[220 / 319] Objects+References

Meena Syamkumar
Andy Kuemmel

Test yourself!

A **what is the type of the following?** `{}`

1. **set**
2. **dict**

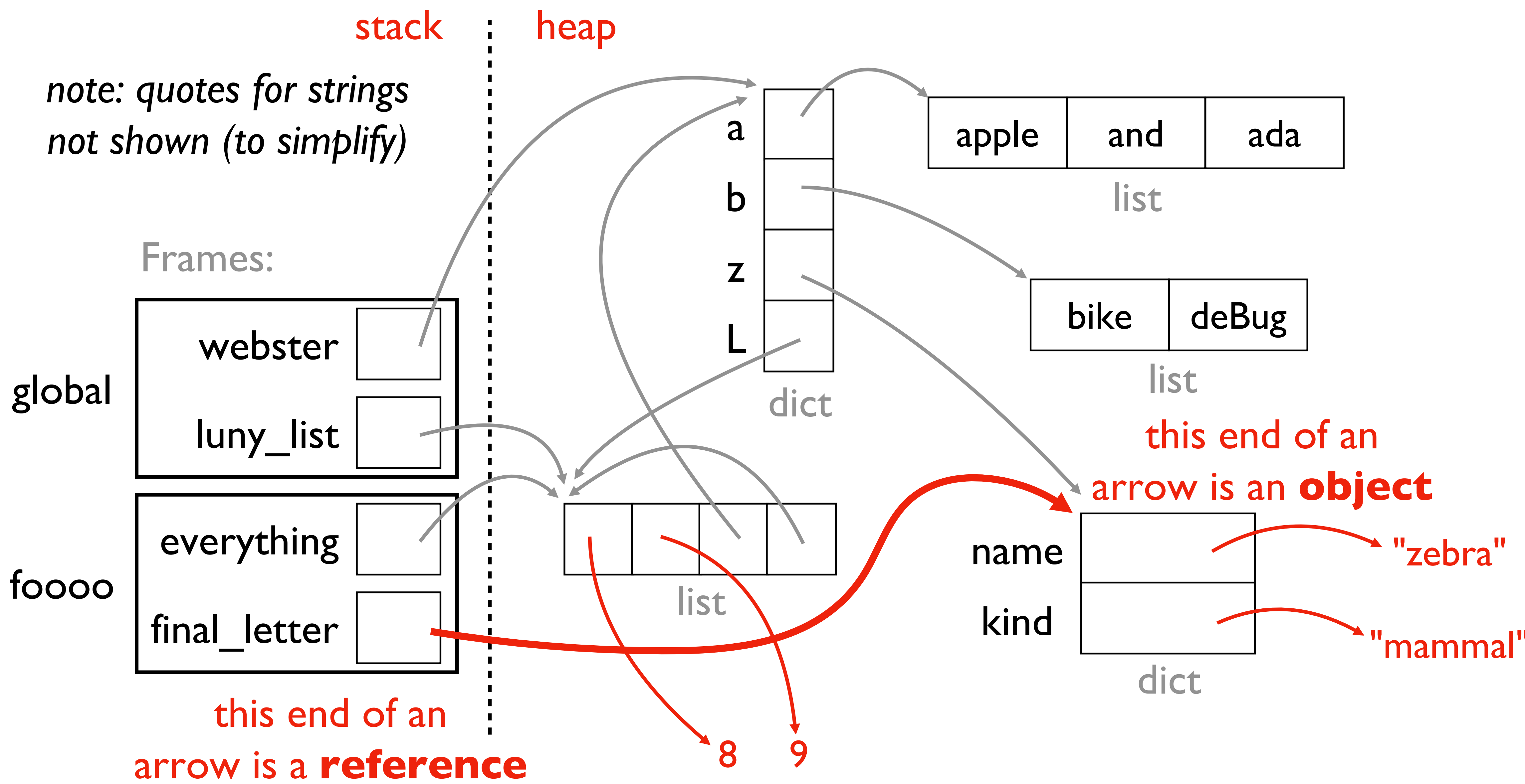
B **if** `S` **is a string and** `L` **is a list, which line definitely fails?**

1. `S[-1] = "."`
2. `L[len(S)] = S`

C **which type is immutable?**

1. `str`
2. `list`
3. `dict`

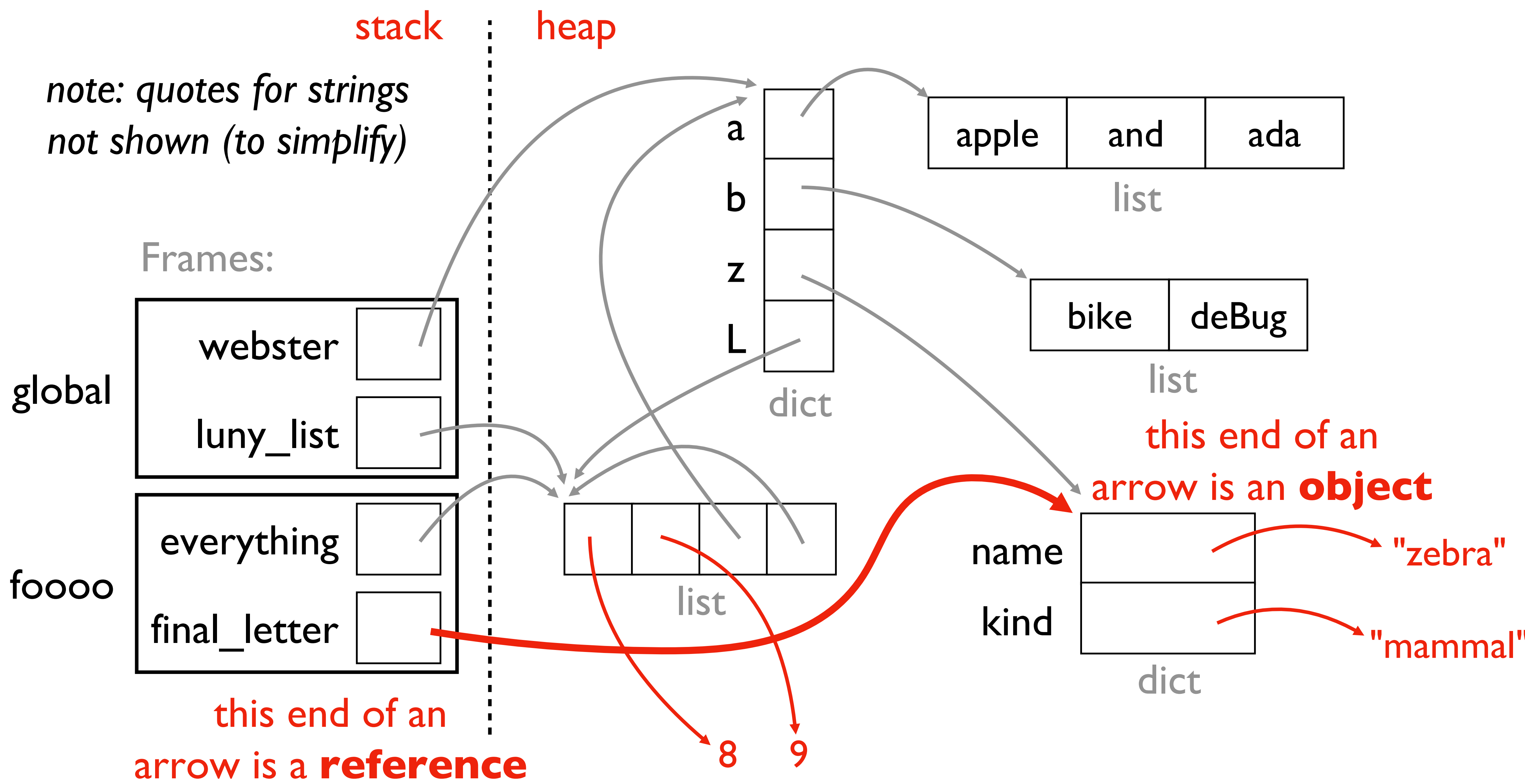
Objects and References



Observations

1. objects have a "life of their own" beyond variables or even function frames
2. here there are dict and list objects (others are possible)
3. references show up two places: as variables and values in data structures
4. technically ints and strs (and all values) are objects too in Python...

Objects and References



Questions

1. why do we need this more complicated model?
2. how can we create new types of objects?
3. how can we compare objects and references?
4. how can we copy objects to create new objects?

Today's Outline

New Types of Objects

- tuple
- namedtuple
- recordclass

References

- motivation
- bugs: accidental argument modification
- “is” vs. “==”

Tuple Sequence

```
nums_list  = [200, 100, 300]
nums_tuple = (200, 100, 300)
```

if you use parentheses (round)
instead of brackets [square]
you get a tuple instead of a list

What is a tuple? A new kind of sequence!

Like a list

- for loop, indexing, slicing, other methods

Unlike a list:

- immutable (like a string)

Tuple Sequence

```
nums_list  = [200, 100, 300]
nums_tuple = (200, 100, 300)
```

```
x = nums_list[2]
x = nums_tuple[2]
```

both put 300 in x

Like a list

- for loop, indexing, slicing, other methods

Unlike a list:

- immutable (like a string)

Tuple Sequence

```
nums_list  = [200, 100, 300]
nums_tuple = (200, 100, 300)
```

```
nums_list[0] = 99
nums_tuple[0] = 99
```

changes list to
[99, 100, 300]

Crashes!

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Like a list

- for loop, indexing, slicing, other methods

Unlike a list:

- immutable (like a string)

Why would we ever want immutability?

1. avoid certain bugs
2. some use cases require it (e.g., dict keys)

Example: location -> building mapping

```
buildings = {
   [0,0]: “Comp Sci”,
   [0,2]: “Psychology”,
   [4,0]: “Noland”,
   [1,8]: “Van Vleck”
}
```

trying to use x,y coordinates as key

FAILS!

```
Traceback (most recent call last):
  File "test2.py", line 1, in <module>
    buildings = {[0,0]: "CS"}
TypeError: unhashable type: 'list'
```

Example: location -> building mapping

```
buildings = {
   (0,0): “Comp Sci”,
   (0,2): “Psychology”,
   (4,0): “Noland”,
   (1,8): “Van Vleck”
}
```

trying to use x,y coordinates as key

Succeeds!

