

# **[301] Objects/References**

Tyler Caraza-Harter

# Learning Objectives Today

## More data types

- tuple (immutable list)
- custom types: creating objects from namedtuple and recordclass

## References

- Motivation
- “is” vs “==”
- Gotchas (interning and argument modification)

### Read:

- Downey Ch 10 ("Objects and Values" and "Aliasing")
- Downey Ch 12

### Announcement:

- My office hour today is cancelled
- Tomorrow: 1-5pm (an extra hour)

# Today's Outline

## New Types

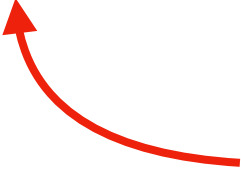
- **tuple**
- namedtuple
- recordclass

## References

- motivation
- unintentional argument modification
- “is” vs. “==”

# Tuple Type

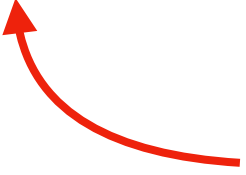
```
nums_list = [200, 100, 300]  
nums_tuple = (200, 100, 300)
```



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

# Tuple Type

```
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?**

# Tuple Type

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

## Like a list

- for loop, indexing, slicing, other methods

## Unlike a list:

- immutable (like a string)

# Tuple Type

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

```
print(nums_list[2])  
print(nums_tuple[2])
```

## Like a list

- for loop, **indexing**, slicing, other methods

## Unlike a list:

- immutable (like a string)

# Tuple Type

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

```
print(nums_list[2])  
print(nums_tuple[2])
```

both of these print 300

## Like a list

- for loop, **indexing**, slicing, other methods

## Unlike a list:

- immutable (like a string)



# Tuple Type

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

```
nums_list[0] = 22  
nums_tuple[0] = 22
```

Like a list

- for loop, indexing, slicing, other methods

Unlike a list:

- **immutable** (like a string)

# Tuple Type

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

```
nums_list[0] = 22  
nums_tuple[0] = 22
```

changes list to  
[22, 100, 300]



Like a list

- for loop, indexing, slicing, other methods

Unlike a list:

- **immutable** (like a string)

# Tuple Type

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

```
nums_list[0] = 22  
nums_tuple[0] = 22
```

**changes list to**  
[22, 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)

# Tuple Type

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

```
nums_list[0] = 22  
nums_tuple[0] = 22
```

changes list to [22, 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)

# Today's Outline

## New Types

- tuple
- **namedtuple**
- recordclass

## References

- motivation
- unintentional argument modification
- “is” vs. “==”

# Organizing Attributes

Often, we have **entities/objects** in programming with many **attributes**. E.g., a tornado. Or a **person**:

- first name, last name
- birth date
- SSN
- address
- phone number



# Organizing Attributes

Often, we have **entities/objects** in programming with many **attributes**. E.g., a tornado. Or a **person**:

- first name, last name
- birth date
- SSN
- address
- phone number

One representation strategy: dictionaries

```
person = {  
    "lname": "Turing", "fname": Alan, ...  
}
```

# Organizing Attributes

Often, we have **entities/objects** in programming with many **attributes**. E.g., a tornado. Or a **person**:

- first name, last name
- birth date
- SSN
- address
- phone number

One representation strategy: dictionaries

```
person = {  
    "lname": "Turing", "fname": Alan, ...  
}  
print(person["fname"] + " " + person["lname"])
```

# Organizing Attributes

Often, we have **entities/objects** in programming with many **attributes**. E.g., a tornado. Or a **person**:

- first name, last name
- birth date
- SSN
- address
- phone number

## Problem with using dicts:

- it's verbose (always typing quotes)
- error prone (same attributes not enforced)

One representation strategy: dictionaries

```
person = {  
    "lname": "Turing", "fname": Alan, ...  
}  
print(person["fname"] + " " + person["lname"])
```

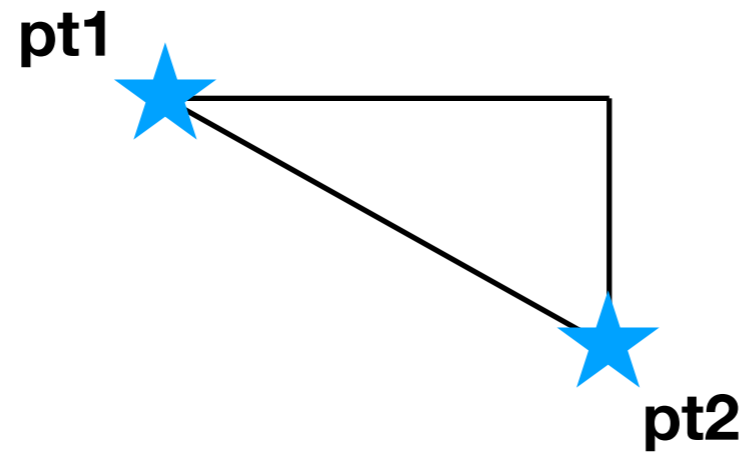
# Tuples, with and without names

regular tuples (**remember** x then y)

```
pt1 = (50,60)
```

```
pt2 = (90,10)
```

```
distance = ((pt1[0]-pt2[0])**2 + (pt1[1]-pt2[1])**2) ** 0.5
```



# Tuples, with and without names

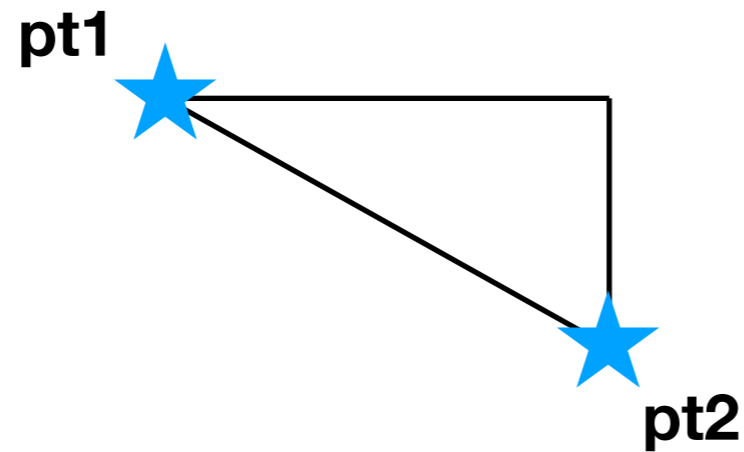
regular tuples (**remember** x then y)

```
pt1 = (50,60)
```

```
pt2 = (90,10)
```

```
distance = ((pt1[0]-pt2[0])**2 + (pt1[1]-pt2[1])**2) ** 0.5
```

pt1[0] is x



# Tuples, with and without names

regular tuples (**remember** x then y)

```
pt1 = (50,60)
```

```
pt2 = (90,10)
```

```
distance = ((pt1[0]-pt2[0])**2 + (pt1[1]-pt2[1])**2) ** 0.5
```

pt1[0] is x

---

```
from collections import namedtuple
```

need to import namedtuple  
(not there by default)

# Tuples, with and without names

regular tuples (**remember** x then y)

```
pt1 = (50,60)
```

```
pt2 = (90,10)
```

```
distance = ((pt1[0]-pt2[0])**2 + (pt1[1]-pt2[1])**2) ** 0.5
```

pt1[0] is x

---

```
from collections import namedtuple
```

```
Point = namedtuple("Point", ["x", "y"])
```



Point is a  
new type



"Point" is the  
type's name



A Point will  
have an x and y

# Tuples, with and without names

regular tuples (**remember x then y**)

```
pt1 = (50,60)
pt2 = (90,10)
distance = ((pt1[0]-pt2[0])**2 + (pt1[1]-pt2[1])**2) ** 0.5
```

pt1[0] is x

---

```
from collections import namedtuple
```

```
Point = namedtuple("Point", ["x", "y"])
```



Point is a  
new type



"Point" is the  
type's name



A Point will  
have an x and y

```
>>> L = list()
>>> type(L)
<class 'list'>
```

```
>>> type(list)
<class 'type'>
```

```
>>> type(Point)
<class 'type'>
```



# Tuples, with and without names

```
pt1 = (50,60)
pt2 = (90,10)
distance = ((pt1[0]-pt2[0])**2 + (pt1[1]-pt2[1])**2) ** 0.5
```

regular tuples (**remember x then y**)

pt1[0] is x

---

```
from collections import namedtuple
```

```
Point = namedtuple("Point", ["x", "y"])
```



Point is a  
new type



"Point" is the  
type's name



A Point will  
have an x and y

```
>>> L = list()
>>> type(L)
<class 'list'>
```

```
>>> type(list)
<class 'type'>
```

```
>>> type(Point)
<class 'type'>
```

Point is now a datatype, like a list or dict.  
Just like dict(...) and list(...) create new instances,  
Point(...) will create new instances

# Tuples, with and without names

regular tuples (**remember** x then y)

```
pt1 = (50,60)
```

```
pt2 = (90,10)
```

```
distance = ((pt1[0]-pt2[0])**2 + (pt1[1]-pt2[1])**2) ** 0.5
```

pt1[0] is x

---

```
from collections import namedtuple
```

```
Point = namedtuple("Point", ["x", "y"])
```

```
pt1 = Point(50,60)
```

x y

# Tuples, with and without names

regular tuples (**remember x then y**)

