[301] Objects/References

Tyler Caraza-Harter

Test yourself!

1. **what is the type of the following?** `{}`

   A. **dict**

   B. **set**

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

   A. `S[-1] = "."`

   B. `L[len(S)] = S`

3. **which type is immutable?**

   A. `str`

   B. `list`

   C. `dict`

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)

mental model of state will be critical!

Read:

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

Today's Outline

New Types

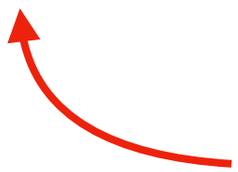
- **tuple**
- namedtuple
- recordclass

References

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

Tuple Sequence

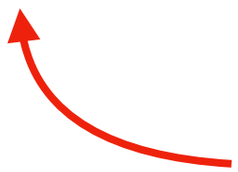
```
nums_list  = [200, 100, 300]
nums_tuple = (200, 100, 300)
```



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

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?

Tuple Sequence

```
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 Sequence

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

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

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 of 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)

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

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] = x
nums_tuple[0] = x
```

changes list to
[300, 100, 300]

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] = x`

✗ `nums_tuple[0] = x`

changes list to [300, 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 Sequence

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

✓ `nums_list[0] = x`

✗ `nums_tuple[0] = x`

changes list to [300, 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:

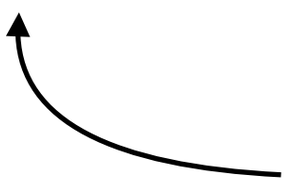
- **immutable** (like a string)

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

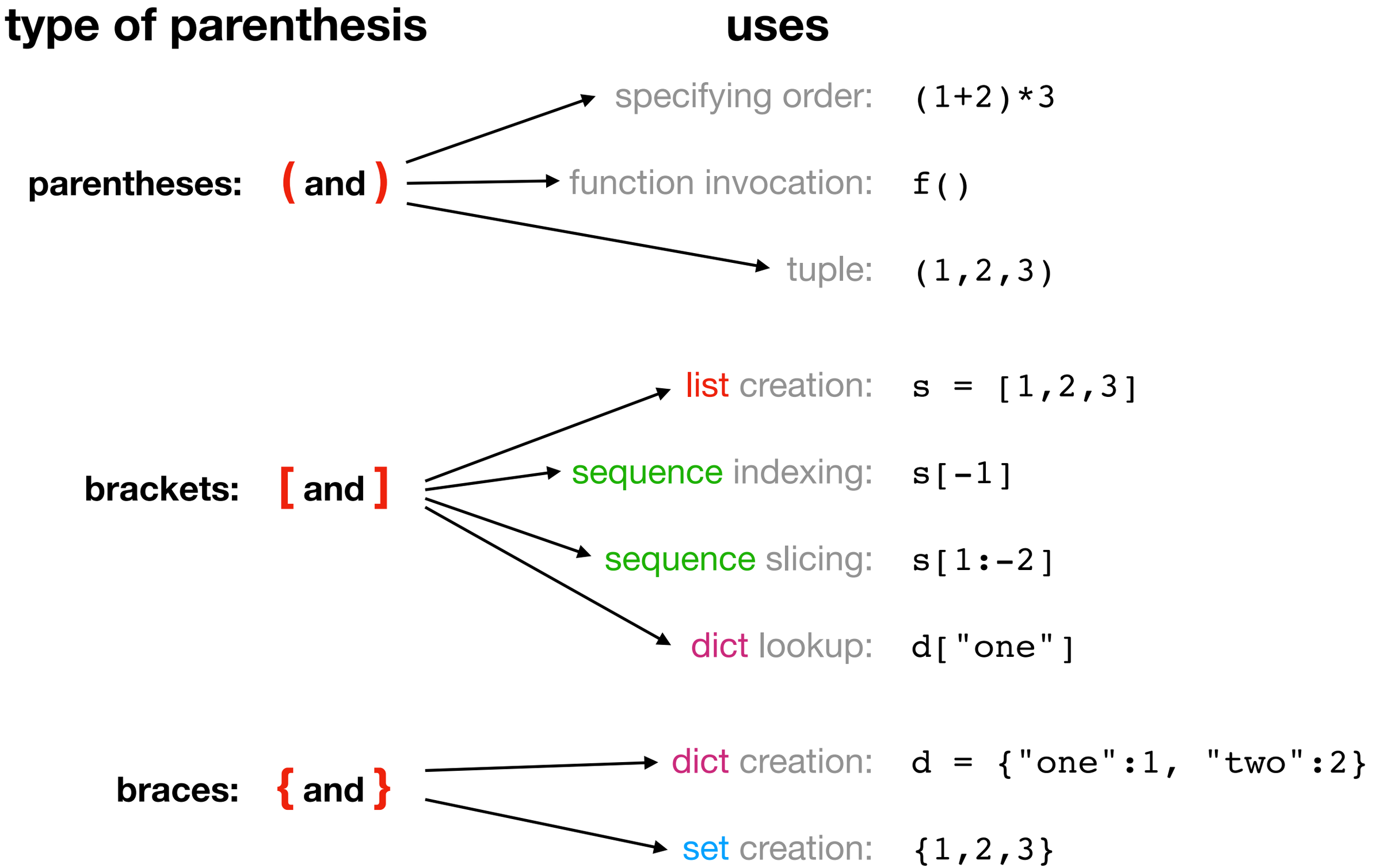
```
buildings = {
   (0,0): “Comp Sci”,
   (0,2): “Psychology”,
   (4,0): “Noland”,
   (1,8): “Van Vleck”
}
```