(with tuples)

A note on parenthetical characters

type of parenthesis	uses		
parentheses: (and)	specifying order:	`(1+2)*3`	**(1+2)**
	function invocation:	`f()`	
	tuple:	`(1,2,3)`	**(1+2,)** tuple of size 1
brackets: [and]	list creation:	`s = [1,2,3]`	
	sequence indexing:	`s[-1]`	
	sequence slicing:	`s[1:-2]`	
	dict lookup:	`d["one"]`	
braces: { and }	dict creation:	`d = {"one":1, "two":2}`	
	set creation:	`{1,2,3}`	

Today's Outline

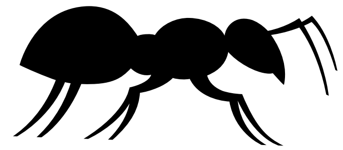
New Types of Objects

- tuple
- namedtuple
- recordclass

References

- motivation
- bugs: accidental argument modification
- “is” vs. “==”

See any bugs?



1

```
people=[
  {"Fname": "Alice", "lname": "Anderson", "age": 30},
  {"fname": "Bob", "lname": "Baker", "age": 31},
]
p = people[0]
print("Hello " + p["fname"] + " " + p["lname"])
```

dict

2

```
people=[
  ("Alice", "Anderson", 30),
  ("Bob", "Baker", 31),
]
p = people[1]
print("Hello " + p[1] + " " + p[2])
```

tuple

Vote: Which is Better Code?

1

```
people=[
  {"fname": "Alice", "lname": "Anderson", "age": 30},
  {"fname": "Bob", "lname": "Baker", "age": 31},
]
p = people[0]
print("Hello " + p["fname"] + " " + p["lname"])
```

dict

2

```
people=[
  ("Alice", "Anderson", 30),
  ("Bob", "Baker", 31),
]
p = people[1]
print("Hello " + p[0] + " " + p[1])
```

tuple

1

```python
people=[
  {"fname": "Alice", "lname": "Anderson", "age": 30},
  {"fname": "Bob", "lname": "Baker", "age": 31},
]
p = people[0]
print("Hello " + p["fname"] + " " + p["lname"])
```

dict

2

```python
people=[
  ("Alice", "Anderson", 30),
  ("Bob", "Baker", 31),
]
p = people[1]
print("Hello " + p[0] + " " + p[1])
```

tuple

3

```python
from collections import namedtuple
Person = namedtuple("Person", ["fname", "lname", "age"])
people=[
    Person("Alice", "Anderson", 30),
    Person("Bob", "Baker", 31),
]
p = people[0]
print("Hello " + p.fname + " " + p.lname)
```

namedtuple

```python
from collections import namedtuple
```

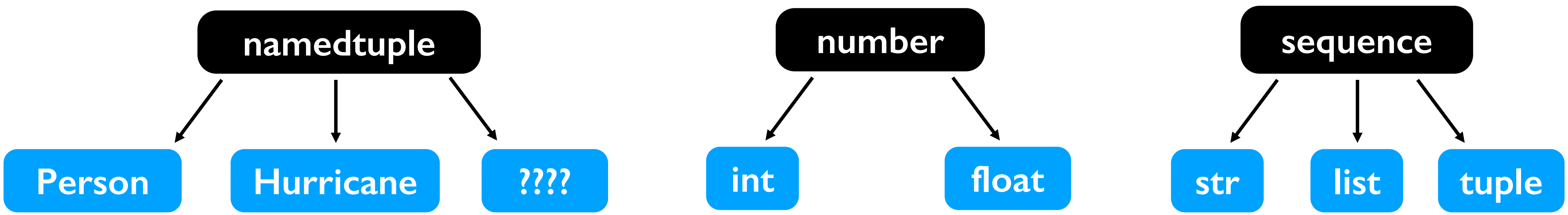
need to import this data struct

name of that type

creates a new type!

name of that type

```python
Person = namedtuple("Person", ["fname", "lname", "age"])
```




```python
p = Person("Alice", "Anderson", 30)
```

creates a object of type Person (sub type of namedtuple)
(like **`str(3)`** creates a new string or **`list()`** creates a new list)

```python
print("Hello " + p.fname + " " + p.lname)
```

```python
from collections import namedtuple

Person = namedtuple("Person", ["fname", "lname", "age"])

p = Person("Alice", "Anderson", 30)
```

can use either **positional** or keyword arguments to create a Person

```python
print("Hello " + p.fname + " " + p.lname)
```

```python
from collections import namedtuple

Person = namedtuple("Person", ["fname", "lname", "age"])

p = Person(age=30, fname="Alice", lname="Anderson")
```

can use either positional or **keyword** arguments to create a Person

```python
print("Hello " + p.fname + " " + p.lname)
```

```python
from collections import namedtuple

Person = namedtuple("Person", ["fname", "lname", "age"])

p = Person(age=30, Fname="Alice", lname="Anderson")

print("Hello " + p.fname + " " + p.lname)
```

crashes
immediately
(good!)

```python
from collections import namedtuple

Person = namedtuple("Person", ["fname", "lname", "age"])

p = Person(age=30, fname="Alice", lname="Anderson")

print("Hello " + p.fname + " " + p.lname)
```

Today's Outline

New Types of Objects

- tuple
- namedtuple
- recordclass ← mutable equivalent of a namedtuple

References

- motivation
- bugs: accidental argument modification
- “is” vs. “==”

which type supports ~~birthdays~~ mutability?

```
from collections import namedtuple

Person = namedtuple("Person", ["fname", "lname", "age"])
p = Person(age=30, fname="Alice", lname="Anderson")

p.age += 1 # it's a birthday!

print("Hello " + p.fname + " " + p.lname)
```

namedtuple

```
from recordclass import recordclass # not in collections!

Person = recordclass("Person", ["fname", "lname", "age"])
p = Person(age=30, fname="Alice", lname="Anderson")

p.age += 1 # it's a birthday!

print("Hello " + p.fname + " " + p.lname)
```

recordclass

which type supports ~~birthdays~~ mutability?

```python
from collections import namedtuple

Person = namedtuple("Person", ["fname", "lname", "age"])
p = Person(age=30, fname="Alice", lname="Anderson")
```

~~`p.age += 1 # it's a birthday!`~~

```python
print("Hello " + p.fname + " " + p.lname)
```

namedtuple

```python
from recordclass import recordclass # not in collections!

Person = recordclass("Person", ["fname", "lname", "age"])
p = Person(age=30, fname="Alice", lname="Anderson")

p.age += 1 # it's a birthday!

print("Hello " + p.fname + " " + p.lname)
```

recordclass

need to install recordclass:

```
pip install recordclass
```

```python
from recordclass import recordclass # not in collections!

Person = recordclass("Person", ["fname", "lname", "age"])
p = Person(age=30, fname="Alice", lname="Anderson")

p.age += 1 # it's a birthday!

print("Hello " + p.fname + " " + p.lname)
```

recordclass

Today's Outline

New Types of Objects

- tuple
- namedtuple
- recordclass

let's evolve our mental model of state!

References

- motivation
- bugs: accidental argument modification
- “is” vs. “==”

Mental Model for State (v1)

Code:

```
x = “hello”
y = x
y += “ world”
```

State:

x

y

note: *we're not drawing frame boxes for simplicity since everything is in the global frame*

Mental Model for State (v1)

Code:

```
x = “hello”
y = x
y += “ world”
```

State:

x	hello
y	

Mental Model for State (v1)

Code:

```
x = “hello”
y = x
y += “ world”
```

State:

x	hello
y	hello

Mental Model for State (v1)

Code:

```
x = "hello"
y = x
y += " world"
```

Common mental model
- equivalent for immutable types
- PythonTutor uses for strings, etc

Issues
- incorrect for mutable types
- ignores performance

State:

x	**hello**
y	**hello world**

Mental Model for State (v2)

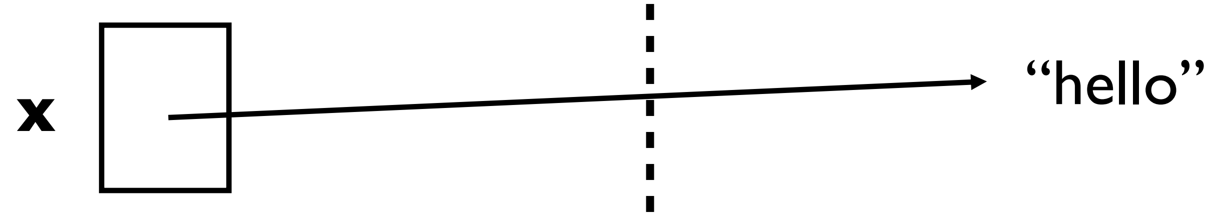
Code:

```
x = “hello”
y = x
y += “ world”
```