```
pt1 = (50,60)
```

```
pt2 = (90,10)
```

```
distance = ((pt1[0]-pt2[0])**2 + (pt1[1]-pt2[1])**2) ** 0.5
```

pt1[0] is x

---

```
from collections import namedtuple
```

```
Point = namedtuple("Point", ["x", "y"])
```

```
pt1 = Point(50,60)
```

```
pt2 = Point(x=90, y=10)
```

x

y

# Tuples, with and without names

regular tuples (**remember** x then y)

```
pt1 = (50,60)
```

```
pt2 = (90,10)
```

```
distance = ((pt1[0]-pt2[0])**2 + (pt1[1]-pt2[1])**2) ** 0.5
```

pt1[0] is x

---

```
from collections import namedtuple
```

```
Point = namedtuple("Point", ["x", "y"])
```

```
pt1 = Point(50,60)
```

```
pt2 = Point(x=90, y=10)
```

```
distance = ((pt1.x - pt2.x)**2 + (pt1.y - pt2.y) ** 2) ** 0.5
```

don't need to remember anything (e.g., "x" is first)

# Tuples, with and without names

regular tuples (**remember x then y**)

```
pt1 = (50,60)
pt2 = (90,10)
distance = ((pt1[0]-pt2[0])**2 + (pt1[1]-pt2[1])**2) ** 0.5
```

pt1[0] is x

---

```
from collections import namedtuple
```

```
Point = namedtuple("Point", ["x", "y"])
```

```
pt1 = Point(50,60)
```

```
pt2 = Point(x=90, y=10)
```

```
distance = ((pt1.x - pt2.x)**2 + (pt1.y - pt2.y) ** 2) ** 0.5
```

```
>>> pt1.x = 3
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

**note that nametuples  
are also immutable**

# Today's Outline

## New Types

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

## References

- motivation
- unintentional argument modification
- “is” vs. “==”

# Today's Outline

## New Types

- tuple
- namedtuple
- **recordclass**

## References

- motivation
- unintentional argument modification
- “is” vs. “==”

# recordclass example

```
>>> from recordclass import recordclass
```

**module is recordclass**

**so is function**



# recordclass example

```
>>> from recordclass import recordclass  
>>> Point = recordclass("Point", ["x", "y"])
```



```
Point = namedtuple("Point", ["x", "y"])
```

# recordclass example

```
>>> from recordclass import recordclass
>>> Point = recordclass("Point", ["x", "y"])
>>> pt1 = Point(0,0)
>>> pt1
Point(x=0, y=0)
```

# recordclass example

```
>>> from recordclass import recordclass
>>> Point = recordclass("Point", ["x", "y"])
>>> pt1 = Point(0,0)
>>> pt1
Point(x=0, y=0)
>>> pt1.x = 5
>>> pt1.y = 6
```

**mutations**

# recordclass example

```
>>> from recordclass import recordclass
>>> Point = recordclass("Point", ["x", "y"])
>>> pt1 = Point(0,0)
>>> pt1
Point(x=0, y=0)
>>> pt1.x = 5
>>> pt1.y = 6
>>> pt1
Point(x=5, y=6)
```

# recordclass example

```
>>> from recordclass import recordclass
>>> Point = recordclass("Point", ["x", "y"])
>>> pt1 = Point(0,0)
>>> pt1
Point(x=0, y=0)
>>> pt1.x = 5
>>> pt1.y = 6
>>> pt1
Point(x=5, y=6)
```

Note: recordclass does not come with Python.  
You must install it yourself.

# Aside: installing packages

There are many Python packages available on PyPI

- <https://pypi.org/>
- short for Python Package Index

Installation example (from terminal):

```
pip install recordclass
```

# Aside: installing packages

There are many Python packages available on PyPI

- <https://pypi.org/>
- short for Python Package Index

Installation example (from terminal):

```
pip install recordclass
```

Anaconda is just Python with a bunch of packages related to data science and quantitative work pre-installed.

# Today's Outline

## New Types

- tuple
- namedtuple
- recordclass


## References

- motivation
- unintentional argument modification
- “is” vs. “==”



# Mental Model for State (v1)

**Code:**

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

---


**State:**

**x**

**y**

# Mental Model for State (v1)

**Code:**

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

---

**State:**

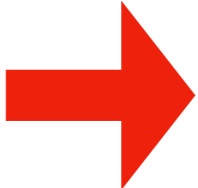
**x**

**y**

# Mental Model for State (v1)

**Code:**

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



---

**State:**

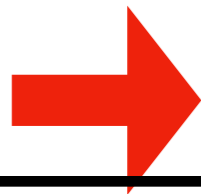
x hello

y hello

# Mental Model for State (v1)

**Code:**

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



---

**State:**

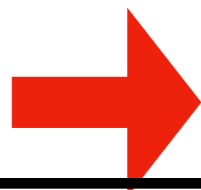
**x** hello

**y** hello world

# Mental Model for State (v1)

**Code:**

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



Common mental model

- correct for immutable types
- PythonTutor uses for strings, etc

---

**State:**

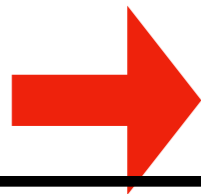
x hello

y hello world

# Mental Model for State (v1)

**Code:**

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



Common mental model

- correct 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*

**x** 

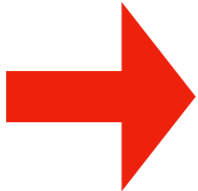
**y** 

*objects*

*(a variable is one kind of reference)*

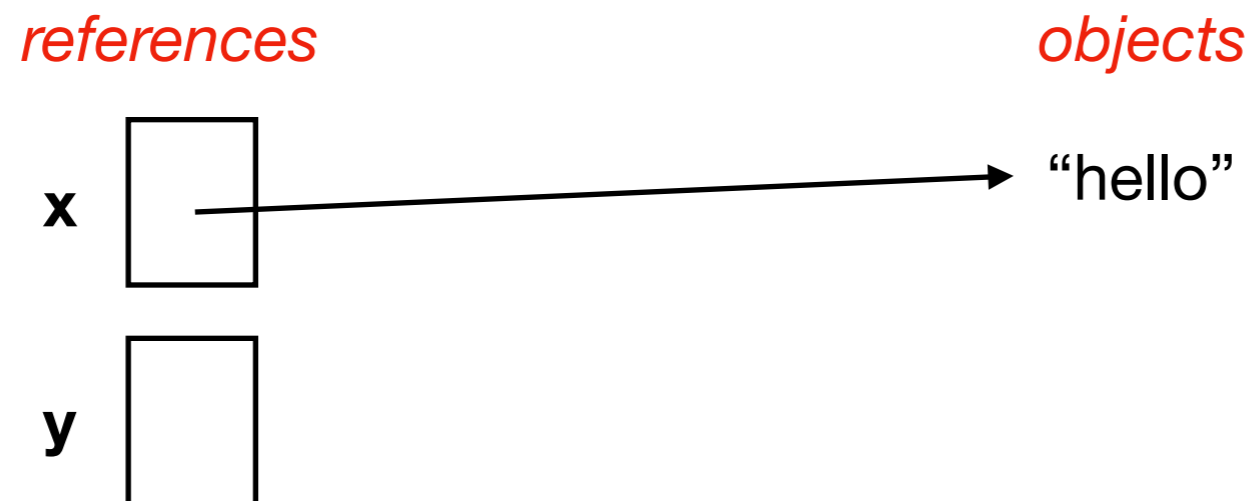
# Mental Model for State (v2)

**Code:**

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

---

**State:**



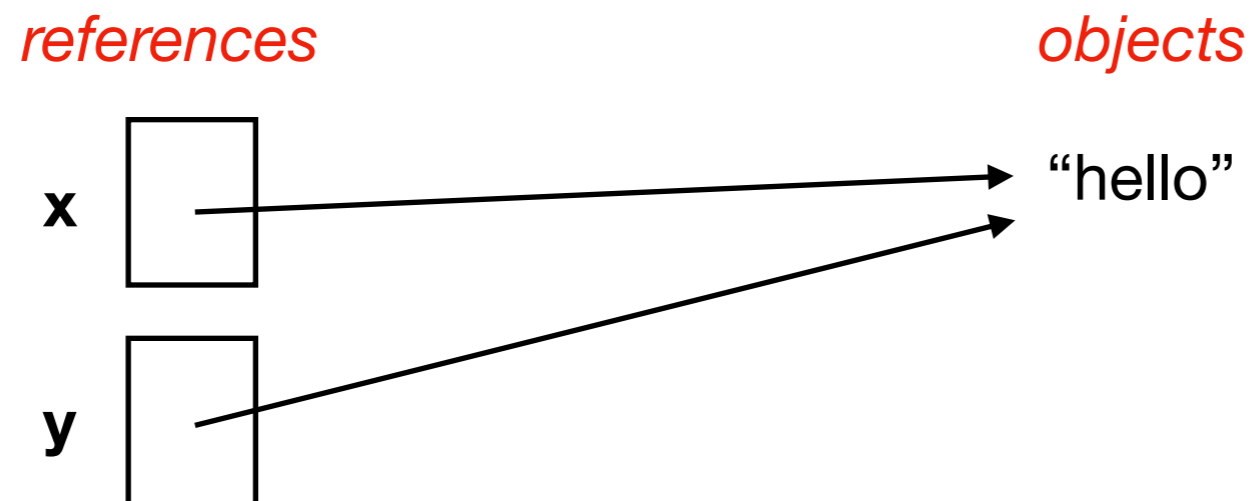


# Mental Model for State (v2)

**Code:**

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

**State:**



# Mental Model for State (v2)

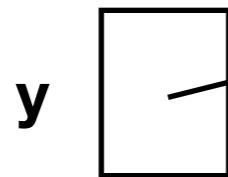
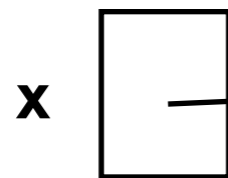
**Code:**

