# Some Objects Can Change

[Demo]

First example in the course of an object changing state

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

same_person └──→ 👶

# Some Objects Can Change

[Demo]

First example in the course of an object changing state

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

same_person └─⟶ 👶

BABY

# Some Objects Can Change

[<u>Demo</u>]

First example in the course of an object changing state

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



same_person

Unicode character name

BABY

# Some Objects Can Change

[Demo]

First example in the course of an object changing state

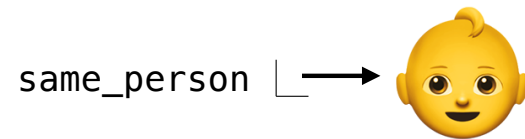The same object can change in value throughout the course of computation

same_person └──→ 👧

GIRL ◁ Unicode character name

# Some Objects Can Change

[Demo]

First example in the course of an object changing state

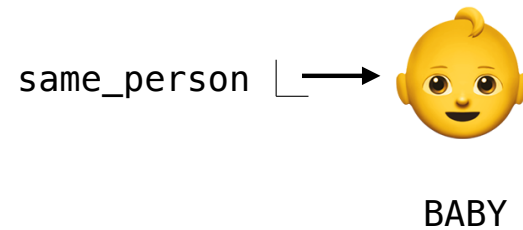The same object can change in value throughout the course of computation

# Some Objects Can Change

[Demo]

First example in the course of an object changing state

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

# Some Objects Can Change

[Demo]

First example in the course of an object changing state

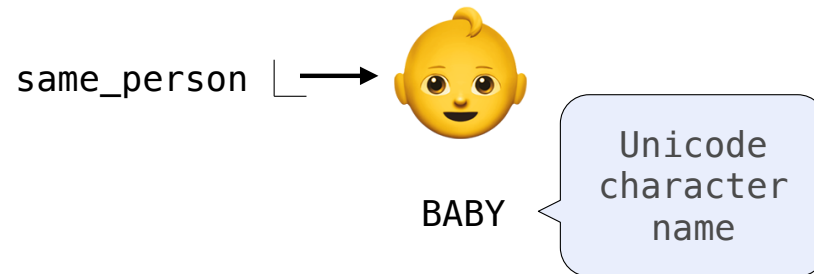The same object can change in value throughout the course of computation

## Some Objects Can Change

[Demo]

First example in the course of an object changing state

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



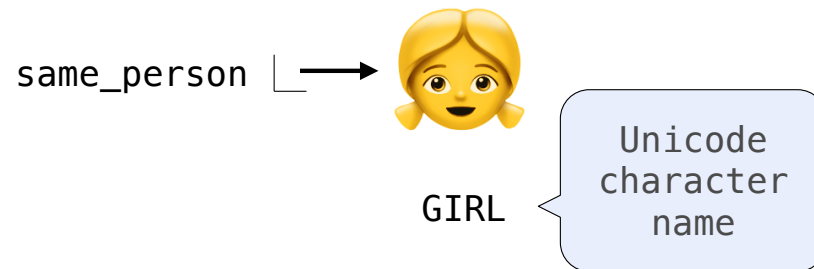All names that refer to the same object are affected by a mutation

## Some Objects Can Change

[Demo]

First example in the course of an object changing state

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



jessica
same_person

OLDER WOMAN

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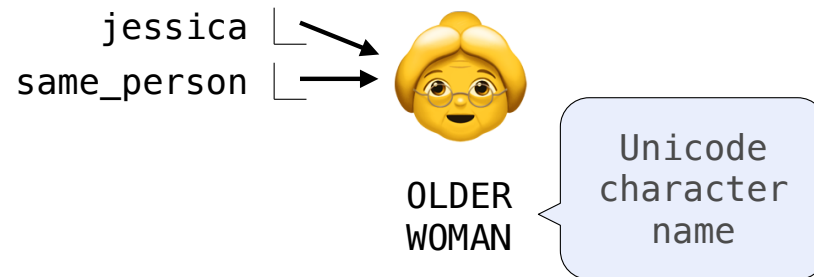
All names that refer to the same object are affected by a mutation