State:

references

objects



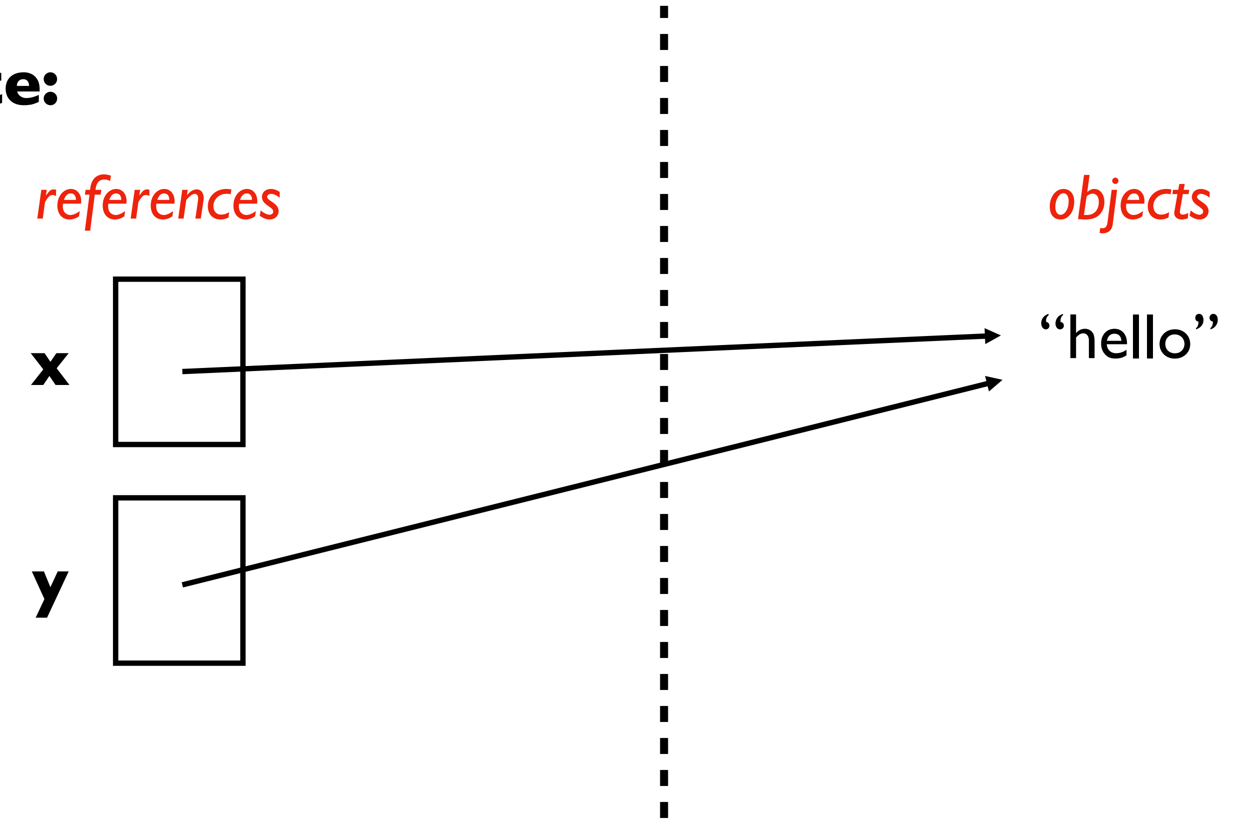
any box with an arrow is a reference
(variables are one kind of reference)

Mental Model for State (v2)

Code:

```
x = "hello"
y = x
y += " world"
```

State:

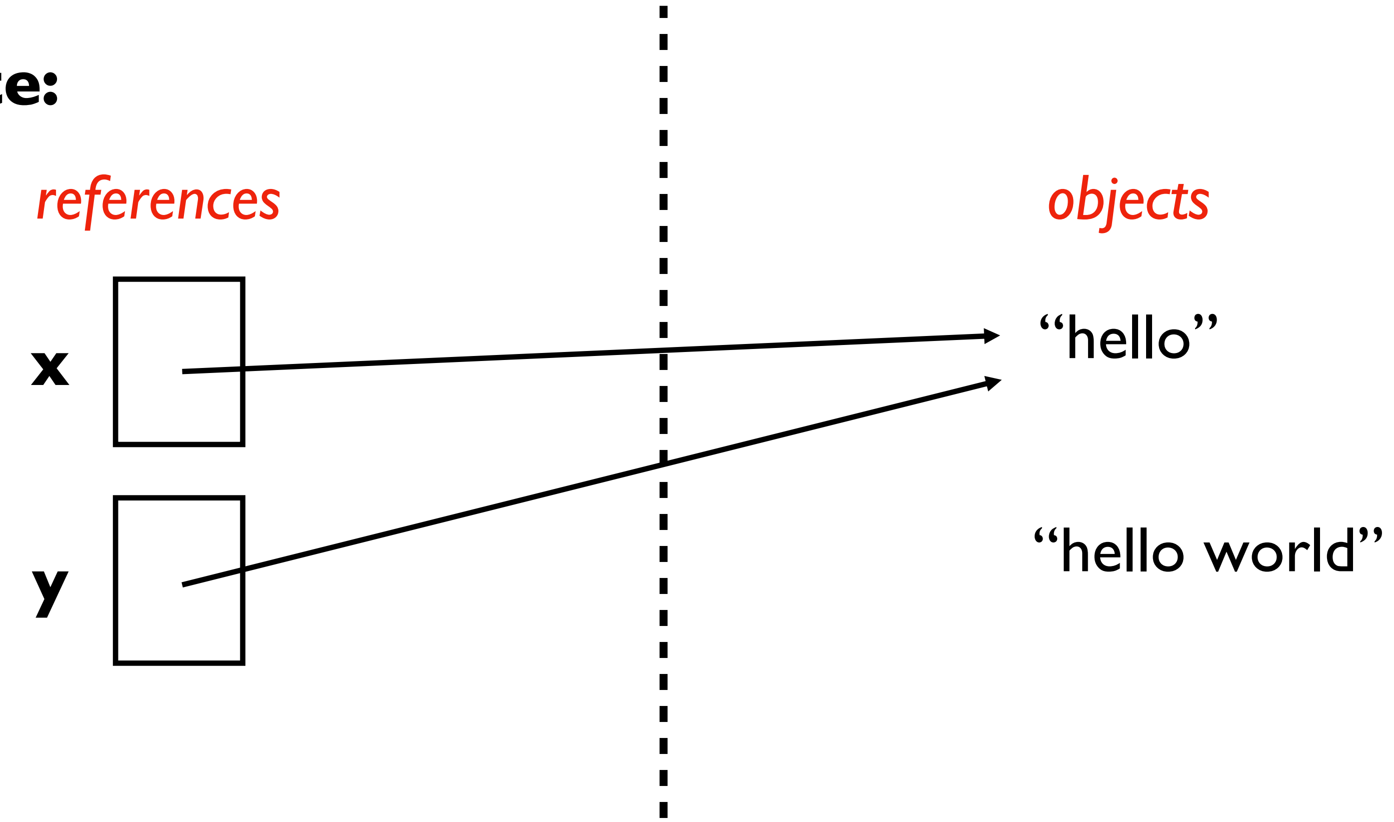


Mental Model for State (v2)

Code:

```
x = “hello”
y = x
y += “ world”
```

State:

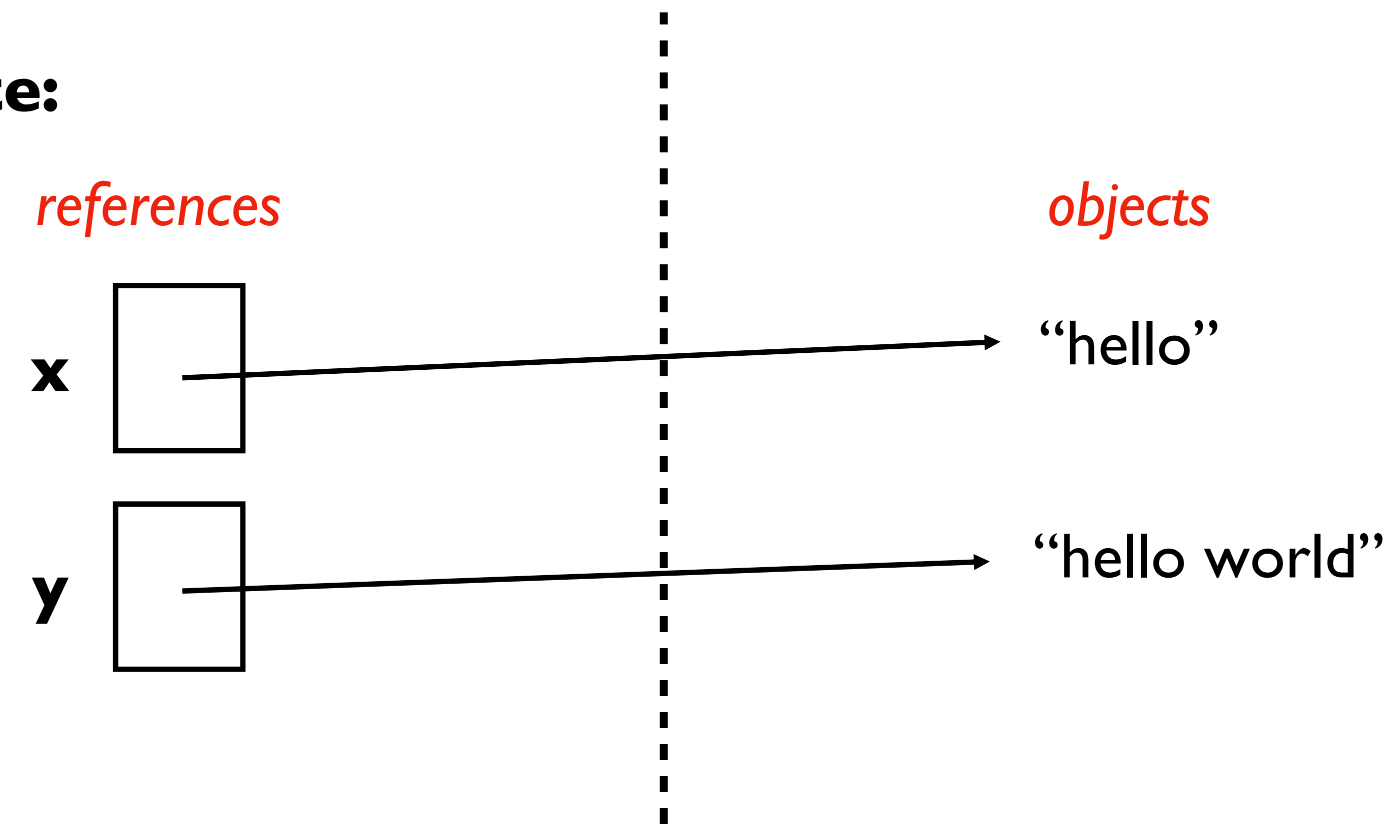


Mental Model for State (v2)

Code:

```
x = "hello"
y = x
y += " world"
```

State:

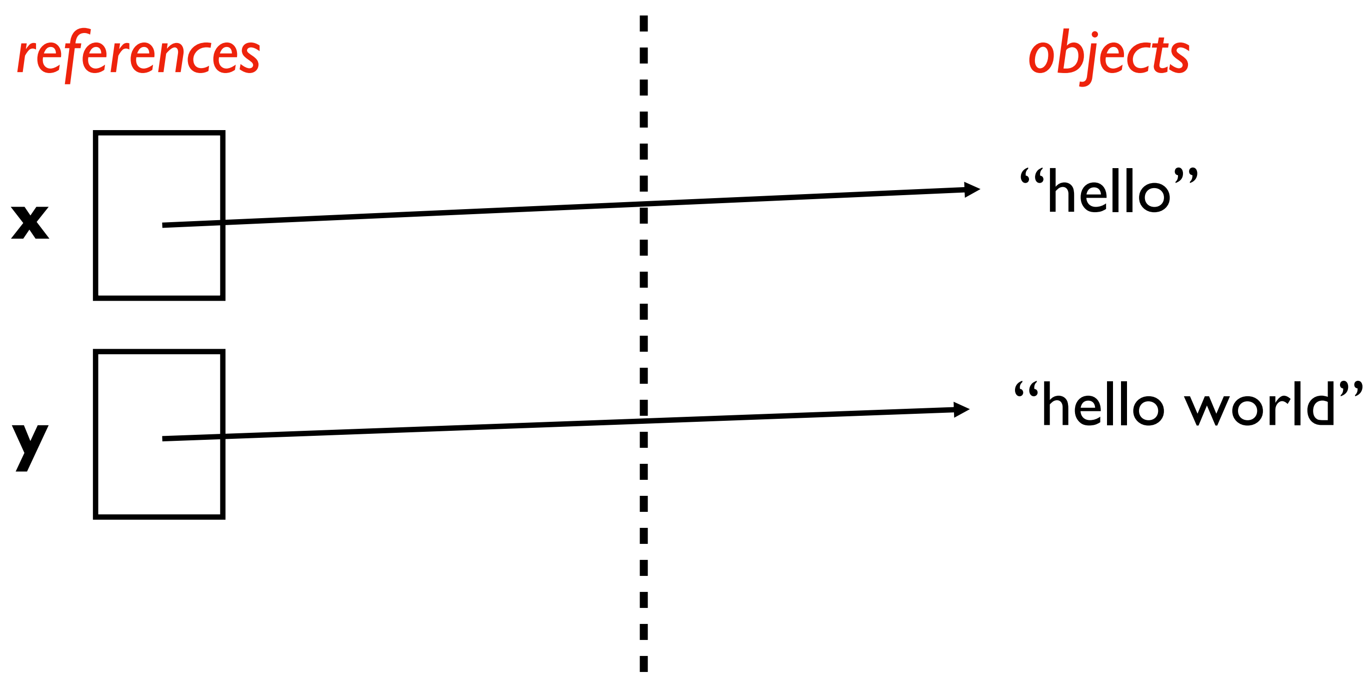


Mental Model for State (v2)

Code:

```
x = “hello”
y = x
y += “ world”      # y = y + “ world”
```

State:



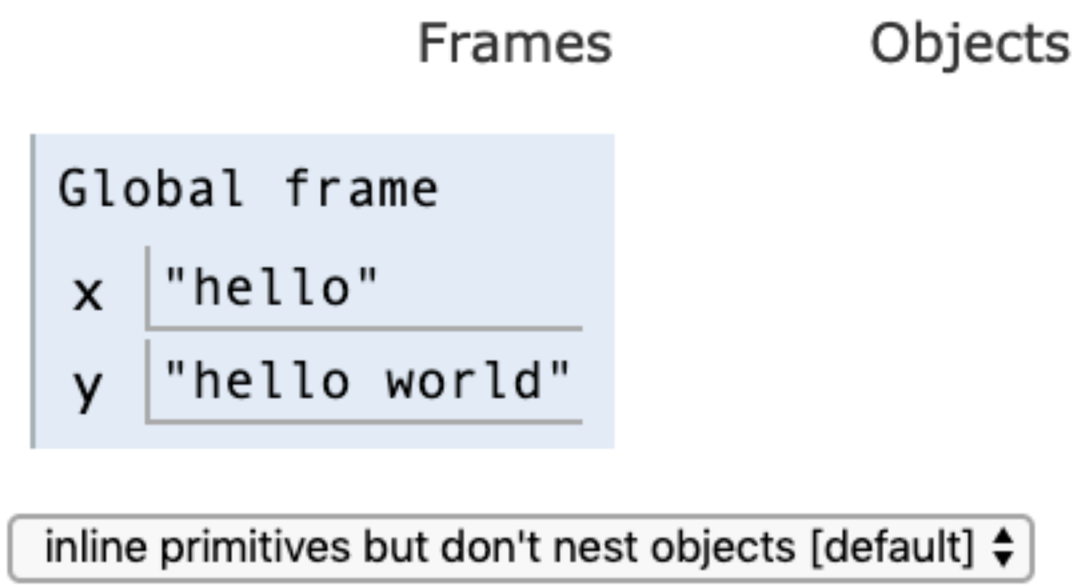
Revisiting Assignment and Passing Rules for v2

```
# RULE 1 (assignment)
x = ????
y = x  # y should reference whatever x references
```

```
# RULE 2 (argument passing)
def f(y):
    pass

x = ????
f(x)  # y should reference whatever x references
```

How PythonTutor renders immutable types is configurable...

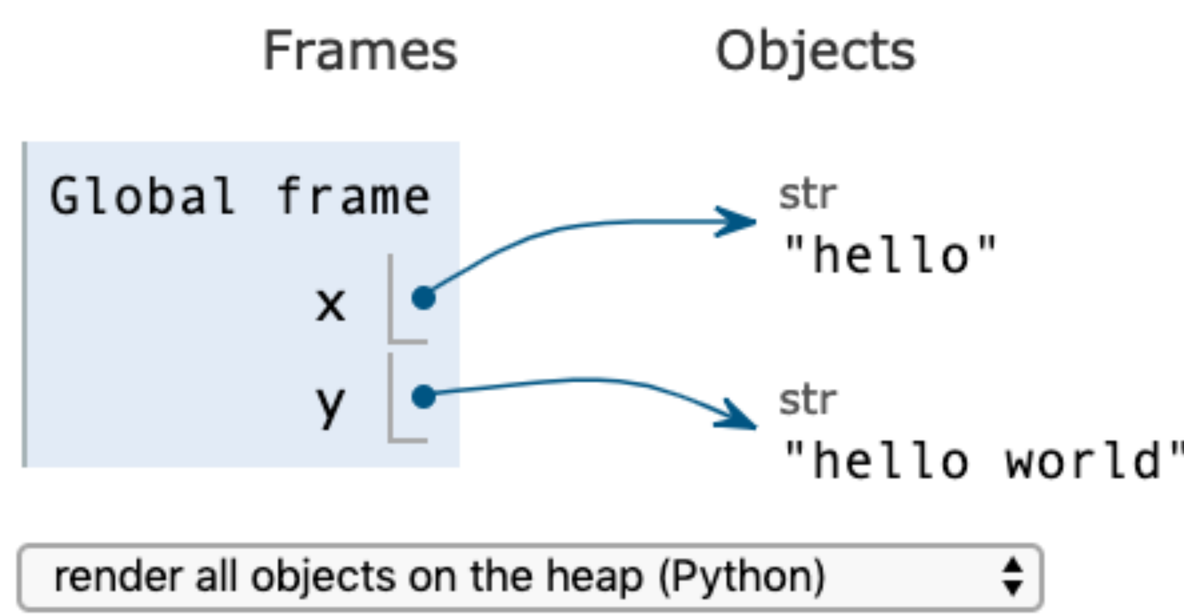
v1



Code:

```
x = "hello"
y = x
y += " world"
```

v2



Today's Outline

New Types of Objects

- tuple
- namedtuple
- recordclass

References

- motivation
- bugs: accidental argument modification
- “is” vs. “==”

Why does Python have the complexity of separate **references** and **objects**?

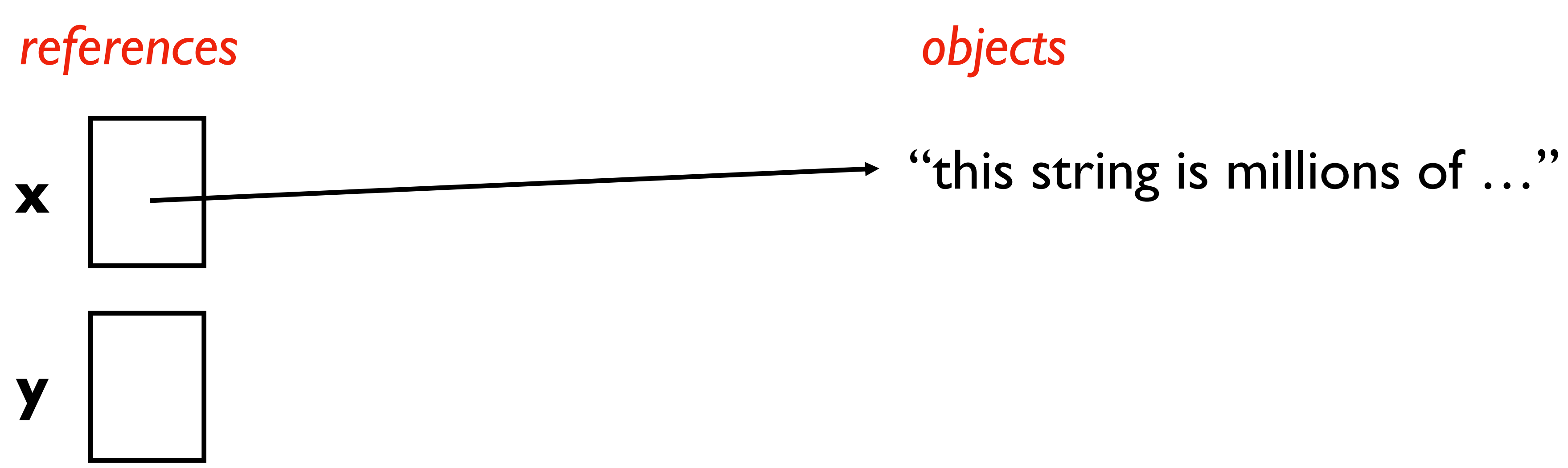
Why not follow the original organization we saw for everything (*i.e.*, boxes of data with labels)?