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

---

**State:**

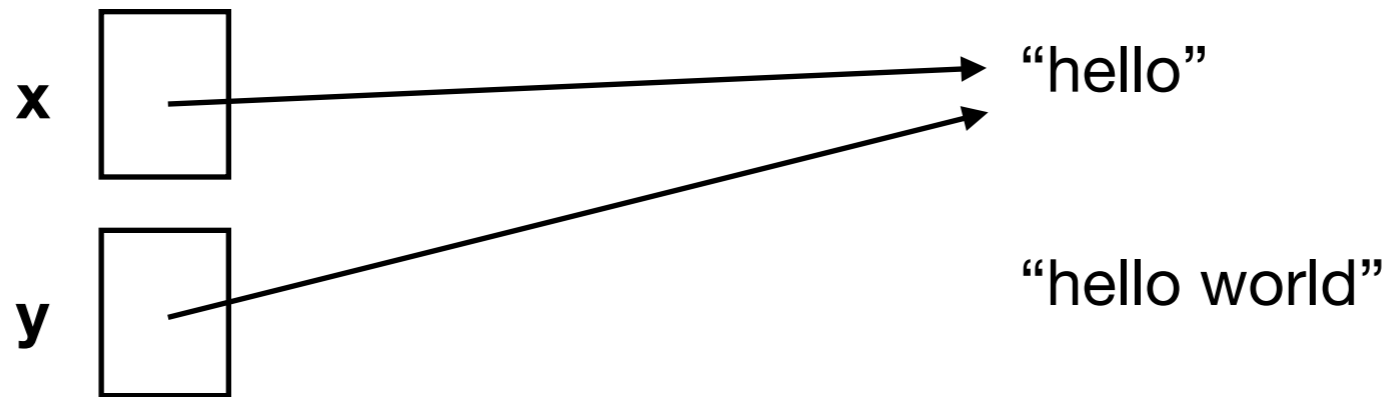
*references*



*objects*

"hello"

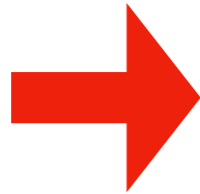
"hello world"



# Mental Model for State (v2)

**Code:**

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

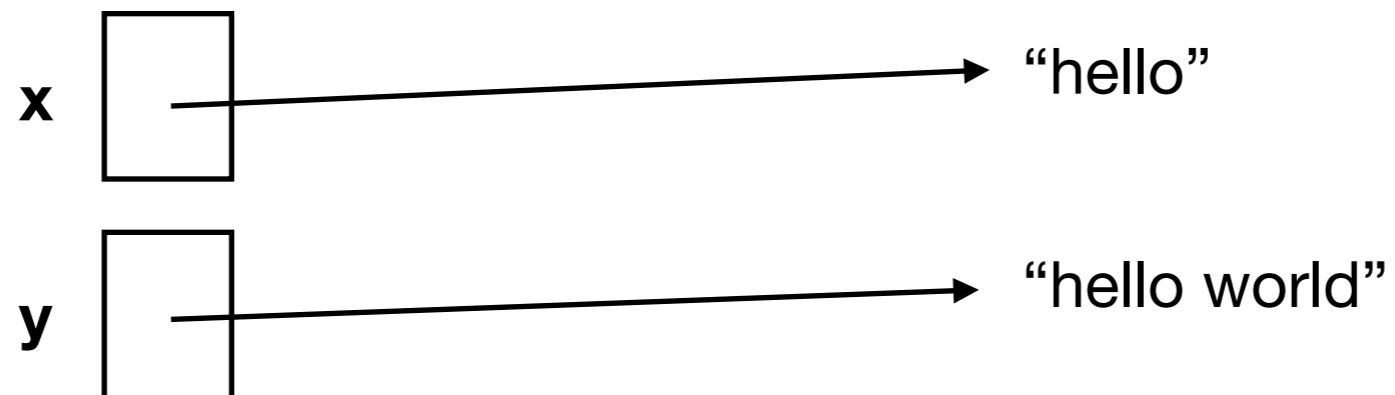


---

**State:**

*references*

*objects*



# Today's Outline

## New Types

- tuple
- namedtuple
- recordclass

## References


- **motivation**
- unintentional argument modification
- “is” vs. “==”

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

Why not follow the original strategy we learned (i.e., boxes of data with labels)?

# Reason 1: Performance

**Code:**

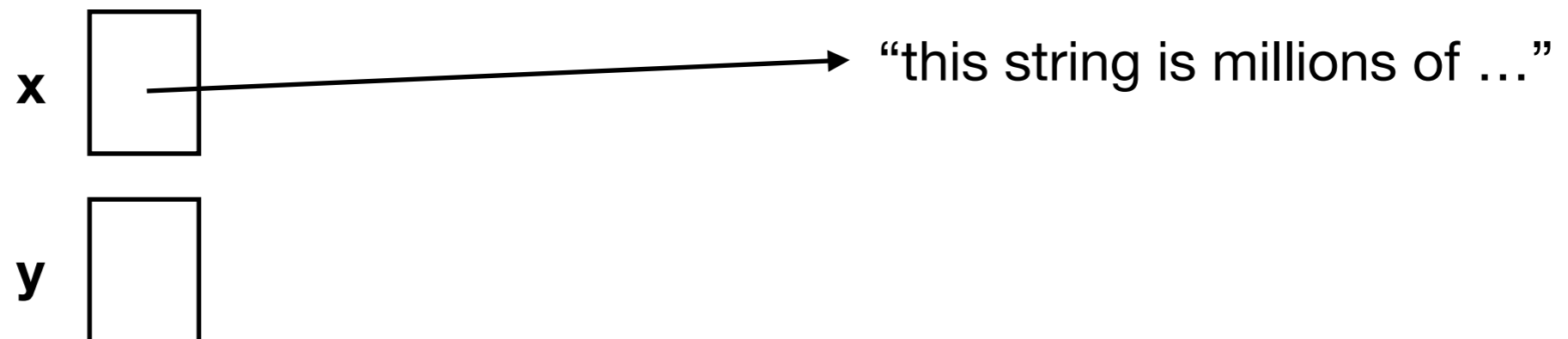
 `x = "this string is millions of characters..."`  
`y = x # this is fast!`

---

**State:**

*references*

*objects*



# Reason 1: Performance

**Code:**

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

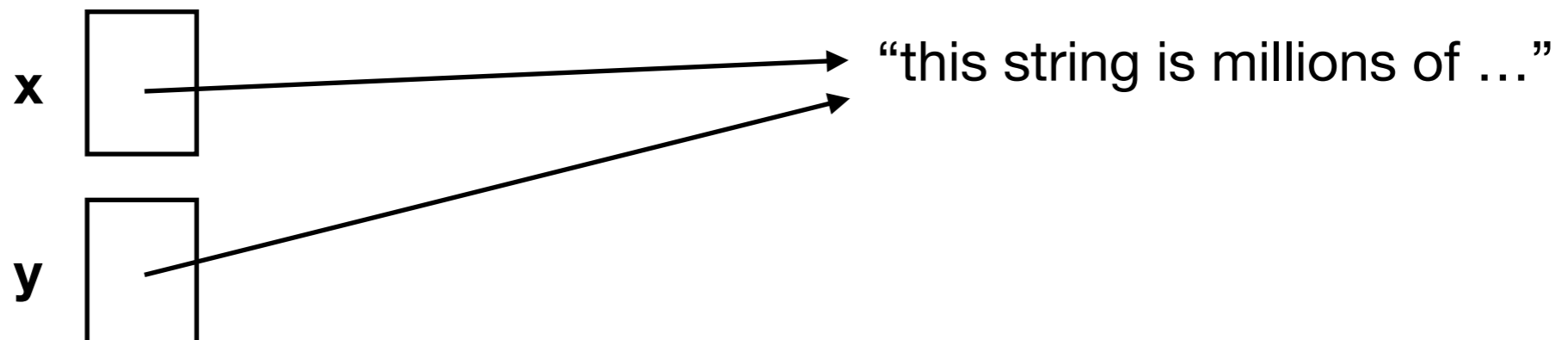


---

**State:**

*references*

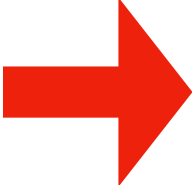
*objects*



# 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 = 31  
print("Winner age:", winner.age)
```

---

**State:**

*references*

alice

bob

winner

*objects*



# 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 = 31
```

```
print("Winner age:", winner.age)
```