trying to use x,y coordinates as key

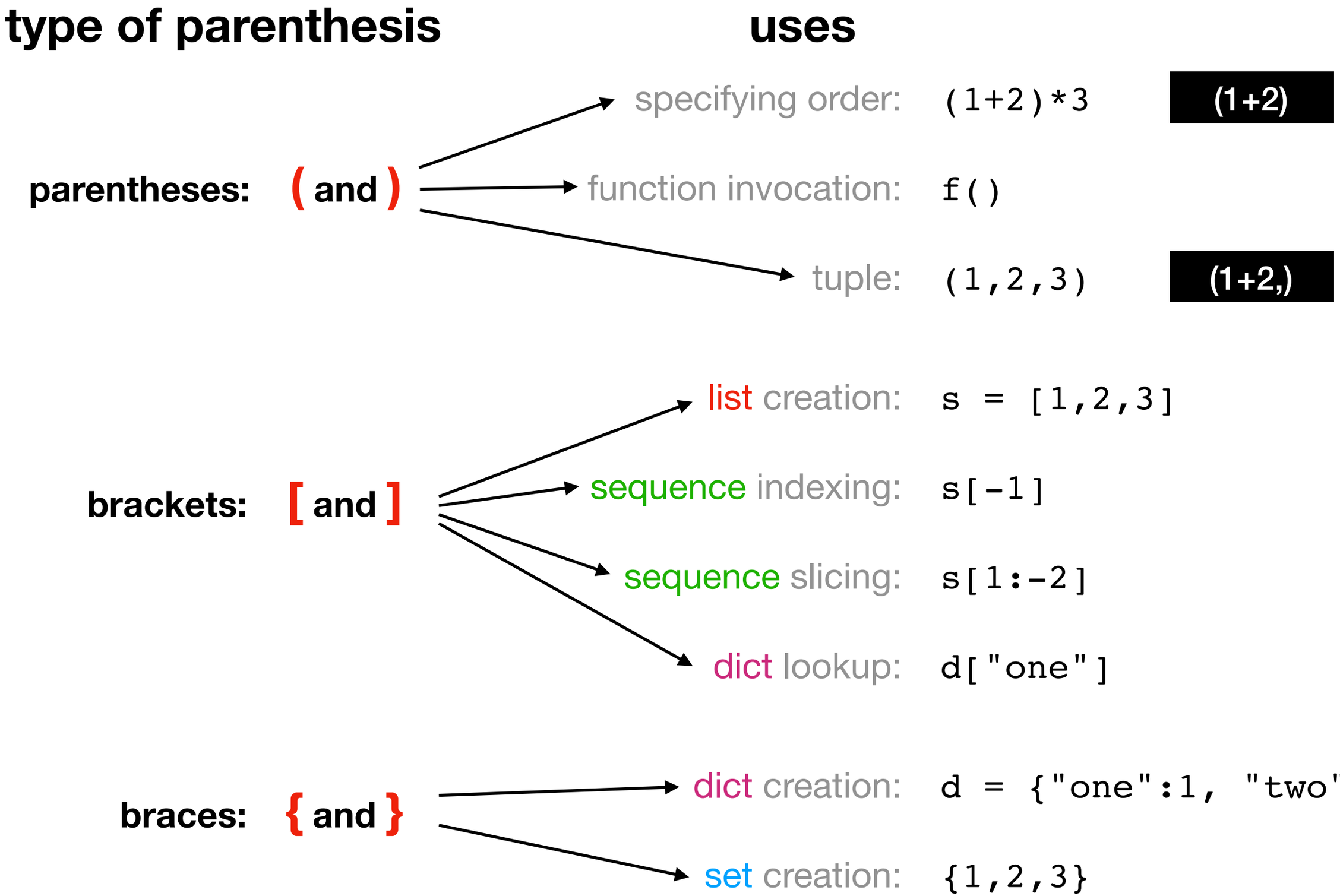
Succeeds!

(with tuples)

A note on parenthetical characters



A note on parenthetical characters



Today's Outline

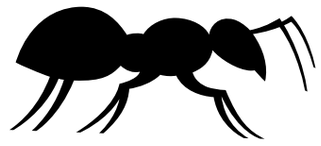
New Types

- tuple
- **namedtuple**
- recordclass

References

- motivation
- unintentional 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

```
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

```
from collections import namedtuple



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

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



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

```
from collections import namedtuple
```

need to import this data struct

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

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


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

```
from collections import namedtuple
```

need to import this data struct

name of that type

creates a new type!

name of that type

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

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

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

```
from collections import namedtuple
```

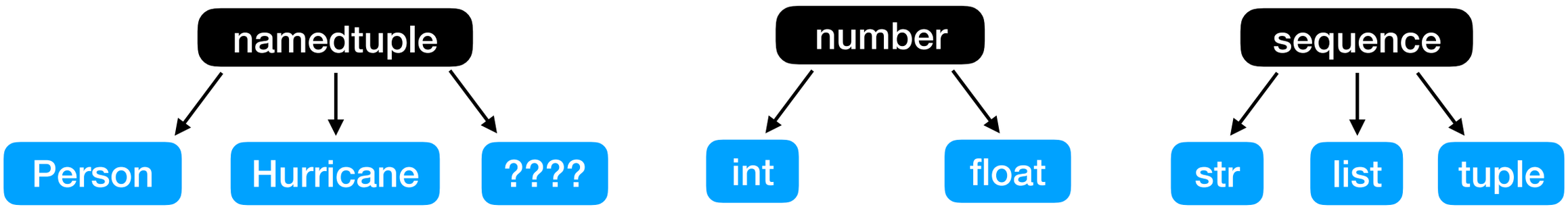
need to import this data struct

name of that type

creates a new type!

name of that type

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




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



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

```python
from collections import namedtuple
```

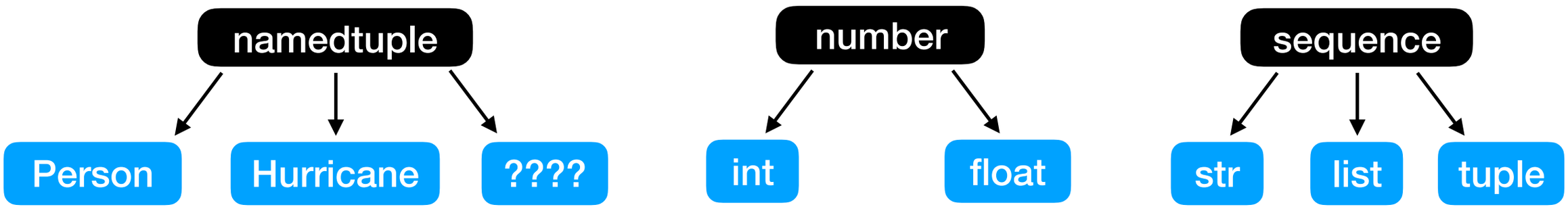
need to import this data struct

name of that type

creates a new type!

name of that type

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

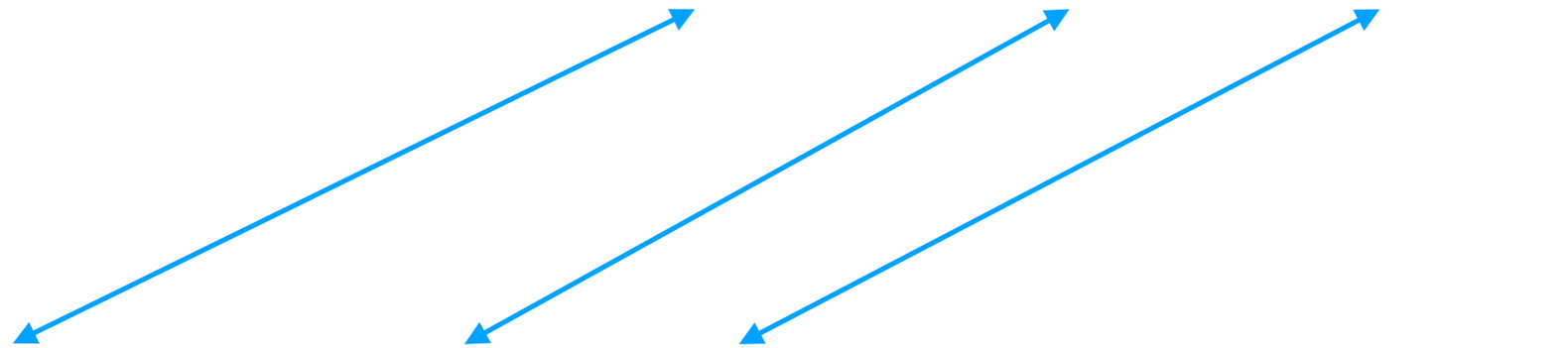



```python
p = Person("Alice", "Anderson", 30)
```

creates a object of type Person
(like **`str(3)`** creates a new string
or **`int(3.14)`** creates a new int)