Reason 1: Performance

Code:

```
x = "this string is millions of characters…"
y = x # this is fast!
```

State:

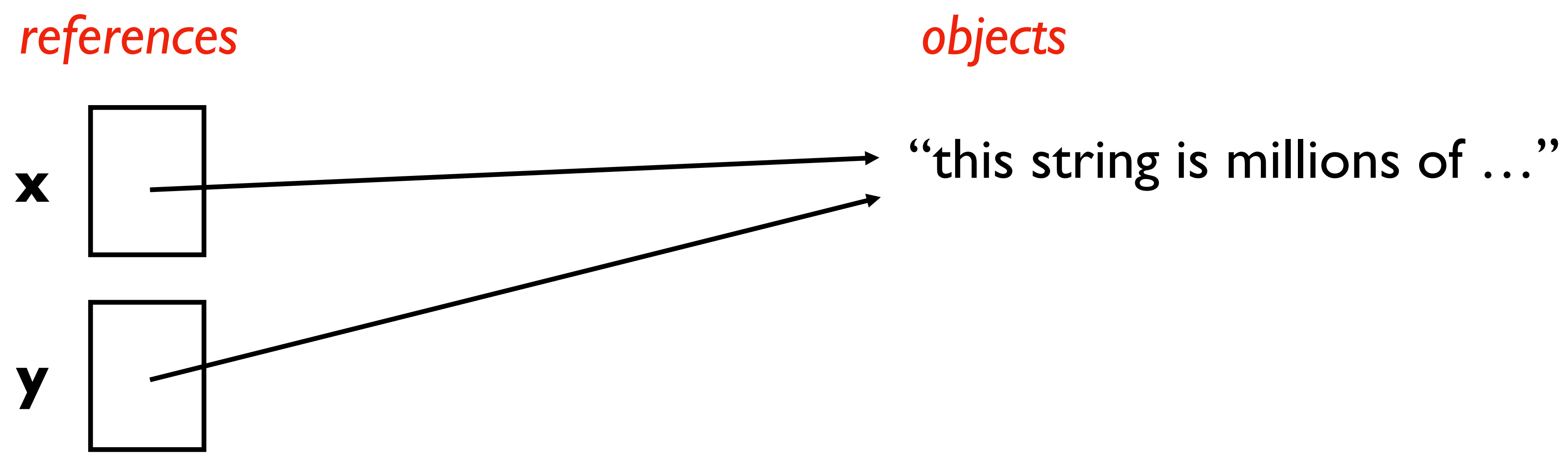


Reason 1: Performance

Code:

```
x = “this string is millions of characters…”
y = x # this is fast!
```

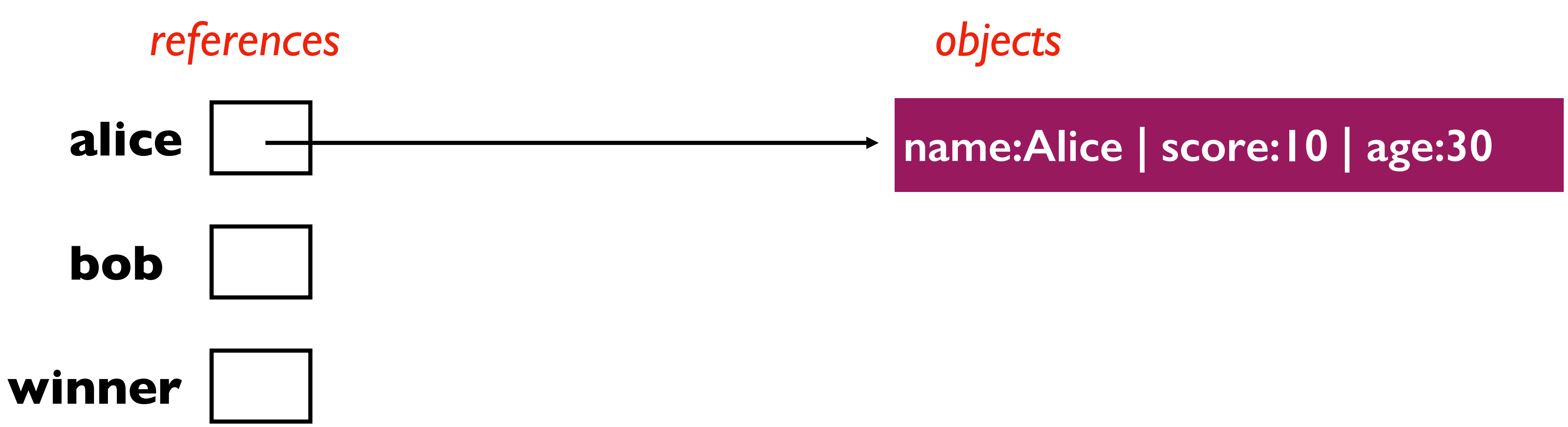
State:



Reason 2: Centralized Updates

```
from recordclass import recordclass

Person = recordclass("Person", ["name", "score", "age"])

alice = Person(name="Alice", score=10, age=30)
bob = Person(name="Bob", score=8, age=25)
winner = alice

alice.age += 1
print("Winner age:", winner.age)
```

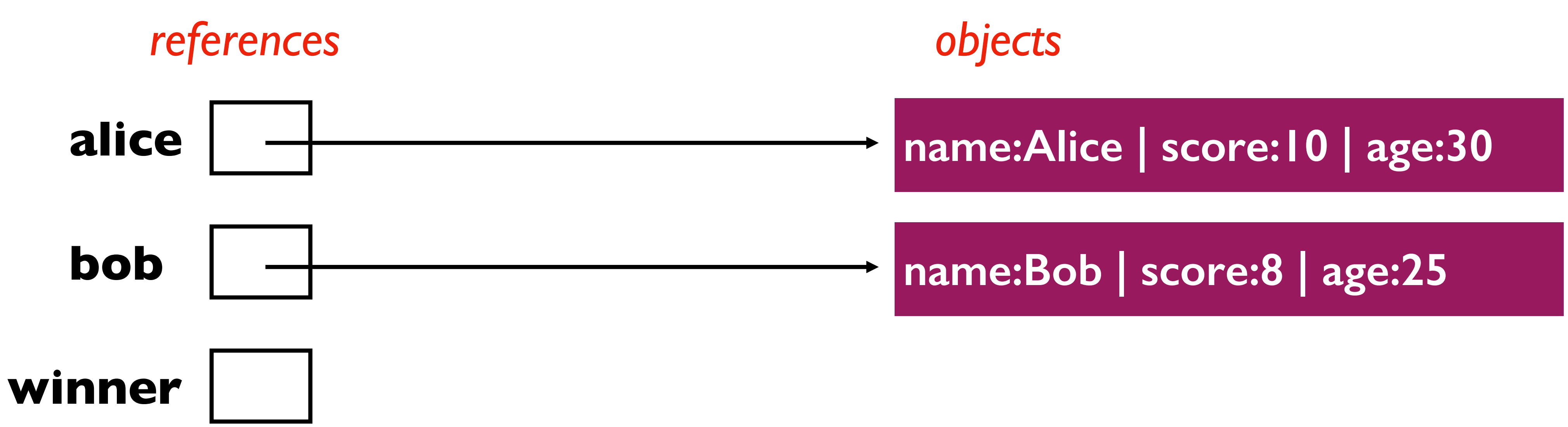
State:



Reason 2: Centralized Updates

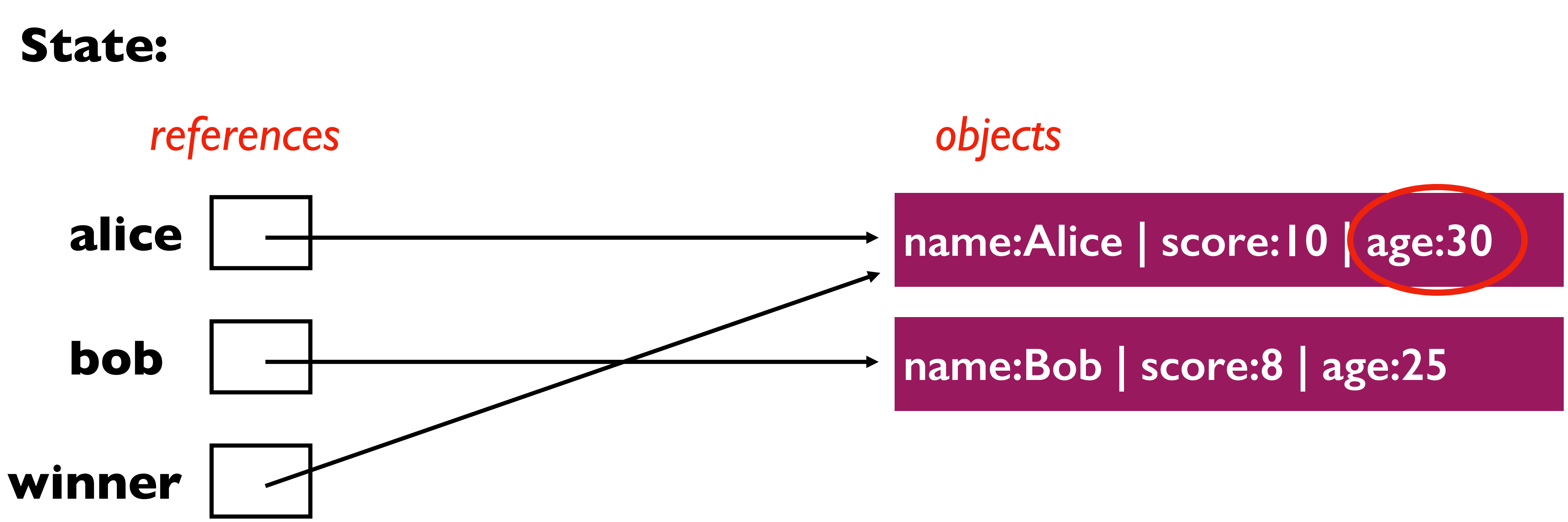
```
from recordclass import recordclass

Person = recordclass("Person", ["name", "score", "age"])

alice = Person(name="Alice", score=10, age=30)
bob = Person(name="Bob", score=8, age=25)
winner = alice

alice.age += 1
print("Winner age:", winner.age)
```