---

**State:**

*references*

alice

bob

winner

*objects*

name:Alice | score:10 | age:30

# 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)
```

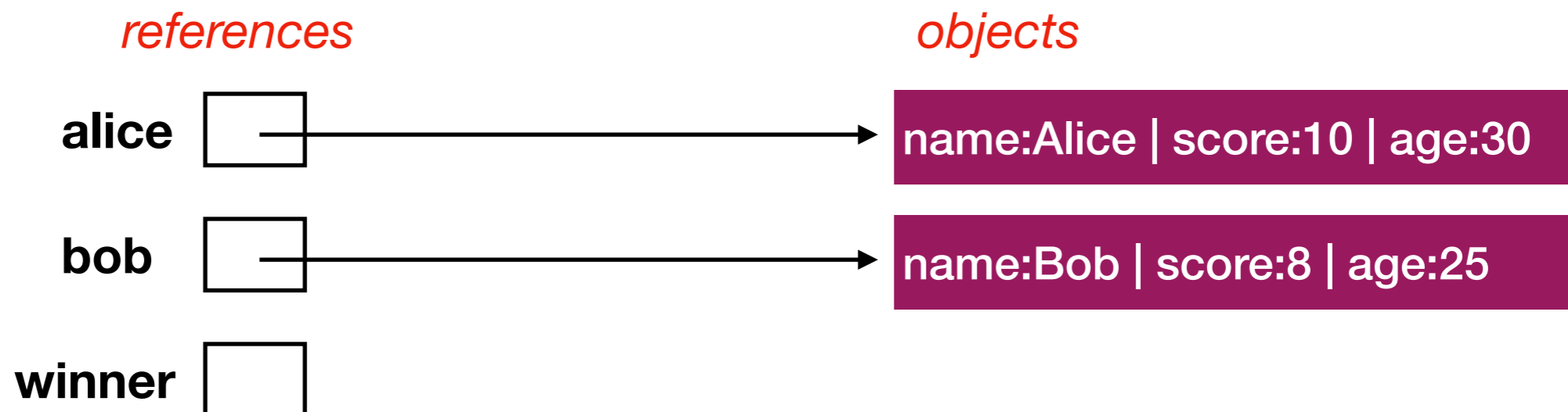
```
winner = alice
```

```
alice.age = 31
```

```
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)
```

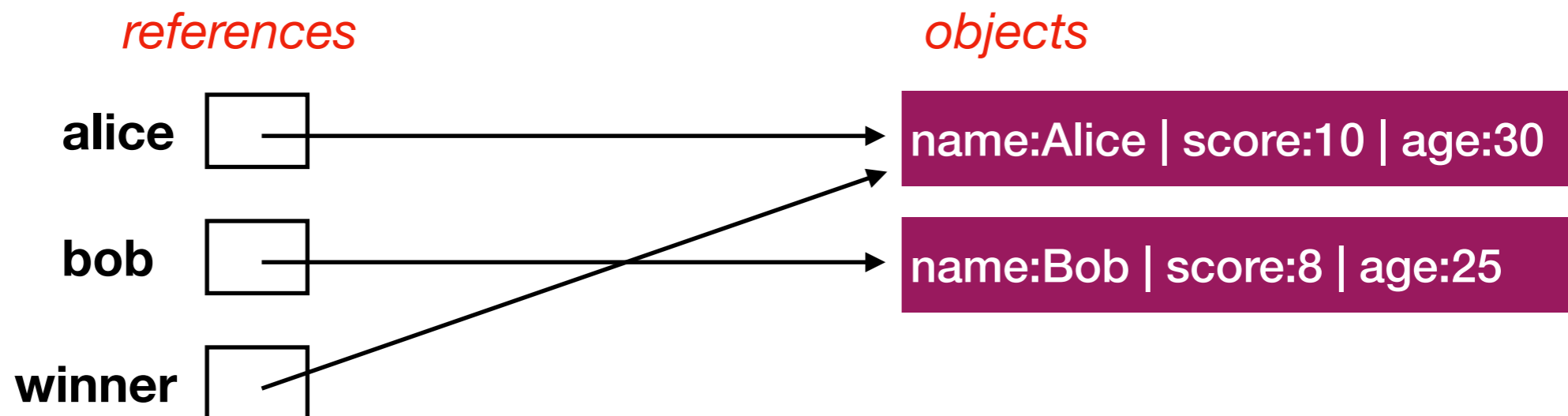
```
bob = Person(name="Bob", score=8, age=25)
```

```
winner = alice
```



```
alice.age = 31  
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)
```

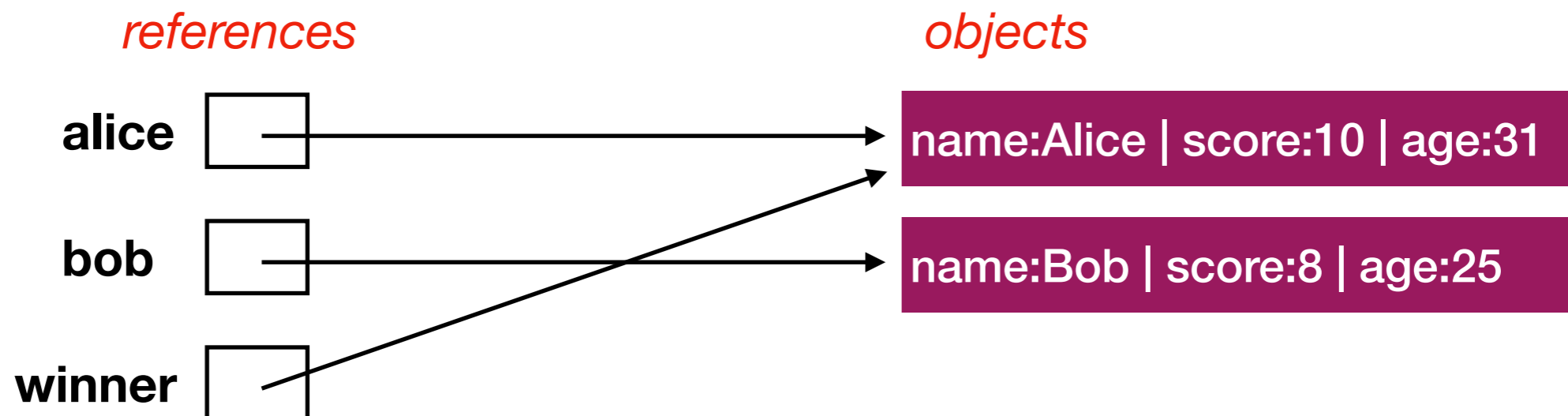
```
bob = Person(name="Bob", score=8, age=25)
```

```
winner = alice
```

```
alice.age = 31
```

```
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)
```

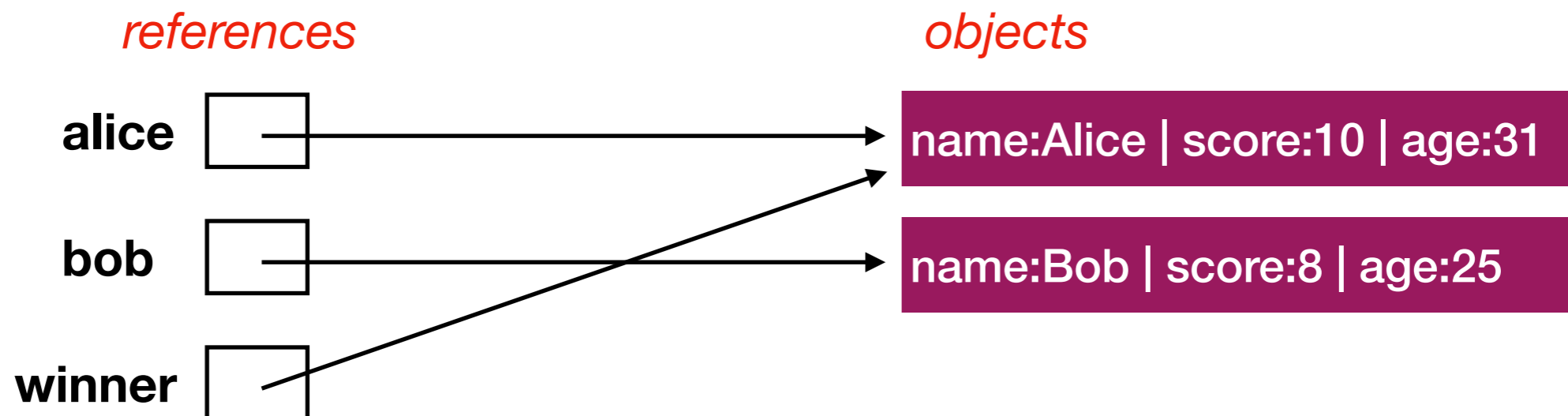
```
winner = alice
```

```
alice.age = 31
```

```
print("Winner age:", winner.age)
```

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

State:



# Today's Outline

## New Types

- tuple
- namedtuple
- recordclass

## References

- motivation
- **unintentional argument modification**
- “is” vs. “==”

# References and Arguments/Parameters

## Python Tutor

- correctly illustrates references with an arrow for mutable types
- thinking carefully about a few examples will prevent many debugging headaches...

# Example 1: reassign parameter

```
def test(items, x):  
    x *= 3  
    print("in test:", items, x)  
  
words = ['hello', 'world']  
letter = 'w'  
print("before:", words, letter)  
test(words, letter)  
print("after:", words, letter)
```



## Example 2: modify list