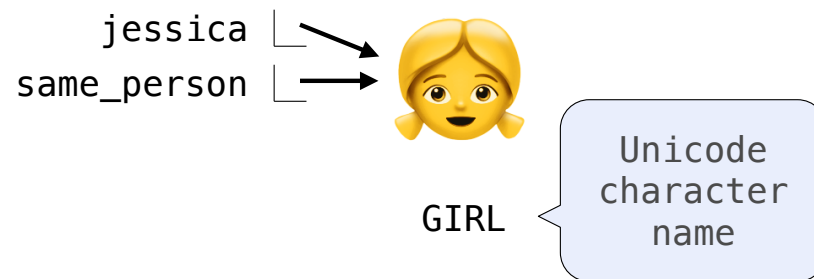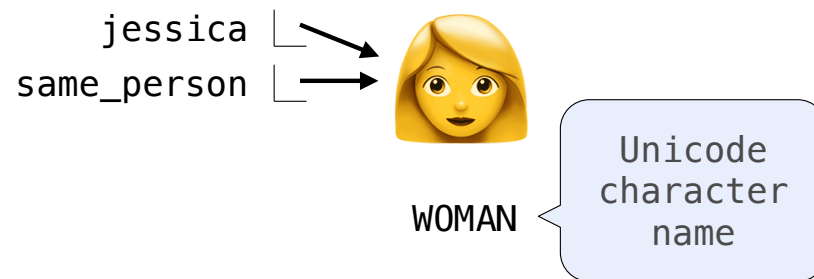Only objects of *mutable* types can change: lists & dictionaries

## Some Objects Can Change

[Demo]

First example in the course of an object changing state

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

jessica

same_person

OLDER
WOMAN

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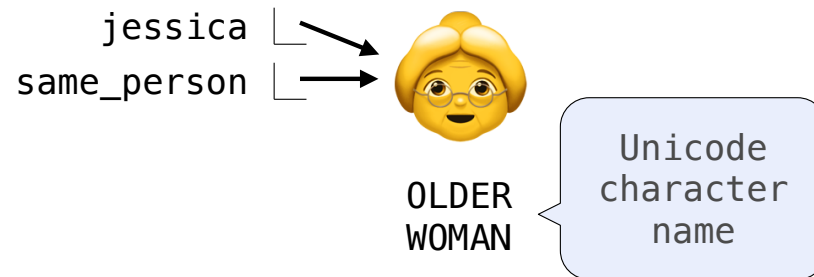
All names that refer to the same object are affected by a mutation

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

{Demo}

# Mutation Can Happen Within a Function Call

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

# Mutation Can Happen Within a Function Call

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

```
>>> four = [1, 2, 3, 4]
```

# 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
```

# 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)
```

# 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
```

# 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
```

```
def mystery(s):
    s.pop()
    s.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%0A%0Afour%20%3D%20[1,%202,%203,%204]%0Amystery%28four%29&mode=display&origin=composingprograms.js&cumulative=true&py=3&rawInputLstJSON=[]&curInstr=0

# 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
```

```
def mystery(s):    or    def mystery(s):
    s.pop()                   s[2:] = []
    s.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

# 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]
```

```
def mystery(s):        or    def mystery(s):
    s.pop()                      s[2:] = []
    s.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

# 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
```

```
def mystery(s):        or   def mystery(s):
    s.pop()                     s[2:] = []
    s.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

# 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
```

```
def mystery(s):   or   def mystery(s):
    s.pop()                 s[2:] = []
    s.pop()
```

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

# 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
```

```
def mystery(s):    or   def mystery(s):
    s.pop()                  s[2:] = []
    s.pop()
```

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

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

# 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
```

```
def mystery(s):    or   def mystery(s):
    s.pop()                 s[2:] = []
    s.pop()
```

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

(Demo)

# Tuples are Immutable Sequences

# Tuples are Immutable Sequences

Immutable values are protected from mutation

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

# 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!']
```

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

Next lecture: ooze can change turtle's binding

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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

> Next lecture: ooze can change turtle's binding

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

**Name change:**

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

Next lecture: ooze can change turtle's binding

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

```
>>> x + x
```

**Name change:**

```
>>> x + x
```

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

Next lecture: ooze can change turtle's binding

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

```
>>> x = 2
>>> x + x
```

**Name change:**

```
>>> x + x
```

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

Next lecture: ooze can change turtle's binding

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

**Name change:**
```
>>> x = 2
>>> x + x
4

>>> x + x
```

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

Next lecture: ooze can
change turtle's binding

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
```