State:



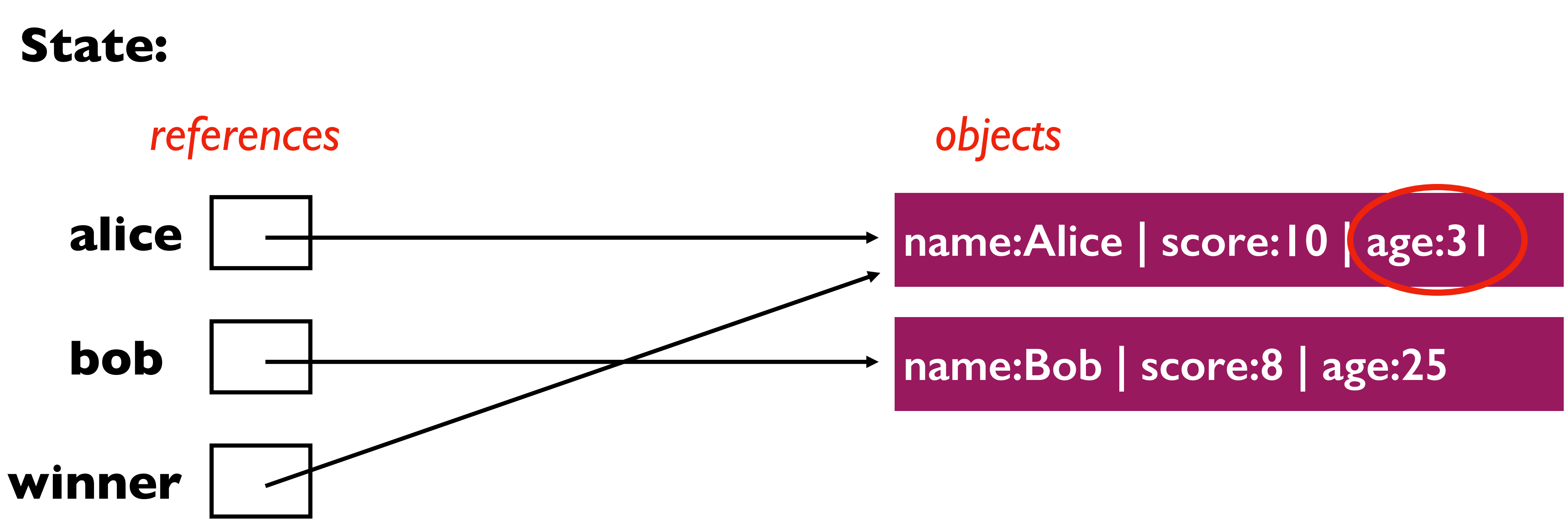
Reason 2: Centralized Updates

```
from recordclass import recordclass

Person = recordclass("Person", ["name", "score", "age"])

alice = Person(name="Alice", score=10, age=30)
bob = Person(name="Bob", score=8, age=25)
winner = alice

alice.age += 1
print("Winner age:", winner.age)
```



Reason 2: Centralized Updates

```
from recordclass import recordclass

Person = recordclass("Person", ["name", "score", "age"])

alice = Person(name="Alice", score=10, age=30)
bob = Person(name="Bob", score=8, age=25)
winner = alice

alice.age += 1
print("Winner age:", winner.age)
```

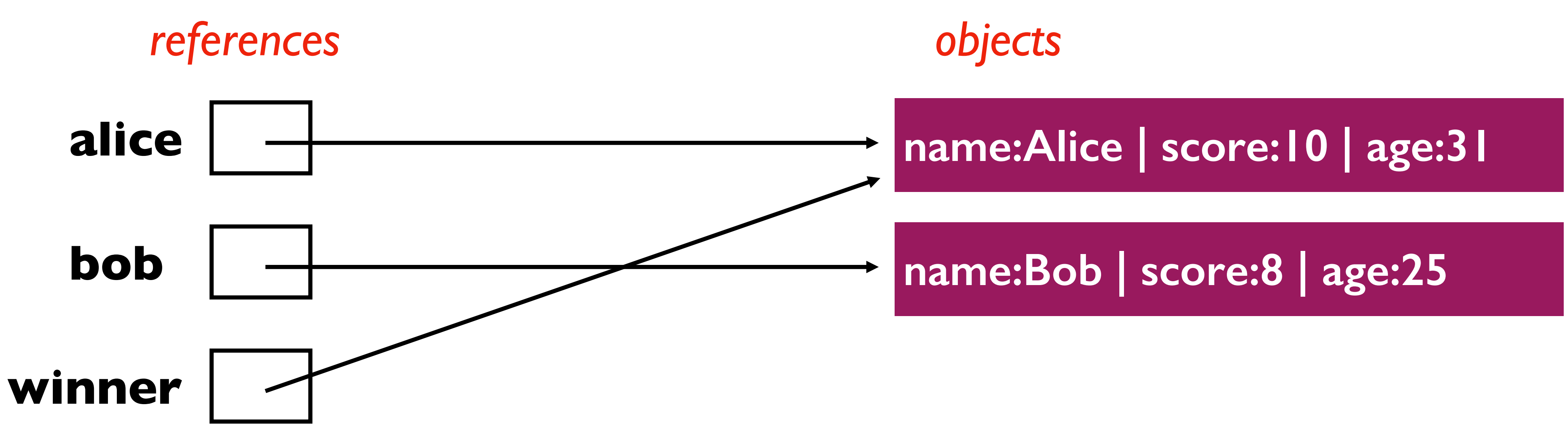


Reason 2: Centralized Updates

```
from recordclass import recordclass

Person = recordclass("Person", ["name", "score", "age"])

alice = Person(name="Alice", score=10, age=30)
bob = Person(name="Bob", score=8, age=25)
winner = alice

alice.age += 1
print("Winner age:", winner.age)
```

prints 31, even though we didn't directly modify winner

State:



Today's Outline

New Types of Objects

- tuple
- namedtuple
- recordclass

References

- motivation
- bugs: accidental argument modification
- “is” vs. “==”

References and Arguments/Parameters

Python Tutor **always** illustrates references with an arrow for mutable types

Thinking carefully about a few examples will prevent many debugging headaches…

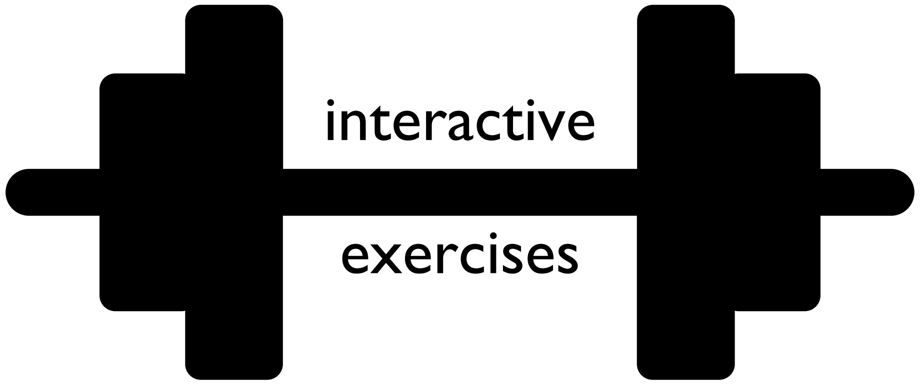
Example 1: reassign parameter

```
def f(x):
  x *= 3
  print("f:", x)

num = 10
f(num)
print("after:", num)
```

interactive
exercises

Example 2: modify list via param

```
def f(items):
  items.append("!!!")
  print("f:", items)

words = ['hello', 'world']
f(words)
print("after:", words)
```



Example 3: reassign new list to param

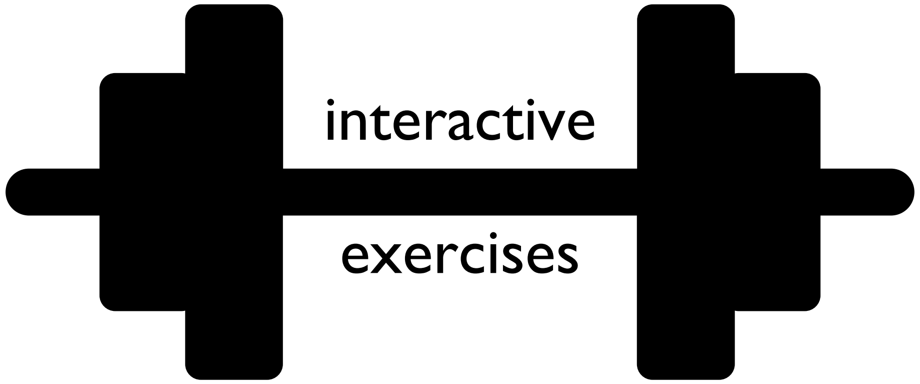
```
def f(items):
    items = items + ["!!!"]
    print("f:", items)

words = ['hello', 'world']
f(words)
print("after:", words)
```

interactive

exercises

Example 4: in-place sort

```python
def first(items):
    return items[0]

def smallest(items):
    items.sort()
    return items[0]

numbers= [4,5,3,2,1]
print("first:", first(numbers))
print("smallest:", smallest(numbers))
print("first:", first(numbers))
```



Example 5: sorted sort

```
def first(items):
    return items[0]

def smallest(items):
    items = sorted(items)
    return items[0]

numbers= [4,5,3,2,1]
print("first:", first(numbers))
print("smallest:", smallest(numbers))
print("first:", first(numbers))
```

interactive
exercises

Today's Outline

New Types of Objects

- tuple
- namedtuple
- recordclass

References

- motivation
- bugs: accidental argument modification
- “is” vs. “==”

are two objects equivalent?