```
def test(items, x):  
    items.append(x)  
    print("in test:", items, x)  
  
words = ['hello', 'world']  
letter = 'w'  
print("before:", words, letter)  
test(words, letter)  
print("after:", words, letter)
```

# Example 3: reassign new list to param

```
def test(items, x):  
    items = items + [x]  
    print("in test:", items, x)  
  
words = ['hello', 'world']  
letter = 'w'  
print("before:", words, letter)  
test(words, letter)  
print("after:", words, letter)
```

# Example 4: in-place sort

```
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))
```

# Today's Outline

## New Types

- tuple
- namedtuple
- recordclass

## References

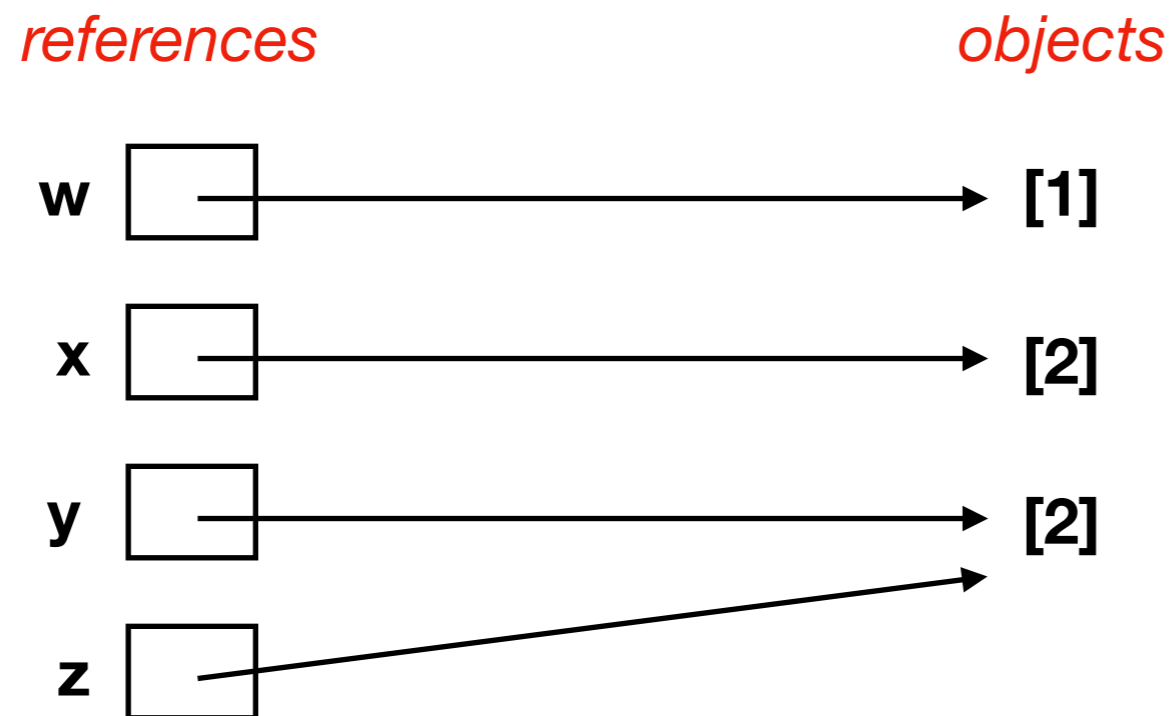
- motivation
- unintentional argument modification