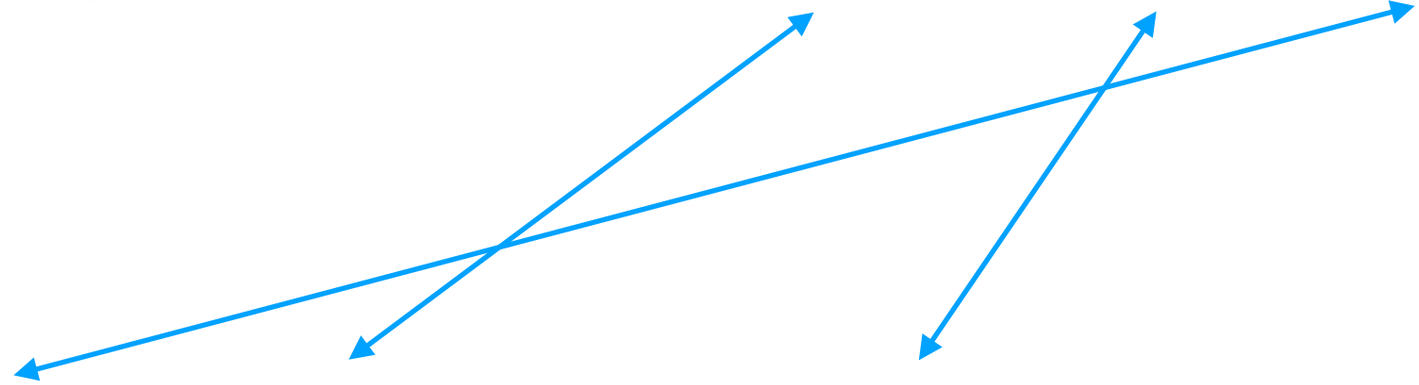
```python
print("Hello " + p.fname + " " + p.lname)
```

```
from collections import namedtuple



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



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



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

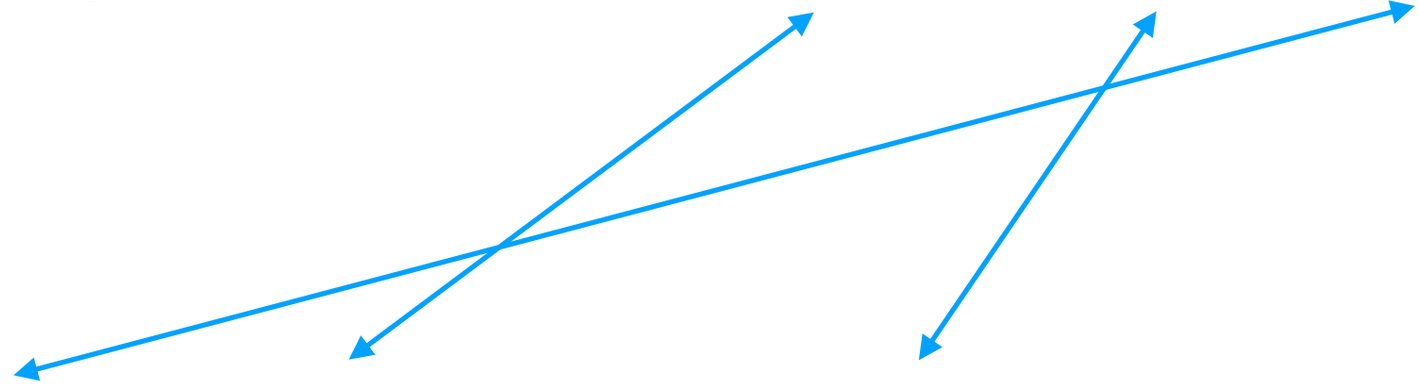
```
from collections import namedtuple



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

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

```
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

```
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

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

References

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

```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!

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

```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!

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


```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)
```

Aside: installing packages

recordclass doesn't come with Python

There are many Python packages available on PyPI

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

Installation example (from terminal):

```
pip install recordclass
```

Today's Outline

New Types

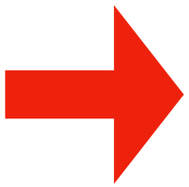
- tuple
- namedtuple
- recordclass

References

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

Mental Model for State (v1)

Code:



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

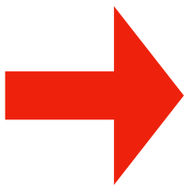
State:

x

y

Mental Model for State (v1)

Code:



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

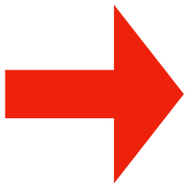
State:

x	hello
y	

Mental Model for State (v1)

Code:

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



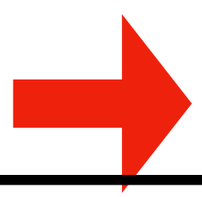
State:

x	hello
y	hello

Mental Model for State (v1)

Code:

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



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

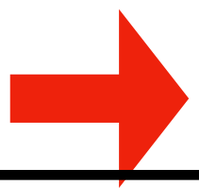
State:

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 *objects*

x

y

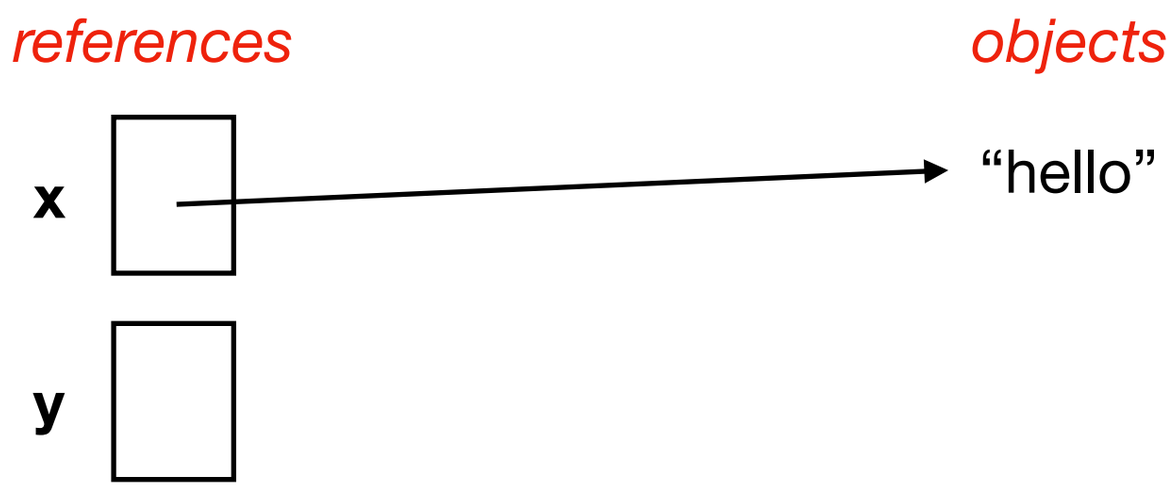
(a variable is one kind of reference)

Mental Model for State (v2)

Code:

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

State:

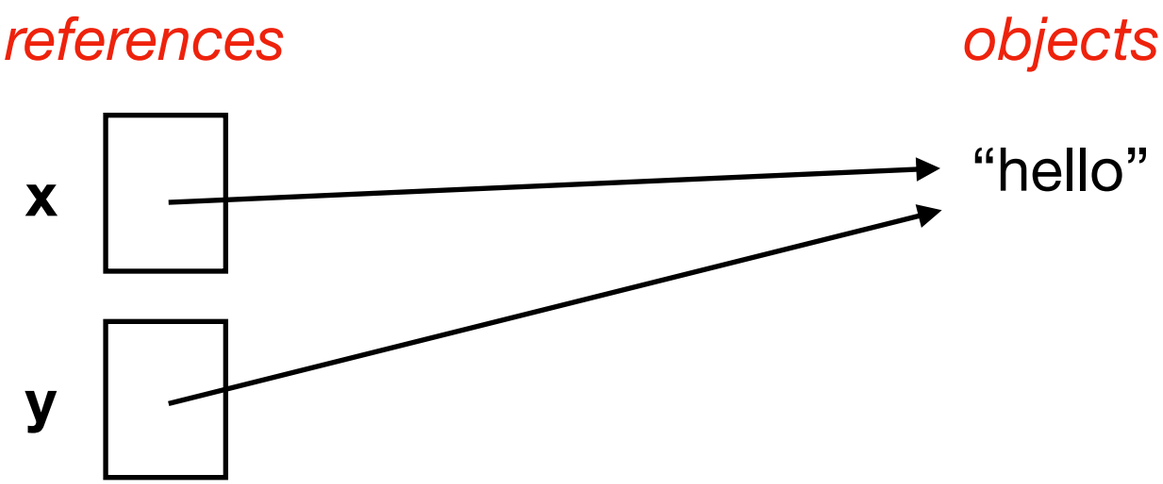


Mental Model for State (v2)