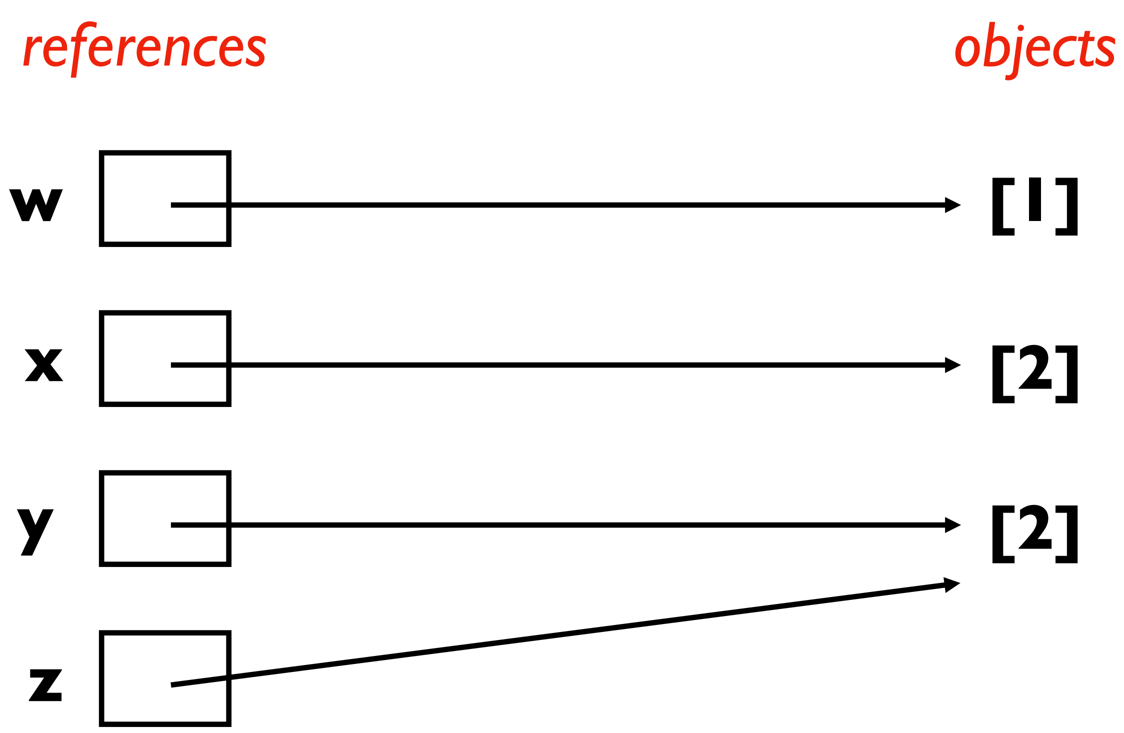
are two references equivalent?

== and is

```
w = [1]
x = [2]
y = [2]
z = y
```

observation: x and y are **equal** to each other, but y and z are **MORE equal** to each other

State:



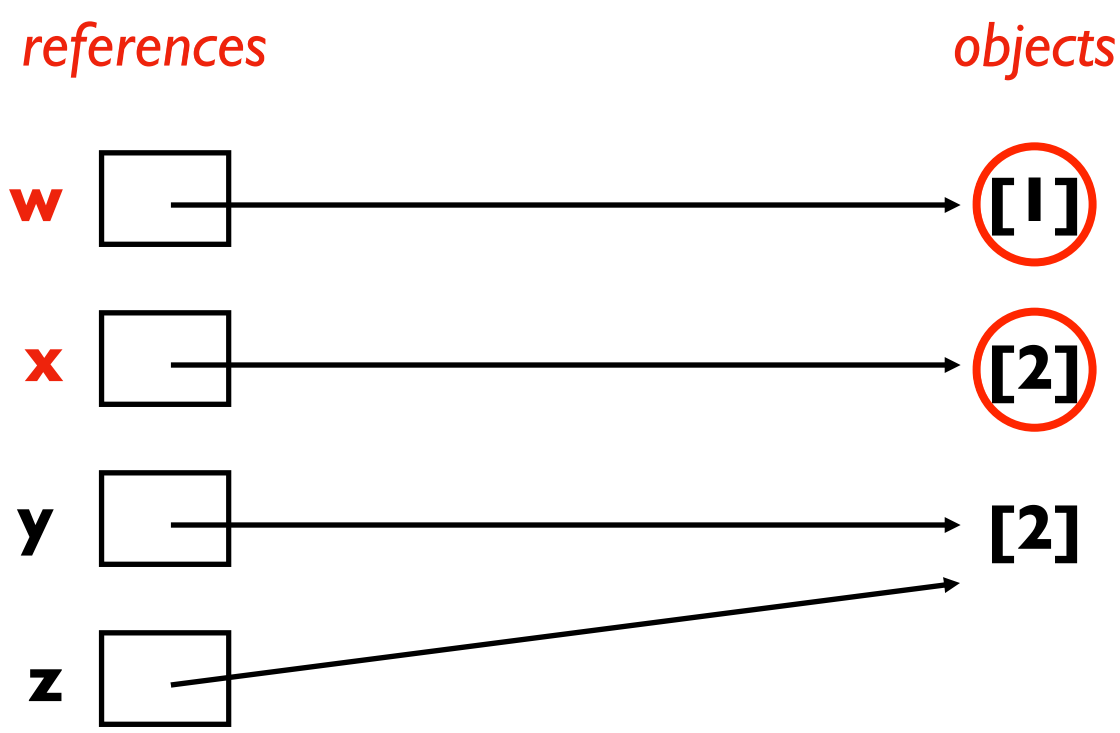
== and is

```
w = [1]
x = [2]
y = [2]
z = y
```

w == x

False

State:

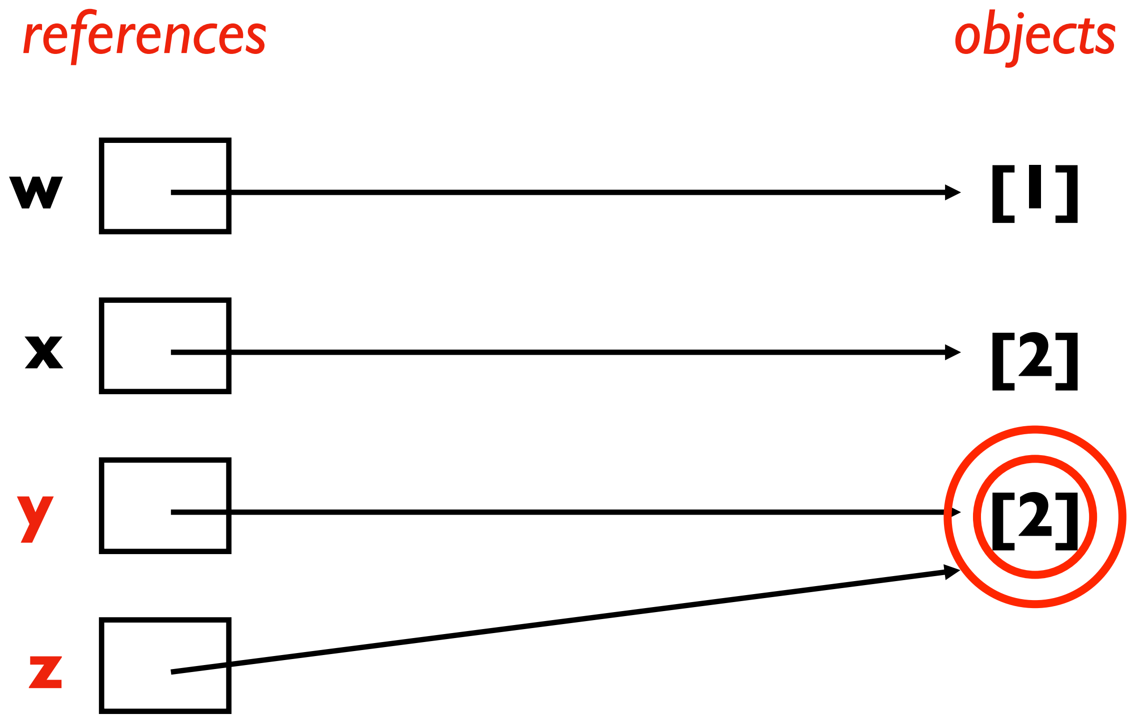


== and is

```
w = [1]
x = [2]
y = [2]
z = y
```

y == z

State:



== and is

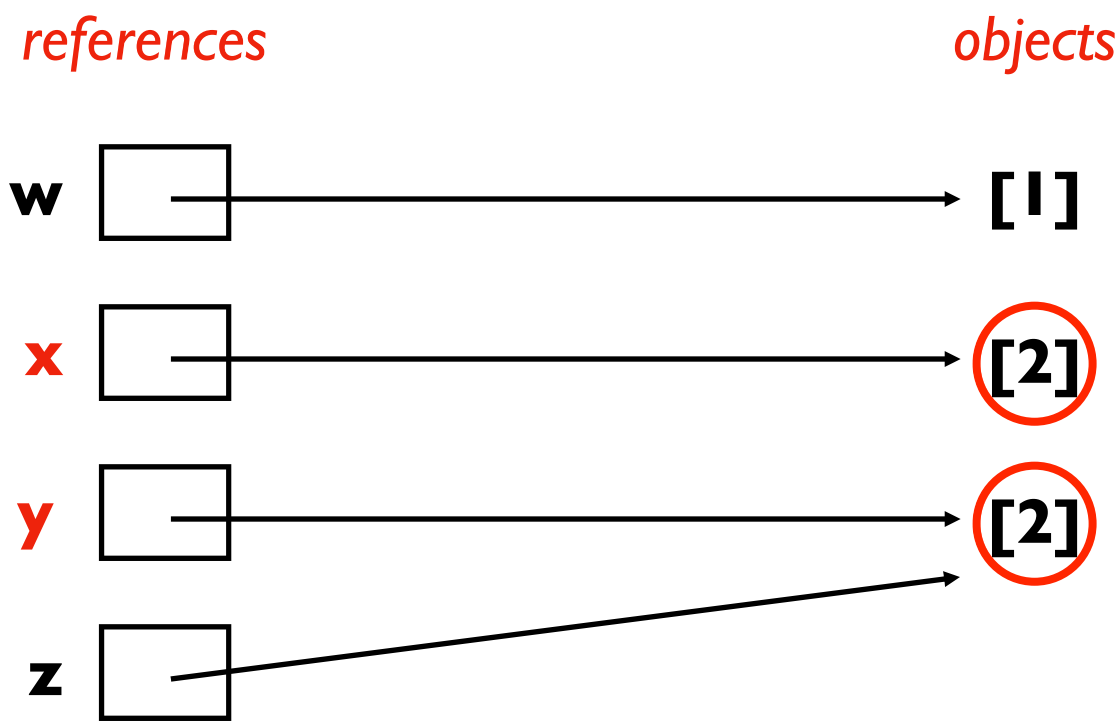
```
w = [1]
x = [2]
y = [2]
z = y
```

x == y

True

because x and y refer to
two equivalent objects

State:



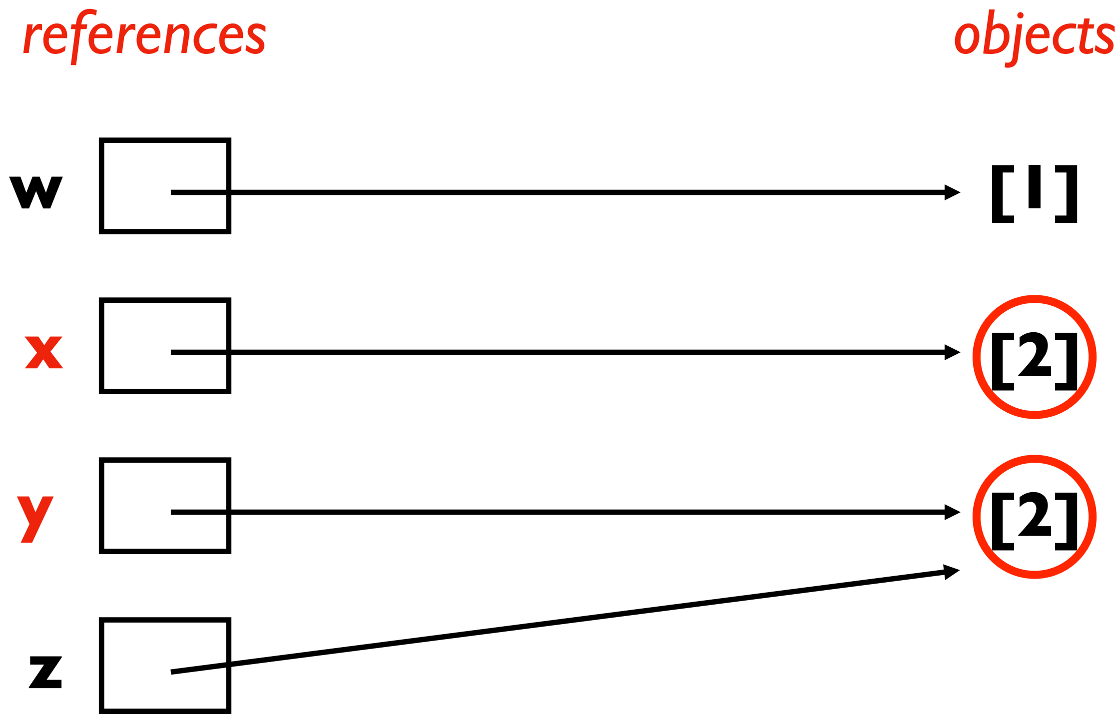
== and is

```
w = [1]
x = [2]
y = [2]
z = y
```

x is y

new operator to check if two references refer to the same object

State:



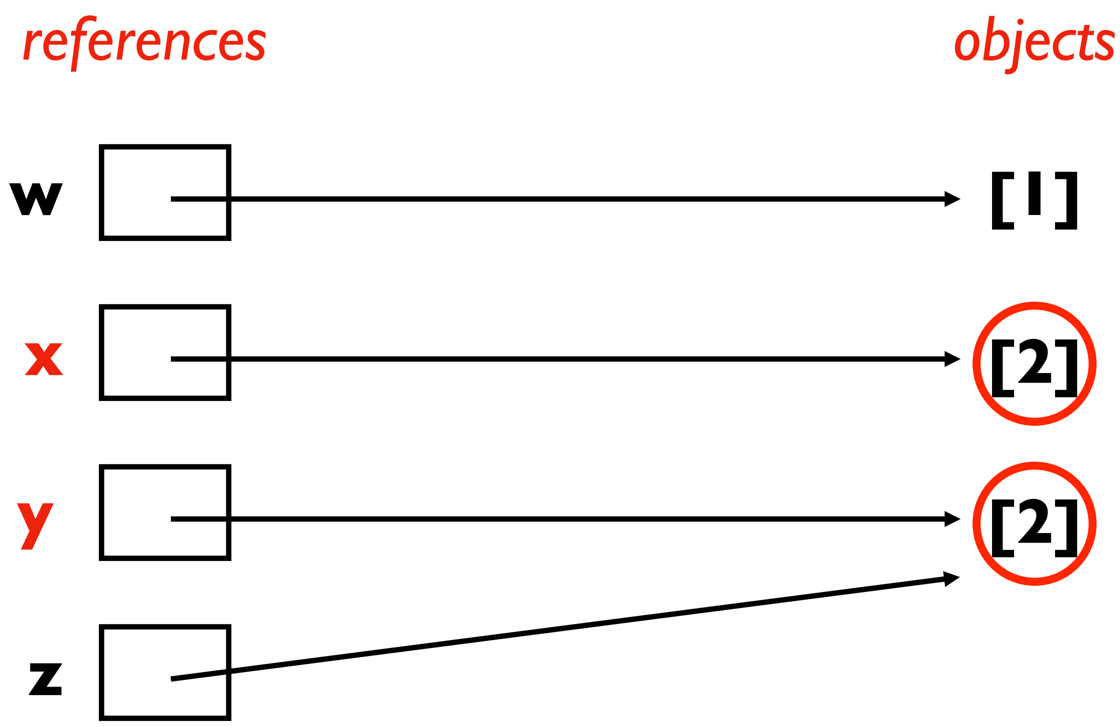
== and is

```
w = [1]
x = [2]
y = [2]
z = y
```

x is y

False

State:



== and is

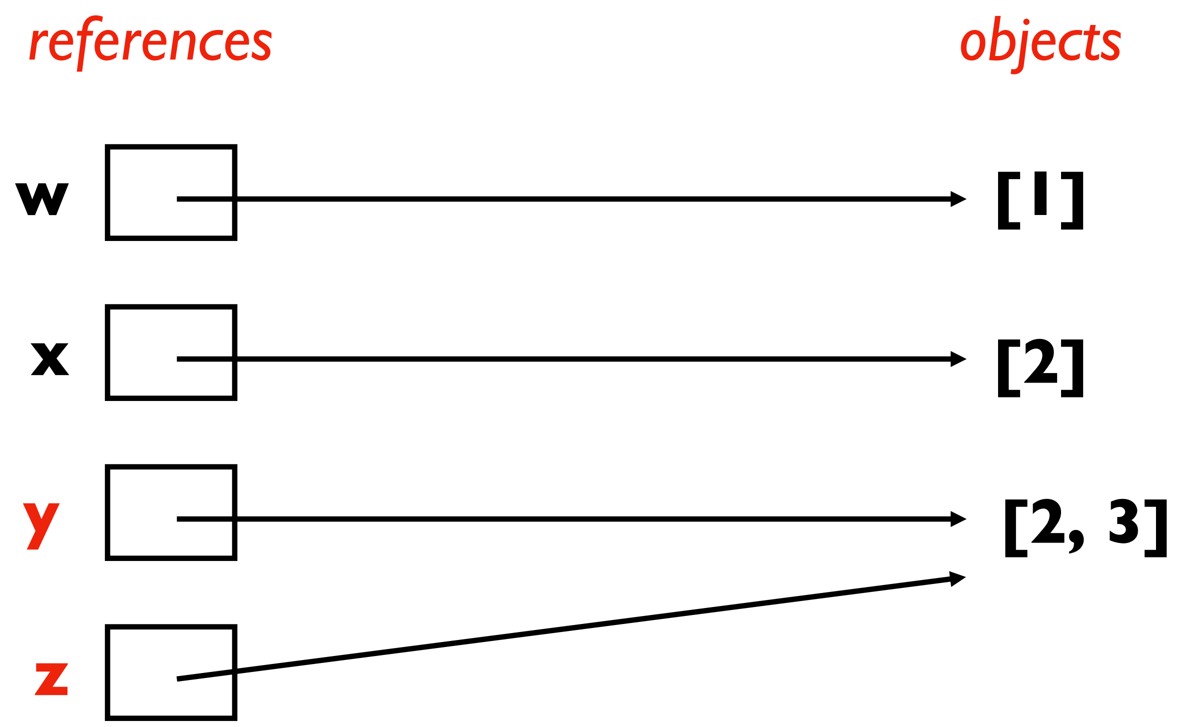
```
w = [1]
x = [2]
y = [2]
z = y
y.append(3)
print(z) # [2,3]
```

y is z

True

This tells you that changes to
y will show up if we check z

State:



Be careful with `is`!

Python sometimes “deduplicates” equal immutable values

- This is an unpredictable optimization (called interning)
- 90% of the time, you want == instead of is
  (then you don’t need to care about this optimization)
- Play with changing replacing 10 with other numbers to see potential pitfalls:

```
a = 'ha' * 10
b = 'ha' * 10
print(a == b)
print(a is b)
```

Conclusion

New Types of Objects

- tuple: immutable equivalent as list
- namedtuple: make your own immutable types!
  - choose names, don't need to remember positions
- recordclass: mutable equivalent of namedtuple
  - need to install with "pip install recordclass"

References

- motivation: faster and allows centralized update
- gotchas: mutating a parameter affects arguments
- is operation: do two variables refer to the same object?