- **“is” vs. “==”**

# Equal and is

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

---

**State:**



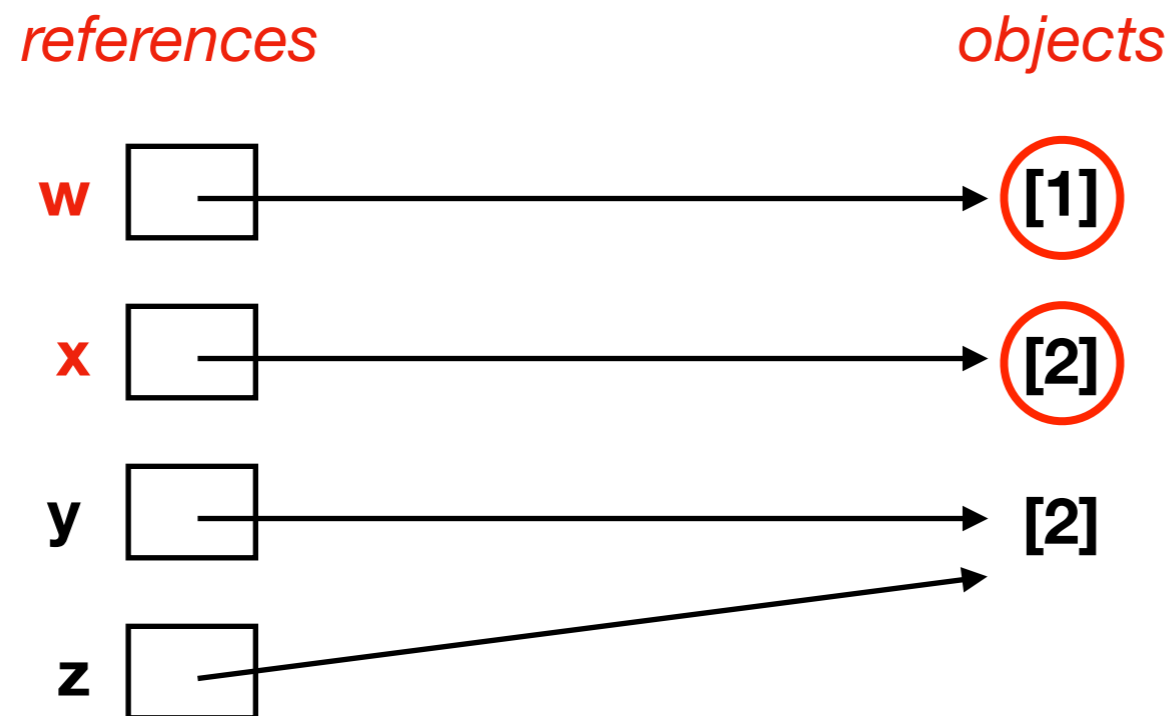
# Equal and is

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

**W == X**

---

**State:**



# Equal and is

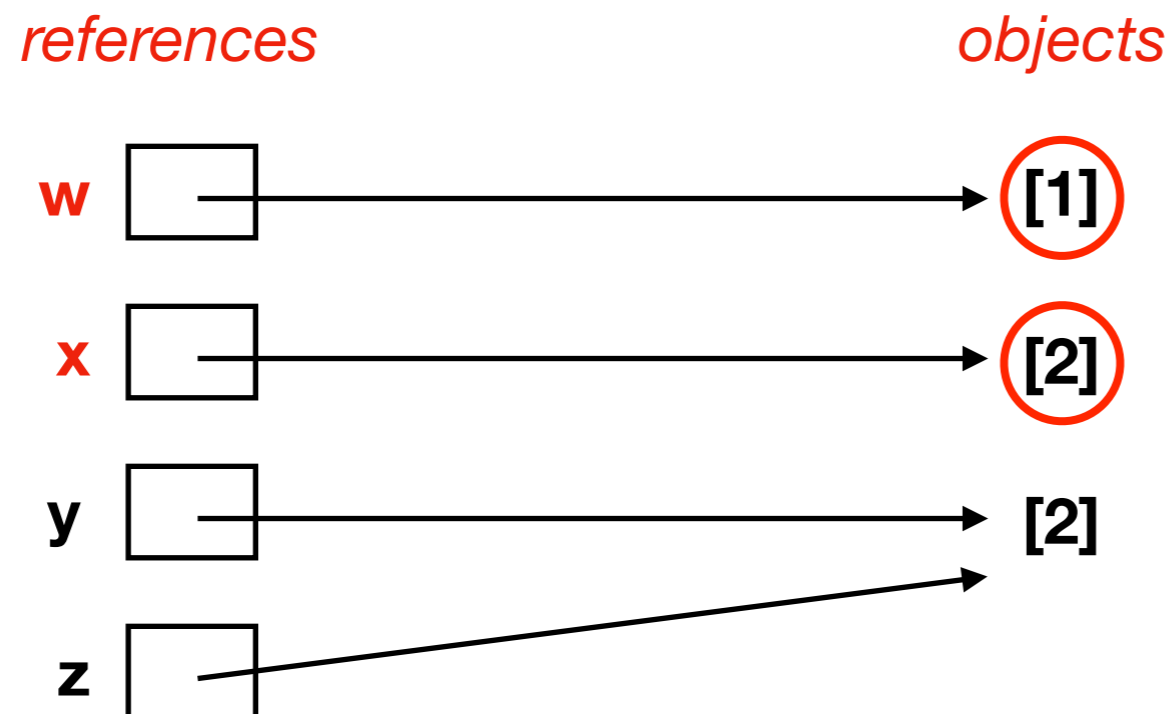
```
w = [1]  
x = [2]  
y = [2]  
z = y
```

**w == x**

**False**

---

**State:**



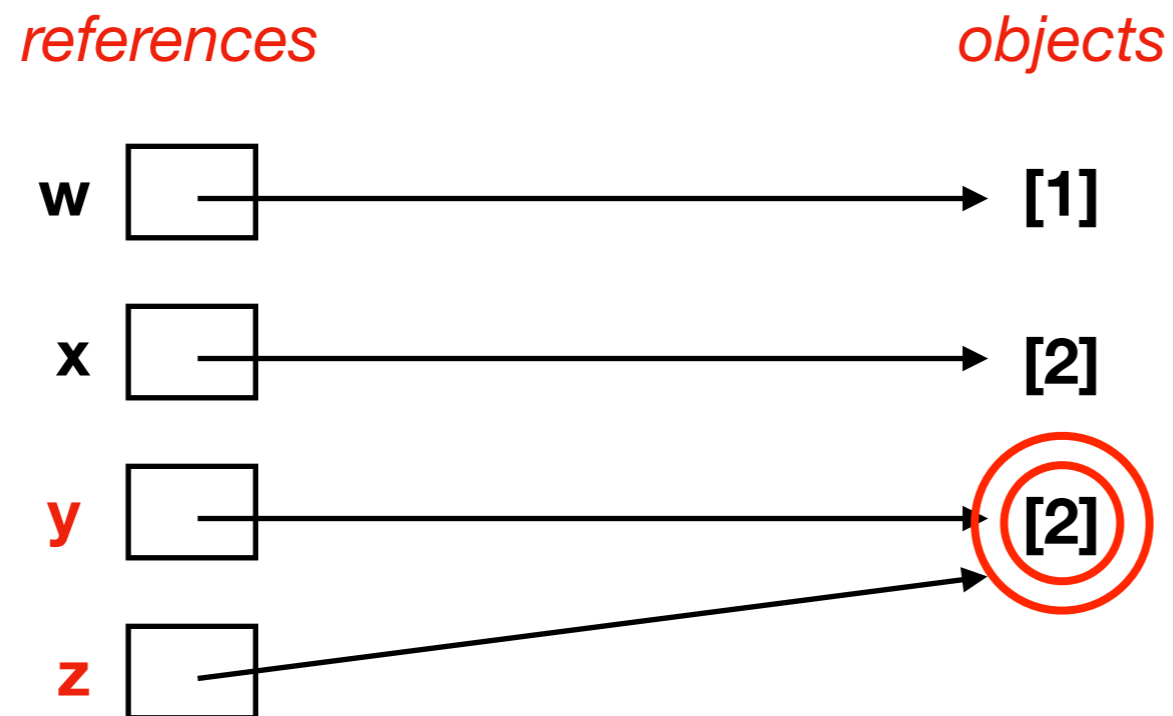
# Equal and is

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

**y == z**

---

**State:**





# Equal and is

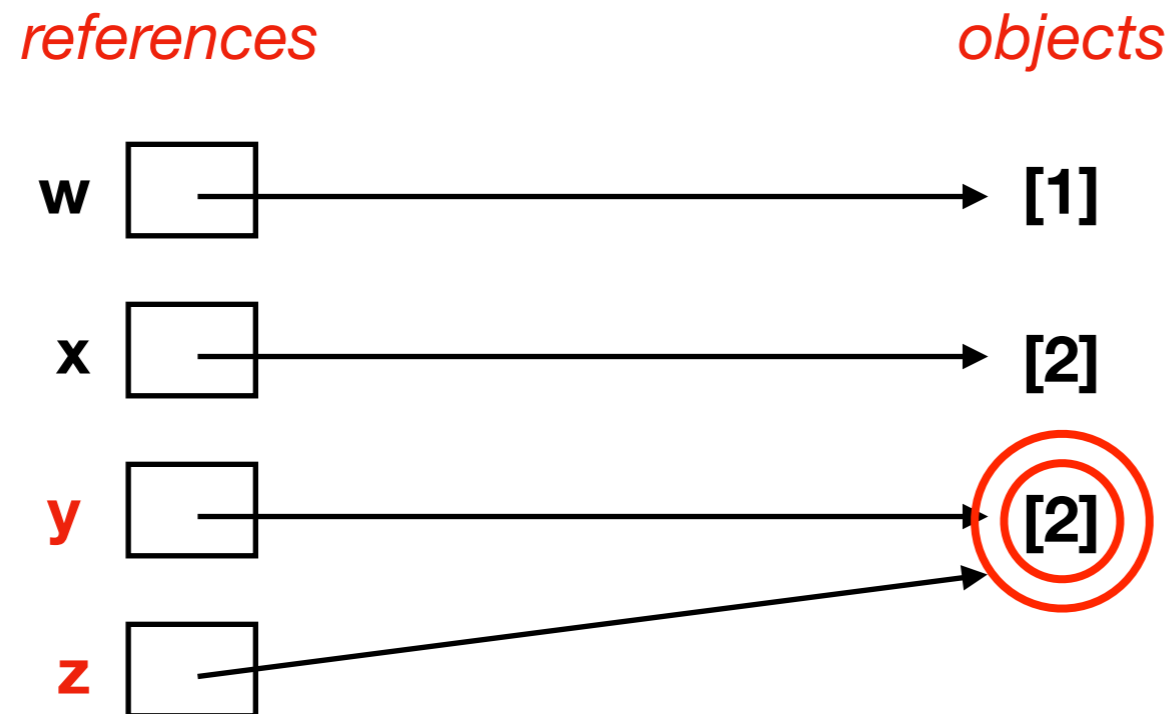
```
w = [1]  
x = [2]  
y = [2]  
z = y
```

**y == z**

**True**

---

**State:**



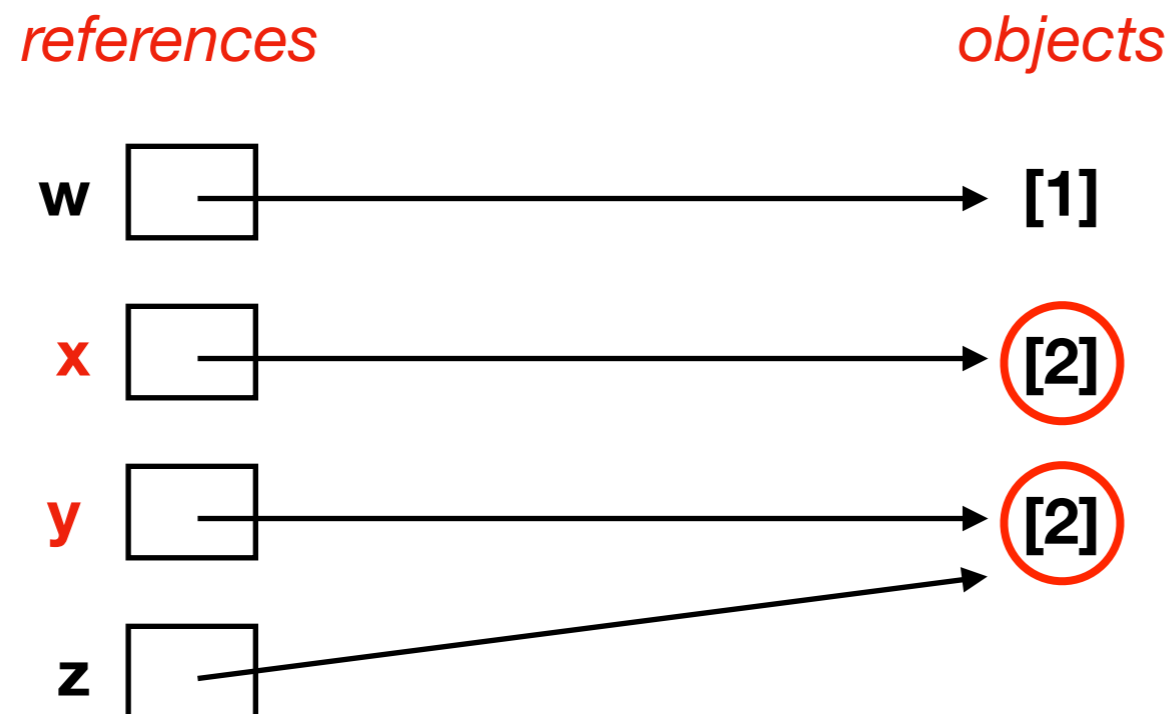