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

> Next lecture: ooze can change turtle's binding

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
```

# 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:**

# 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 + x
```

```
>>> x + x
```

# 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

>>> x + x
```

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

> Next lecture: ooze can change turtle's binding

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 + x
```

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

> Next lecture: ooze can change turtle's binding

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
```

# 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]
```

# 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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

> Next lecture: ooze can change turtle's binding

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)
```

# Tuples are Immutable Sequences

Immutable values are protected from mutation

```
>>> turtle = (1, 2, 3)              >>> turtle = [1, 2, 3]
>>> ooze()                           >>> ooze()
>>> turtle   Next lecture: ooze can  >>> turtle
(1, 2, 3)    change turtle's binding ['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
```

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

Next lecture: ooze can change turtle's binding

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
```

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

Next lecture: ooze can change turtle's binding

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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

Next lecture: ooze can change turtle's binding

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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

> Next lecture: ooze can change turtle's binding

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

# Tuples are Immutable Sequences

Immutable values are protected from mutation

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

Next lecture: ooze can change turtle's binding

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

# Mutation

# Sameness and Change

# Sameness and Change

- As long as we never modify objects, a compound object is just the totality of its pieces

## 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

# 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

# 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

# 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

# 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

```
>>> a = [10]
```

# 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

```
>>> a = [10]
>>> b = a
```

# 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

```
>>> a = [10]
>>> b = a
>>> a == b
True
```

## 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

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

# 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

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

# 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

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

# 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

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

# 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
```

# 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]
```

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

# 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
```

# 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)
```

# 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]
```

# 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]
```

# 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 Operators

**Identity**

<exp0> **is** <exp1>

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

# 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

# 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**

# 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

# Mutable Default Arguments are Dangerous

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

# 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)
...
```
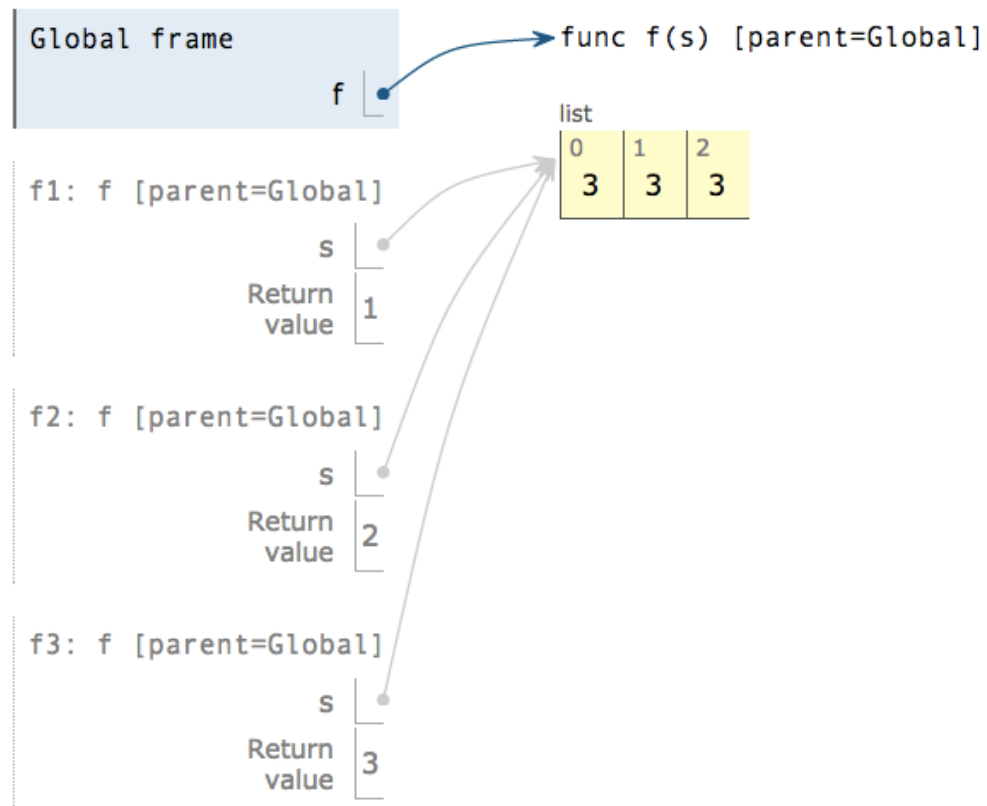
# Mutable Default Arguments are Dangerous

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
```