Code:

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

State:

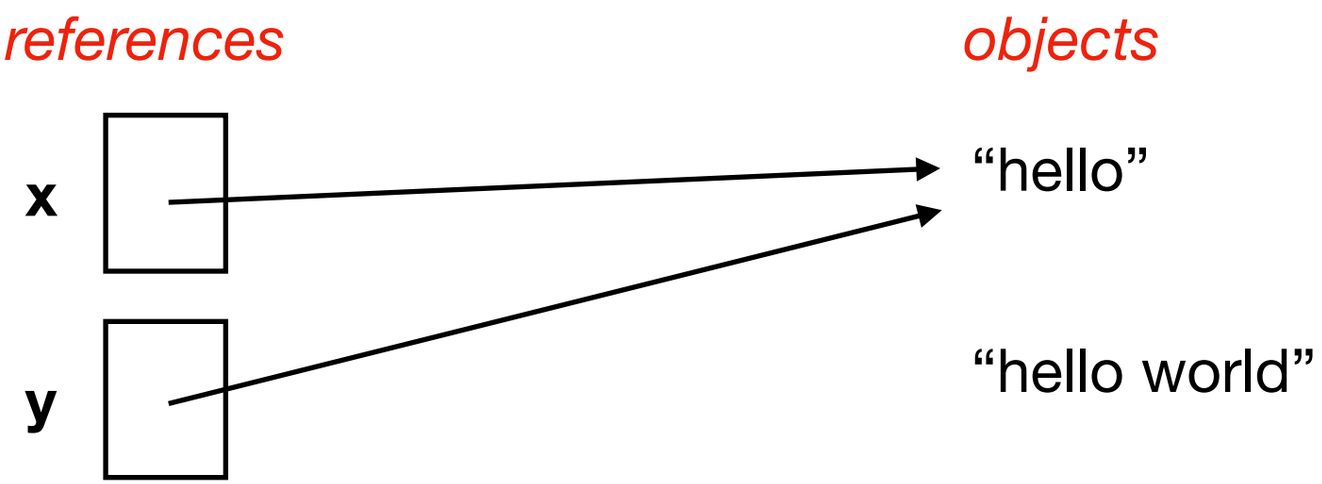


Mental Model for State (v2)

Code:

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

State:

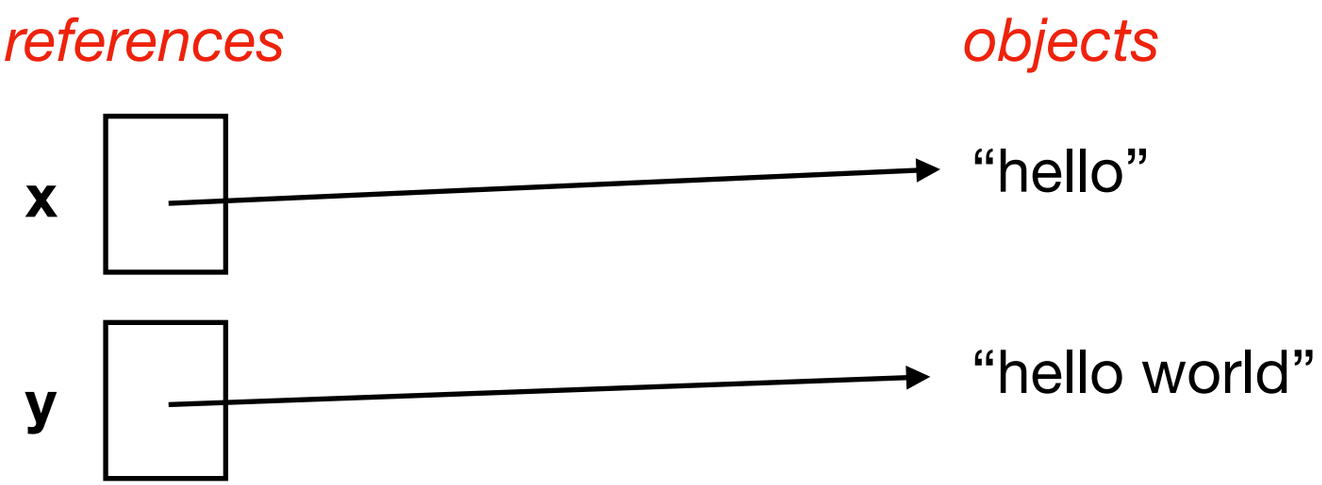


Mental Model for State (v2)

Code:

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

State:

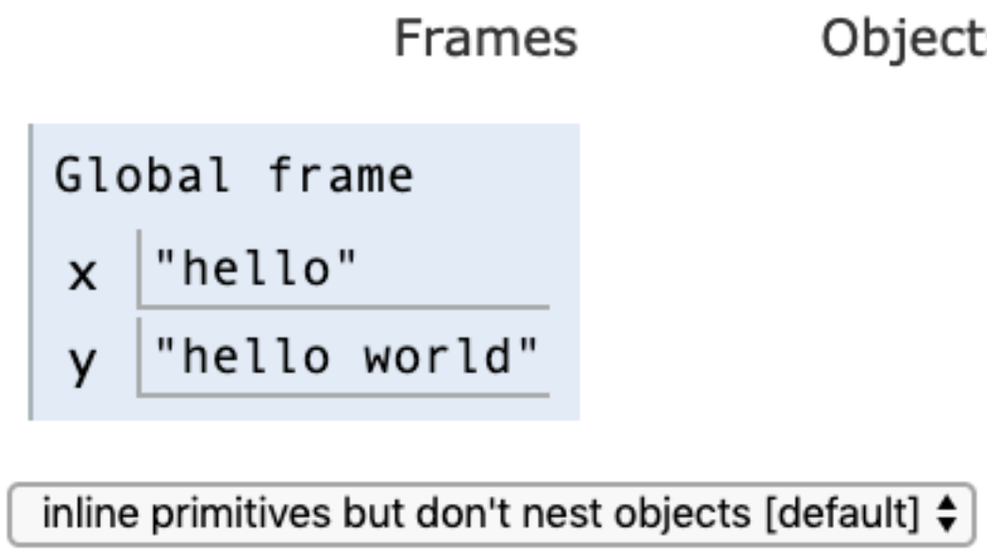


Python Tutor Supports Both

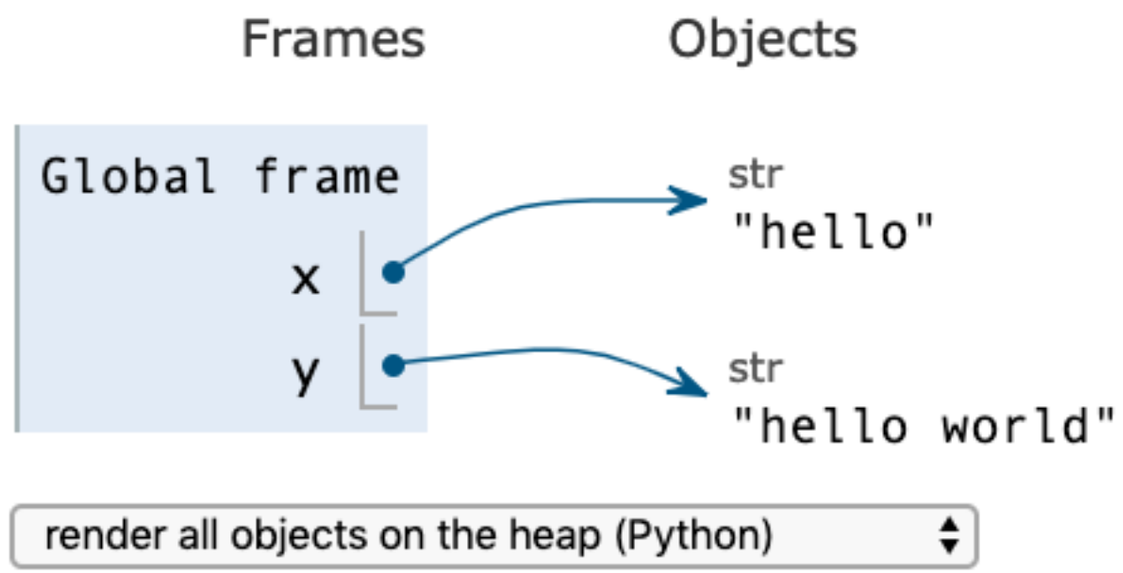
Code:

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

v1



v2



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**?

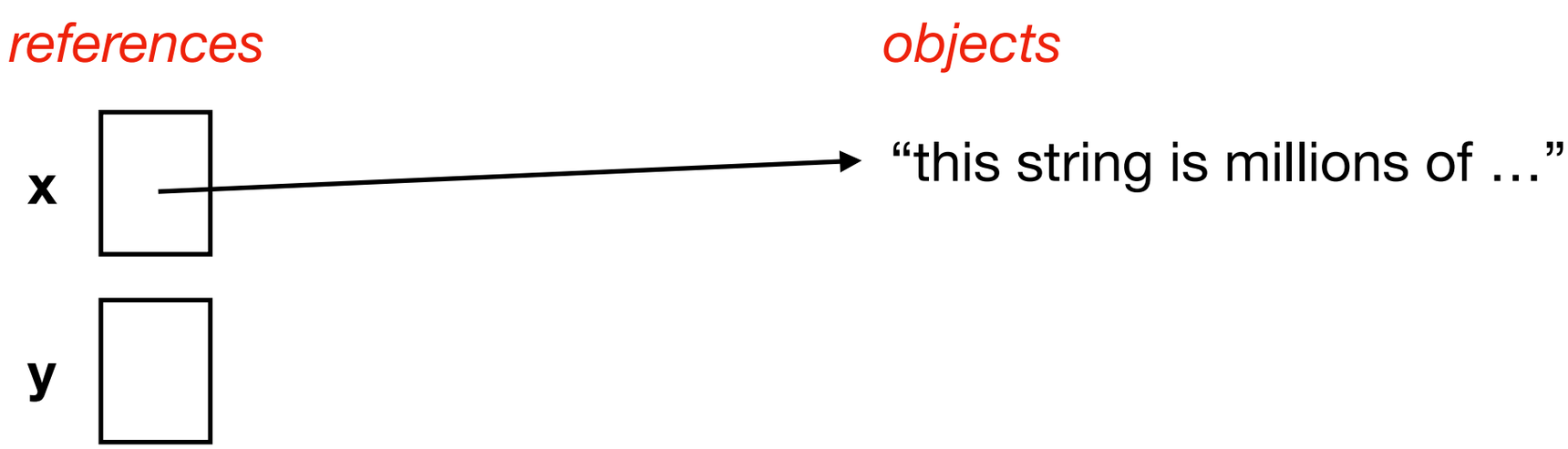
Why not follow the original strategy we learned (i.e., boxes of data with labels)?

Reason 1: Performance

Code:

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

State:

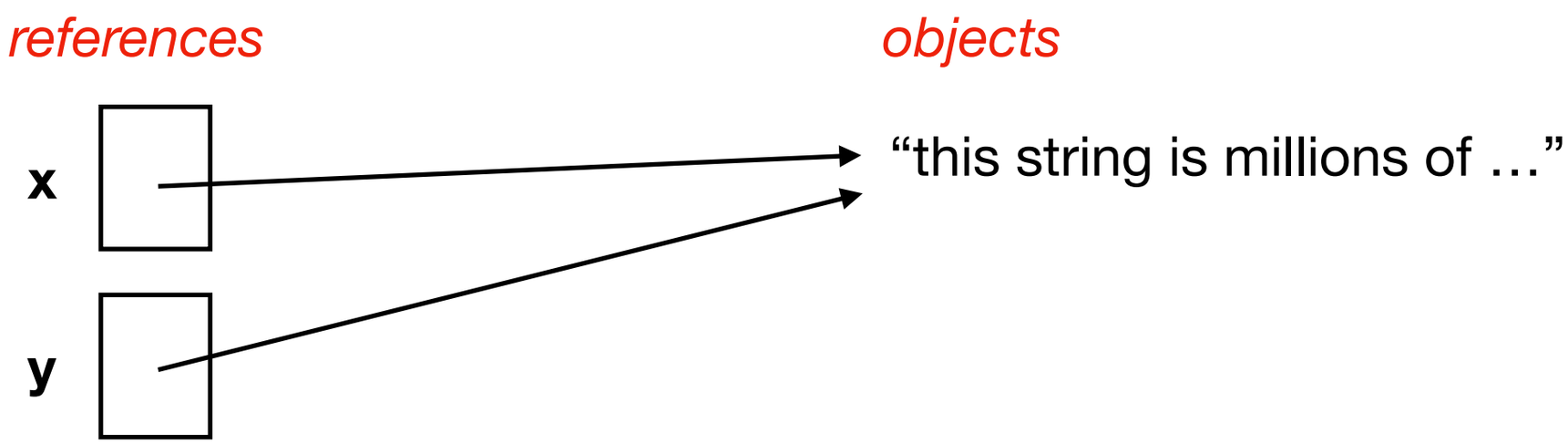


Reason 1: Performance

Code:

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

State:



Reason 2: Centralized Updates

```python
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 *objects*

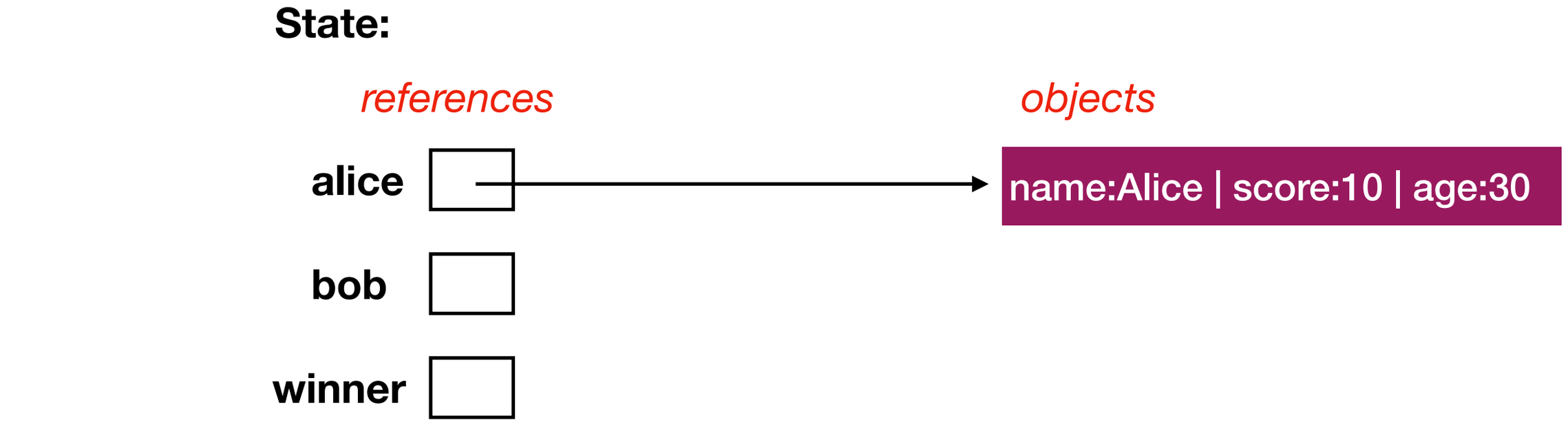
alice

bob

winner

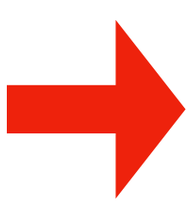
Reason 2: Centralized Updates

```
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)
```

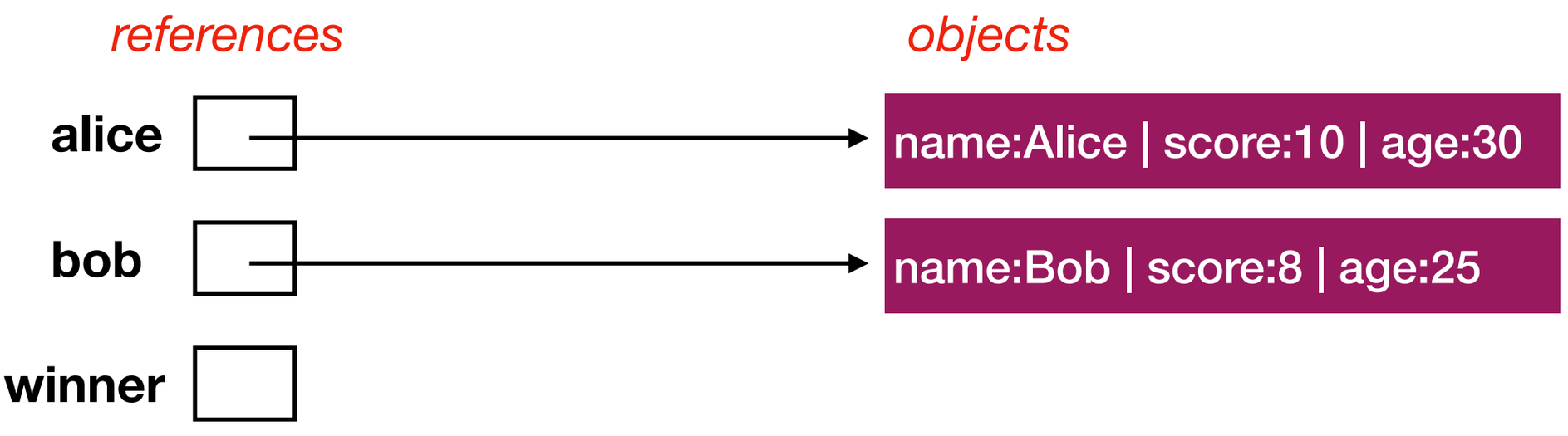


Reason 2: Centralized Updates

```
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:

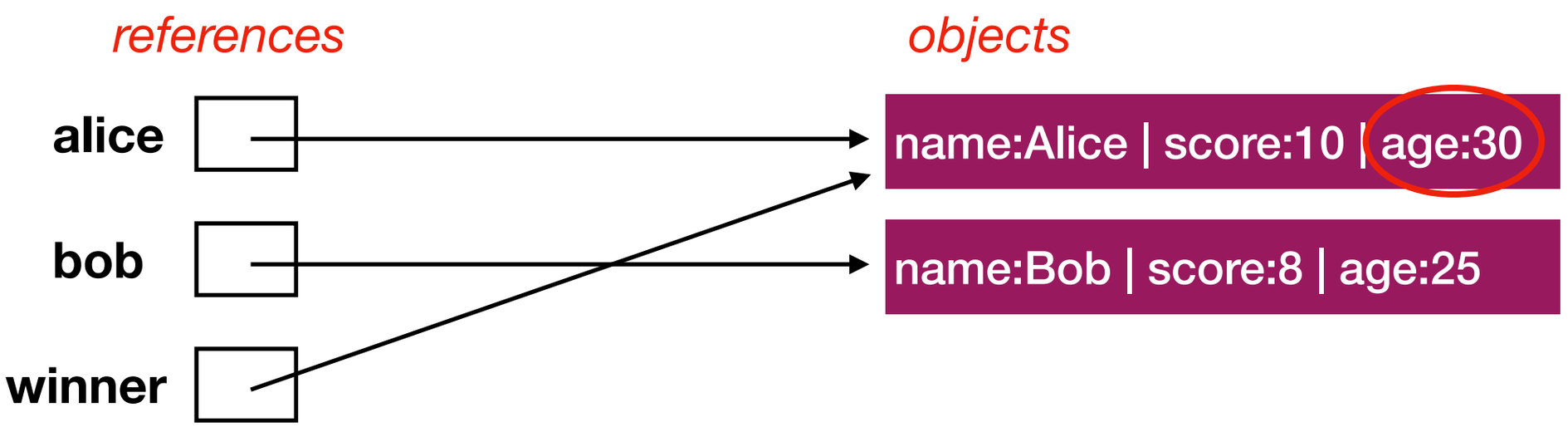


Reason 2: Centralized Updates

```
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
```