# Equal and is

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

**x == y**

---

**State:**



# Equal and is

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

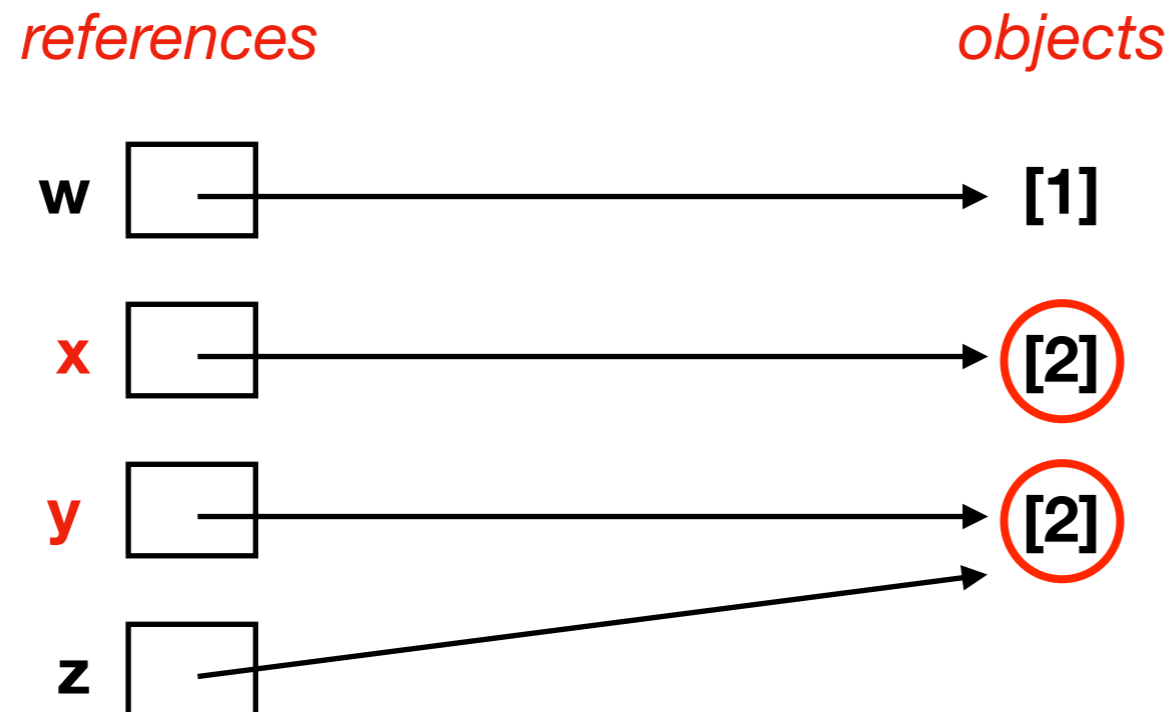
**x == y**

**True**

because x and y refer to equivalent  
(though not identical) values

---

**State:**



# Equal and is

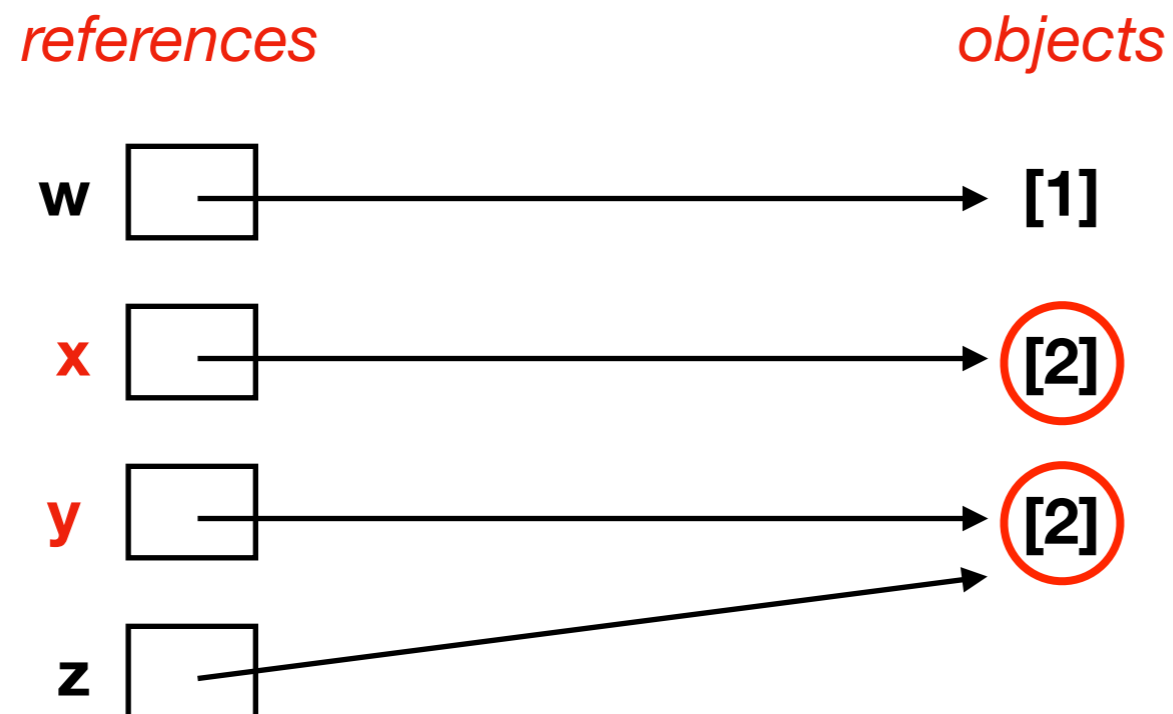
```
w = [1]  
x = [2]  
y = [2]  
z = y
```

**x is y**



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

**State:**



# Equal and is

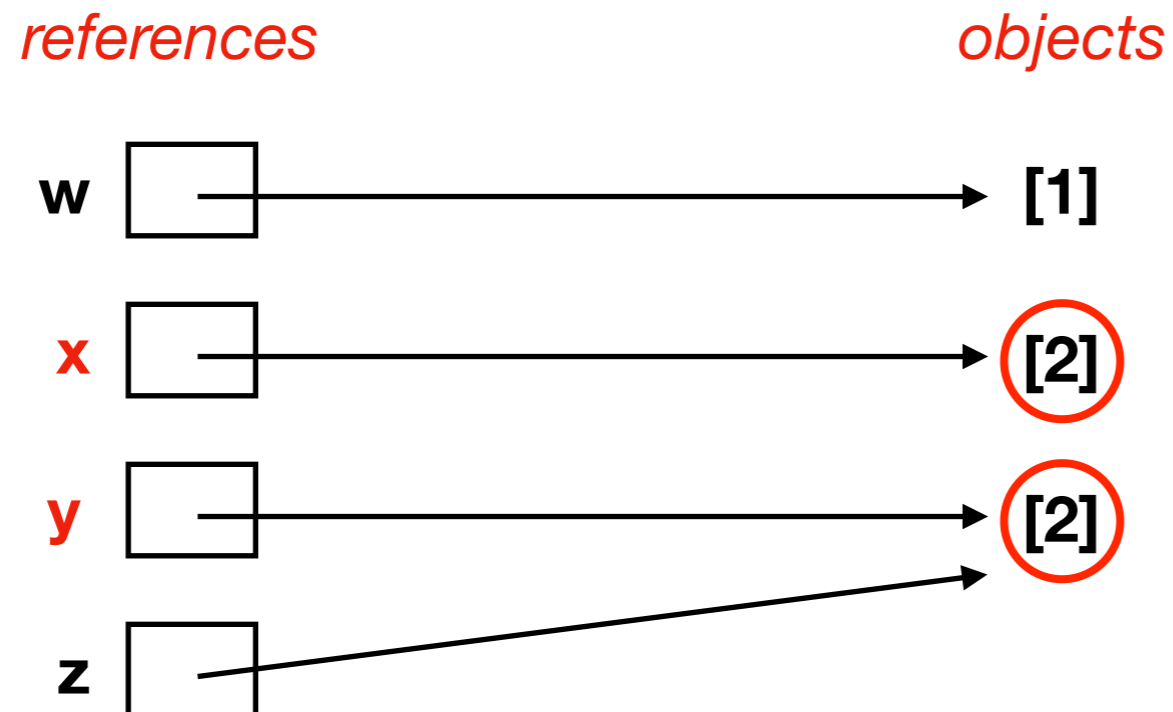
```
w = [1]  
x = [2]  
y = [2]  
z = y
```

**x is y**

**False**

---

**State:**



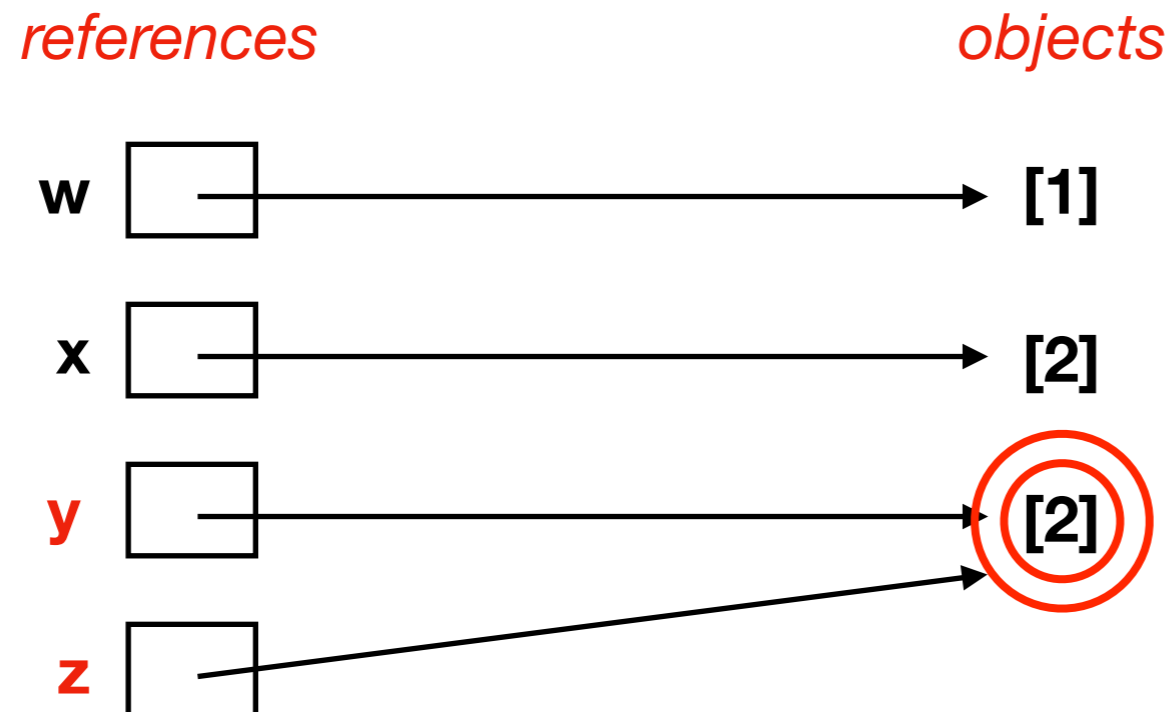
# Equal and is

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