# 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
```

# Mutable Default Arguments are Dangerous

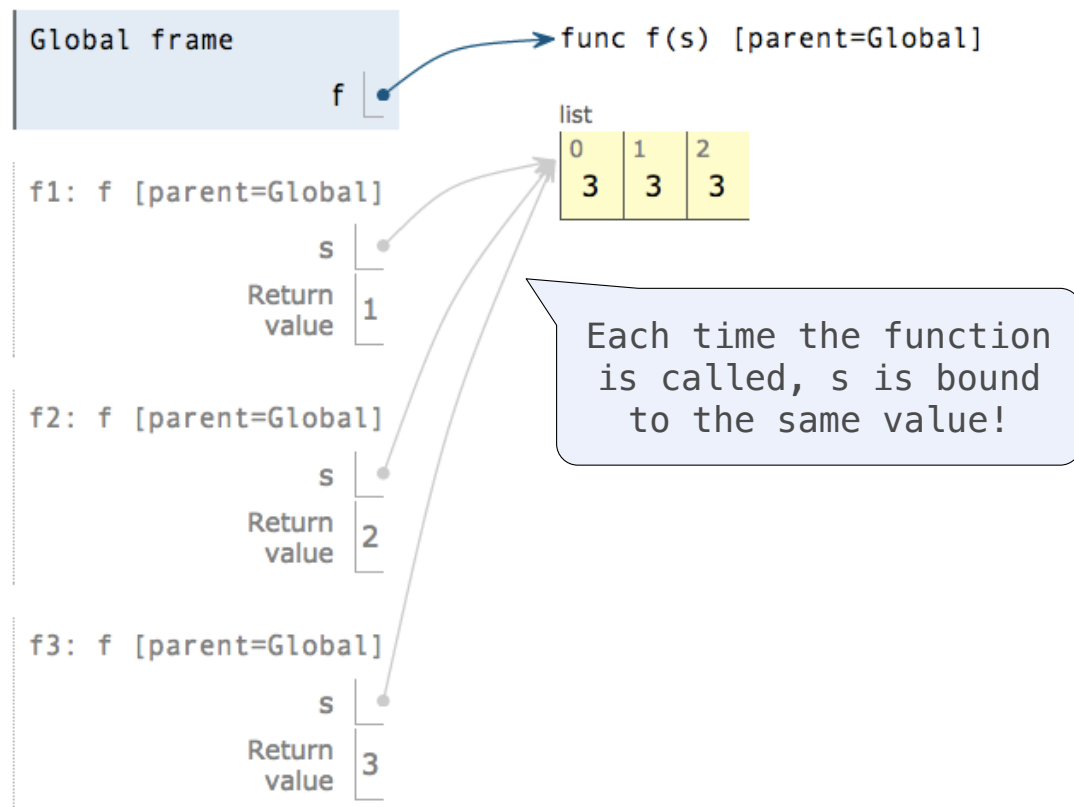A default argument value is part of a function value, not generated by a call

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

# Mutable 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
```

Global frame

f → func f(s) [parent=Global]

list
| 0 | 1 | 2 |
|---|---|---|
| 3 | 3 | 3 |

f1: f [parent=Global]
s
Return value  1

f2: f [parent=Global]
s
Return value  2

f3: f [parent=Global]
s
Return value  3

# 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
```

Global frame

f → func f(s) [parent=Global]

list

| 0 | 1 | 2 |
|---|---|---|
| 3 | 3 | 3 |

f1: f [parent=Global]

s

Return value  1

f2: f [parent=Global]

s

Return value  2

f3: f [parent=Global]

s

Return value  3

> Each time the function is called, s is bound to the same value!

# Mutable Functions

# A Function with Behavior That Varies Over Time

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

# A Function with Behavior That Varies Over Time

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

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

# A Function with Behavior That Varies Over Time

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

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

# A Function with Behavior That Varies Over Time

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

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

Argument:
amount to withdraw

# A Function with Behavior That Varies Over Time

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

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

Argument:
amount to withdraw

```
>>> withdraw(25)
75

>>> withdraw(25)
50
```

# 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

```
>>> 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

# 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

```
>>> 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
```

Argument:
amount to withdraw

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

Second withdrawal of
the same amount

Different
return value!

```
>>> 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
```

Argument:
amount to withdraw

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

Second withdrawal of
the same amount

Different
return value!