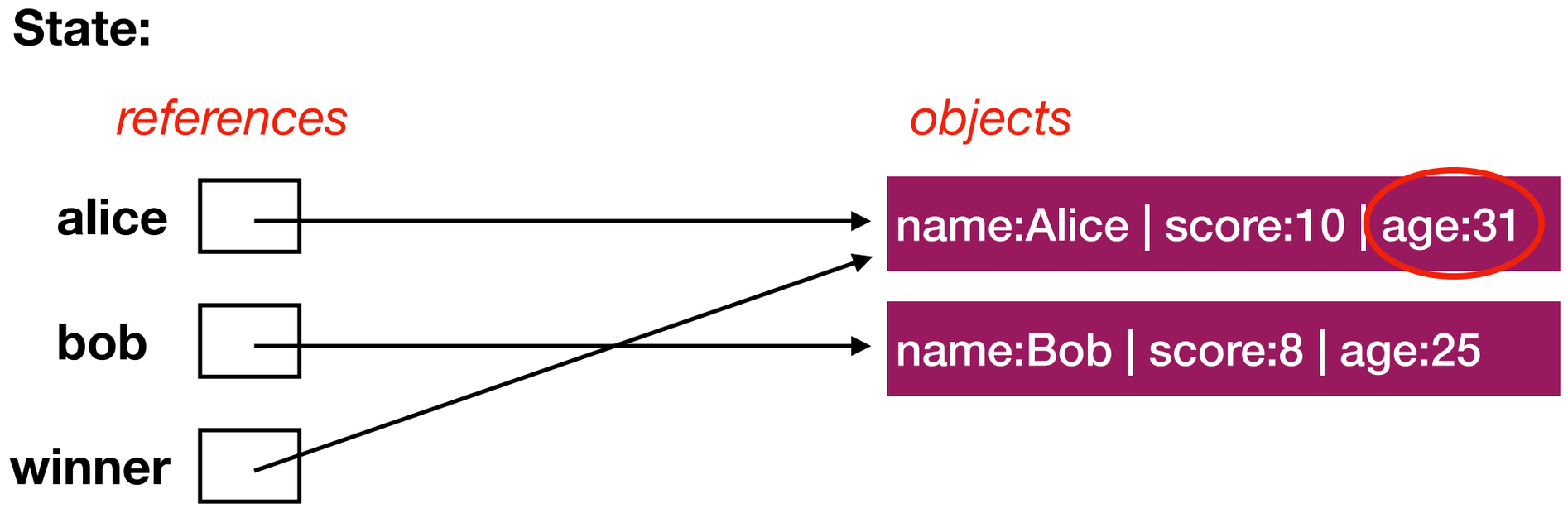
```
alice.age = 31
print("Winner age:", winner.age)
```

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)
```

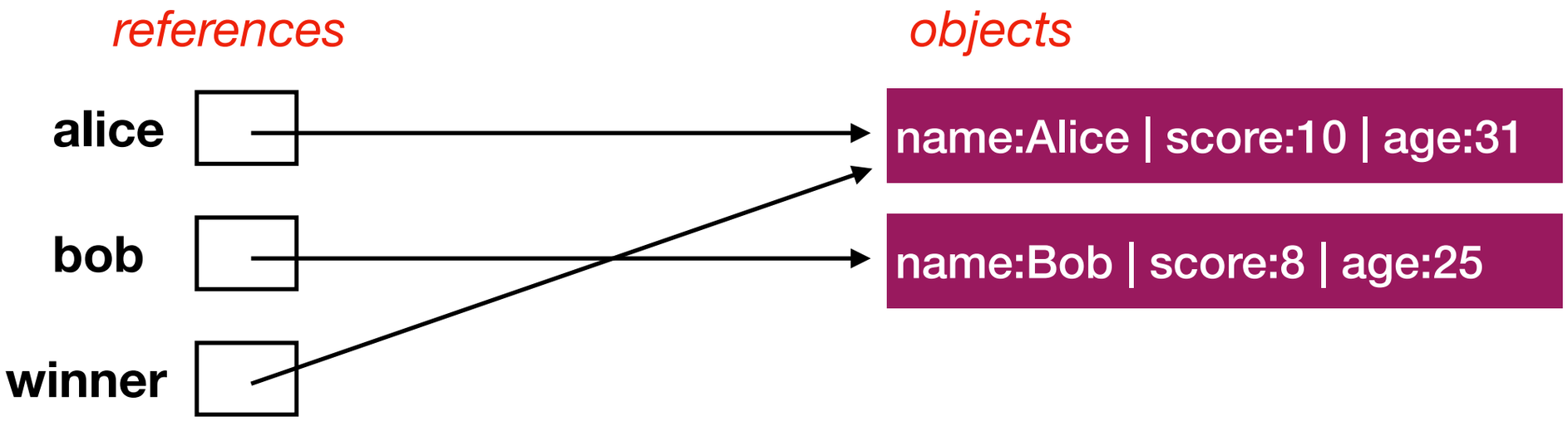


Reason 2: Centralized Updates

```
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. “==”