**y is z**

---

**State:**



# Equal and is

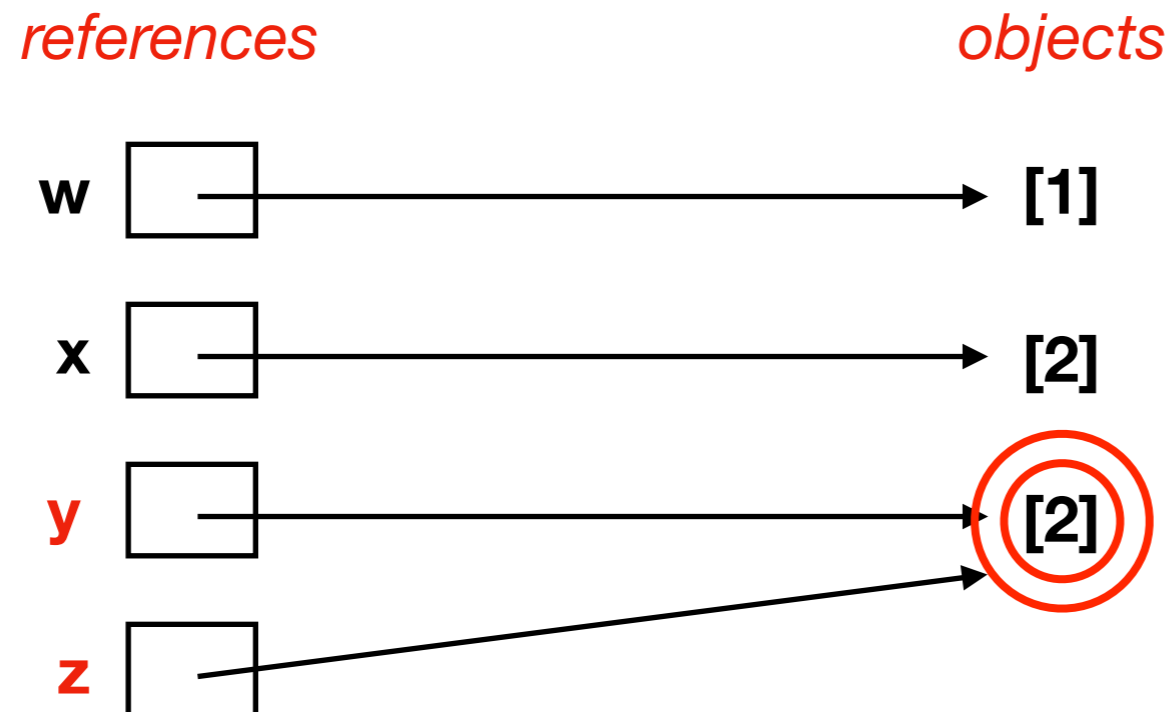
```
w = [1]  
x = [2]  
y = [2]  
z = y
```

**y is z**

**True**

---

**State:**



# Equal and is

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

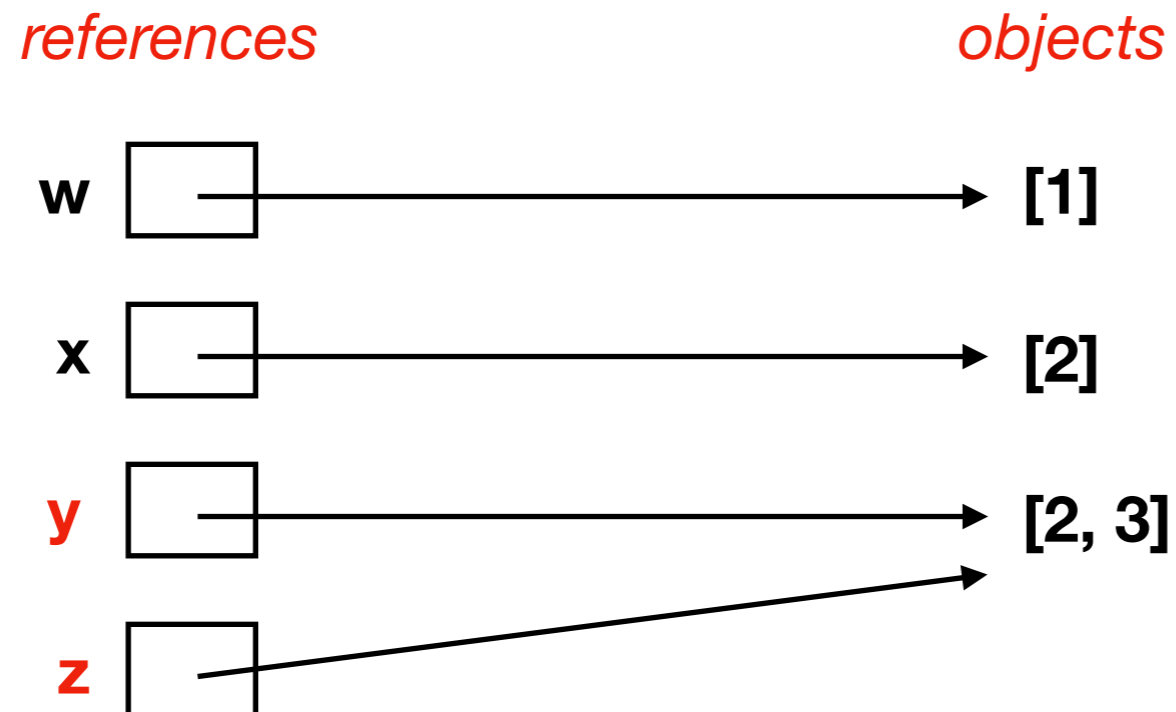
**y is z**

**True**

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

---

## State:





# Equal and is

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

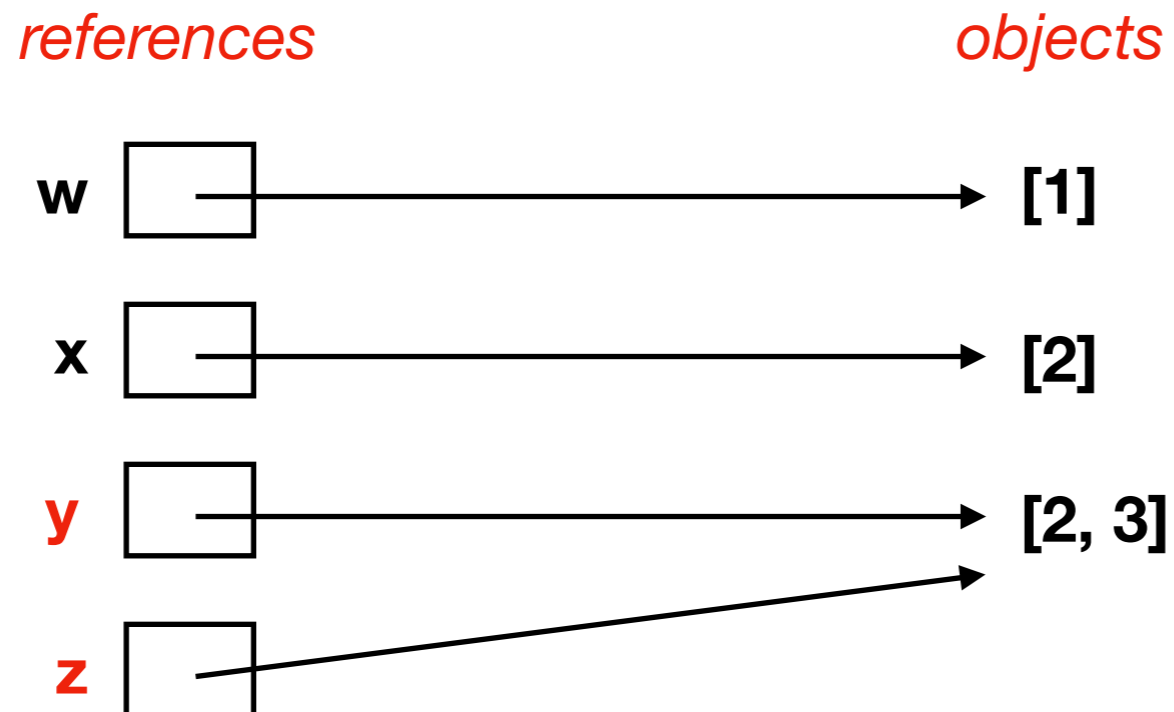
**y is z**

**True**

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

---

## State:



# Be careful with `is`!

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

- **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?