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

```
>>> withdraw(15)
35
```

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

```
>>> withdraw = make_withdraw_list(100)

>>> withdraw(25)
75

>>> withdraw(25)
50

>>> withdraw(60)
'Insufficient funds'

>>> withdraw(15)
35
```

Return value:
remaining balance

Different
return value!

Argument:
amount to withdraw

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

In a (mutable) list referenced in the parent frame of the function

```
>>> withdraw = make_withdraw_list(100)
```

Return value: remaining balance

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

Argument: amount to withdraw

Different return value!

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

Second withdrawal of the same amount

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

```
>>> withdraw(15)
35
```

Where's this balance stored?

# Mutable Values & Persistent Local State

# Mutable Values & Persistent Local State

```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)
```
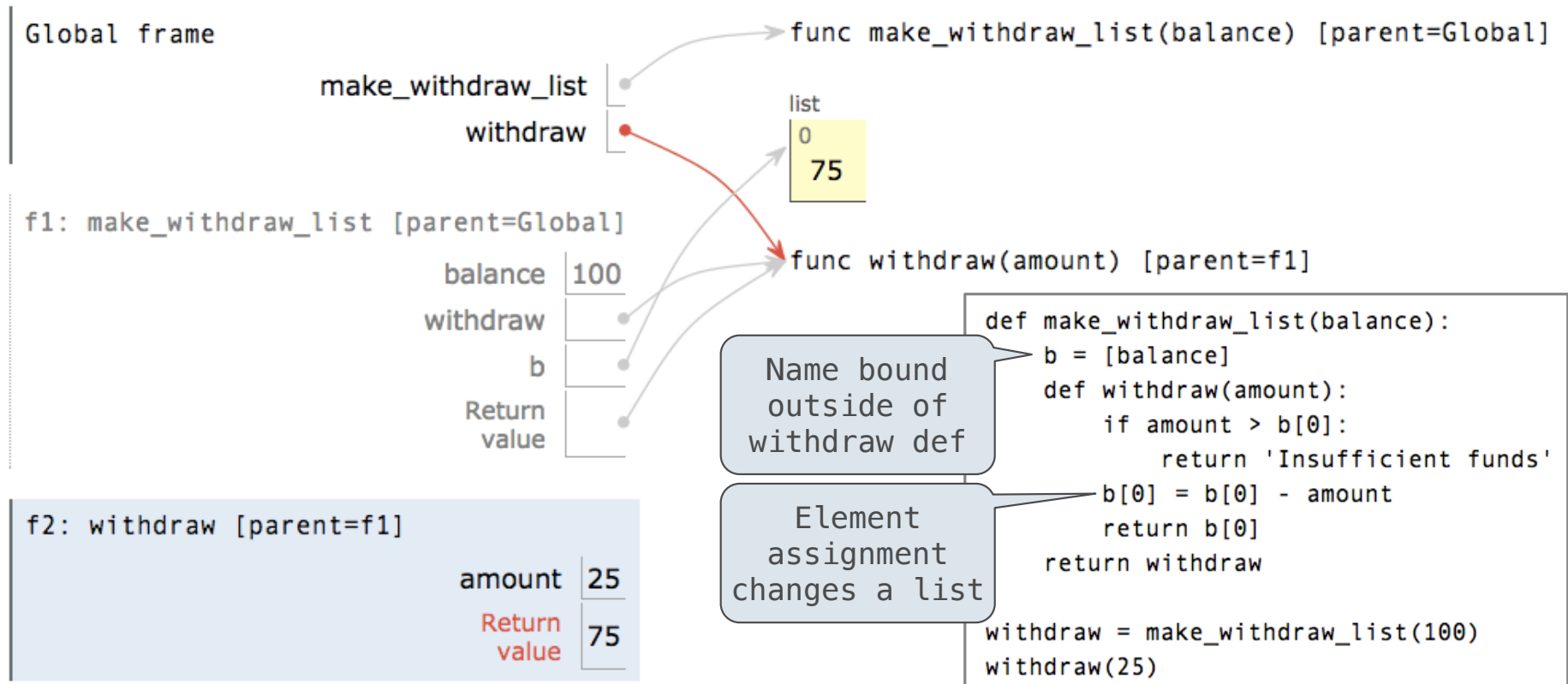
# Mutable Values & Persistent Local State

# Mutable Values & Persistent Local State



```
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

# Mutable Values & Persistent Local State



Global frame

make_withdraw_list → func make_withdraw_list(balance) [parent=Global]

withdraw

list
0
75

f1: make_withdraw_list [parent=Global]

balance | 100
withdraw
b
Return value

func withdraw(amount) [parent=f1]

f2: withdraw [parent=f1]

amount | 25
Return value | 75

Name bound outside of withdraw def

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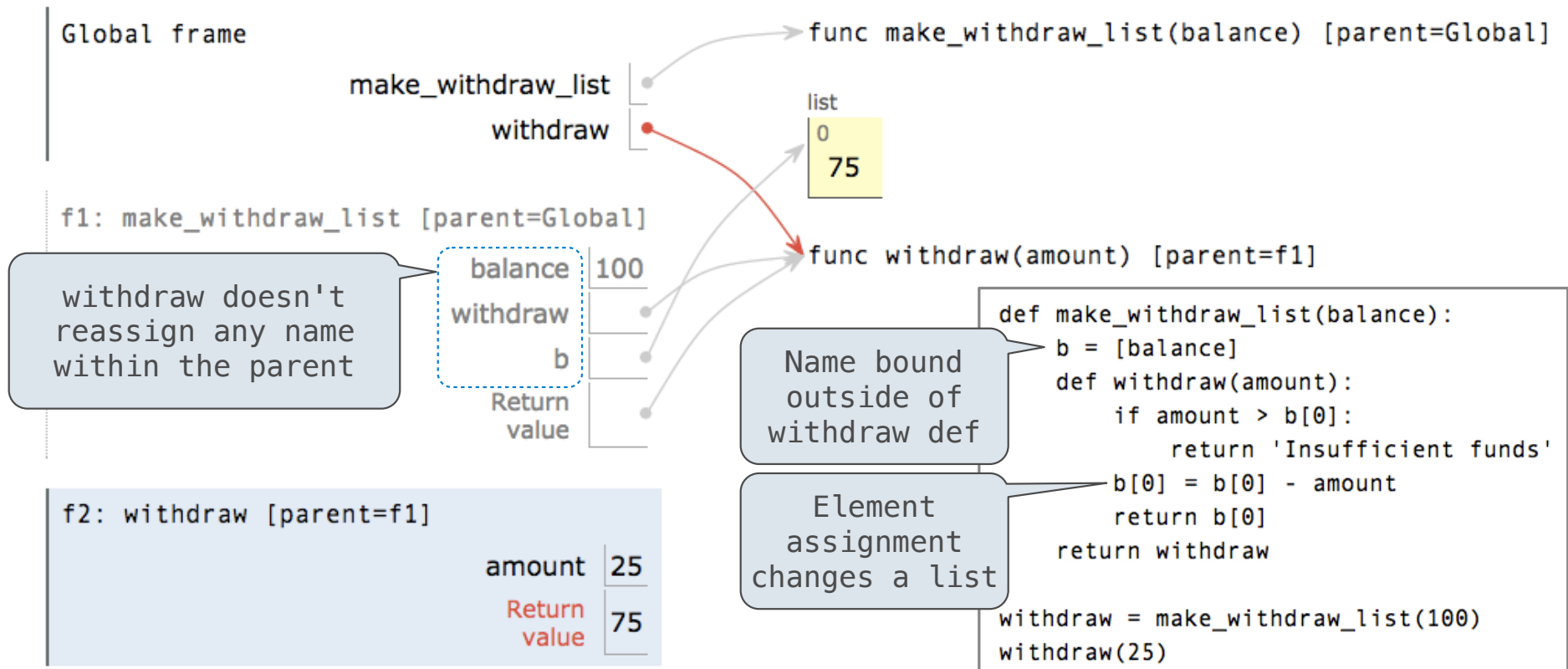
```
def make_withdraw_list(balance):
    b = [balance]
    def withdraw(amount):
        if amount > b[0]:
            return 'Insufficient funds'
        b[0] = b[0] - amount
        return b[0]
    return withdraw

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

# Mutable Values & Persistent Local State

# Mutable Values & Persistent Local State