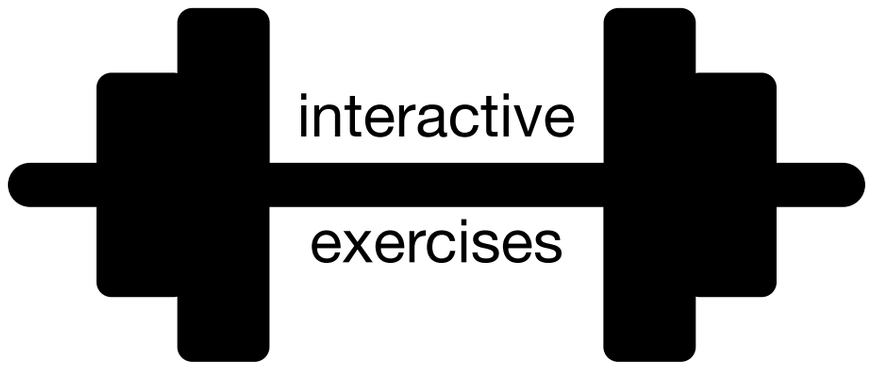
References and Arguments/Parameters

Python Tutor

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

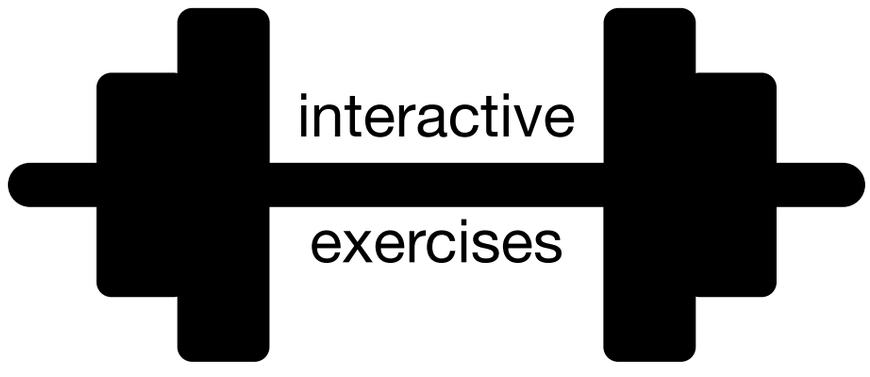
Example 1: reassign parameter

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

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



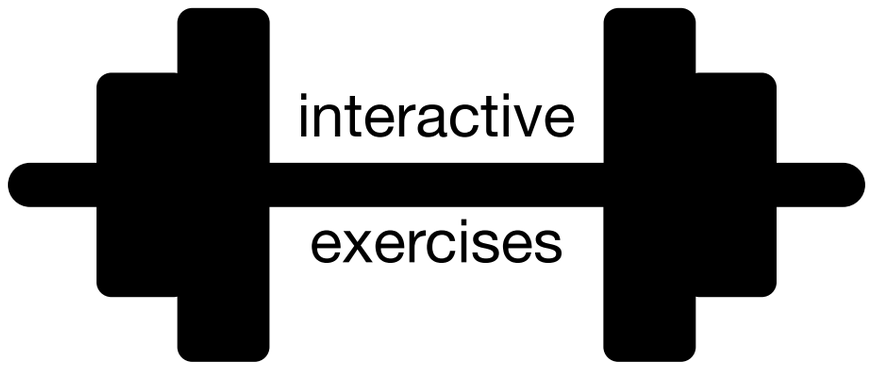
Example 2: modify list via param

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

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



Example 3: reassign new list to param

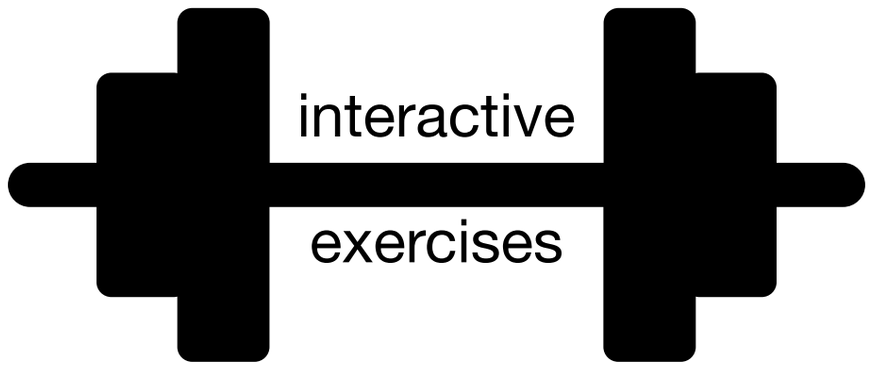
```
def f(items):
    items = items + ["!!!"]
    print("f:", items)

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



Example 4: in-place sort

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

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

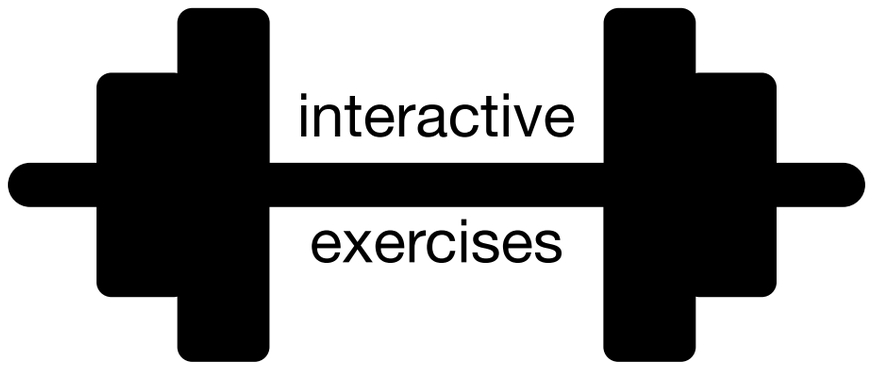
```
numbers= [4,5,3,2,1]
print("first:", first(numbers))
print("smallest:", smallest(numbers))
print("first:", first(numbers))
```



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))
```



Today's Outline

New Types

- tuple
- namedtuple
- recordclass

References

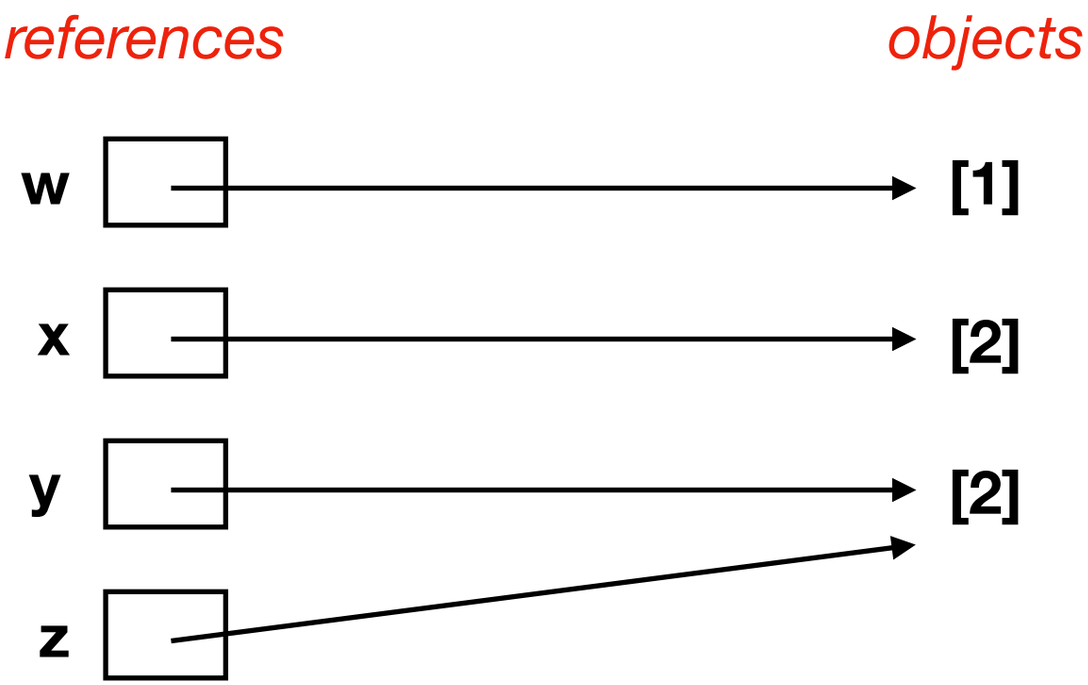
- motivation
- unintentional argument modification
- **“is” vs. “==”**

== 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:

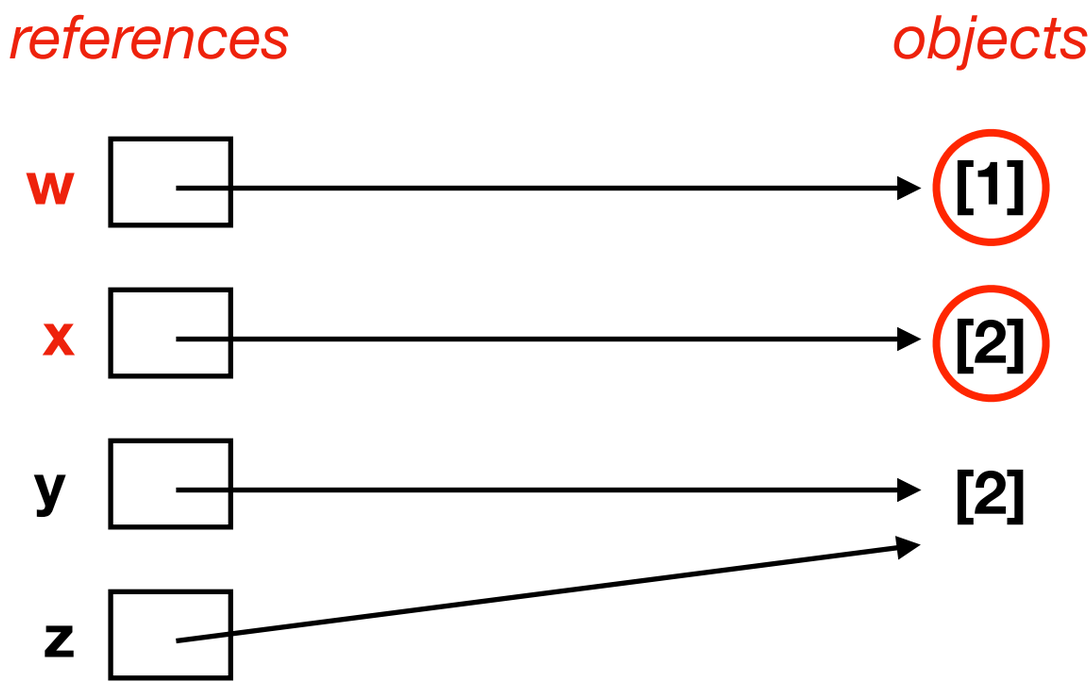


== and is

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

w == x

State:



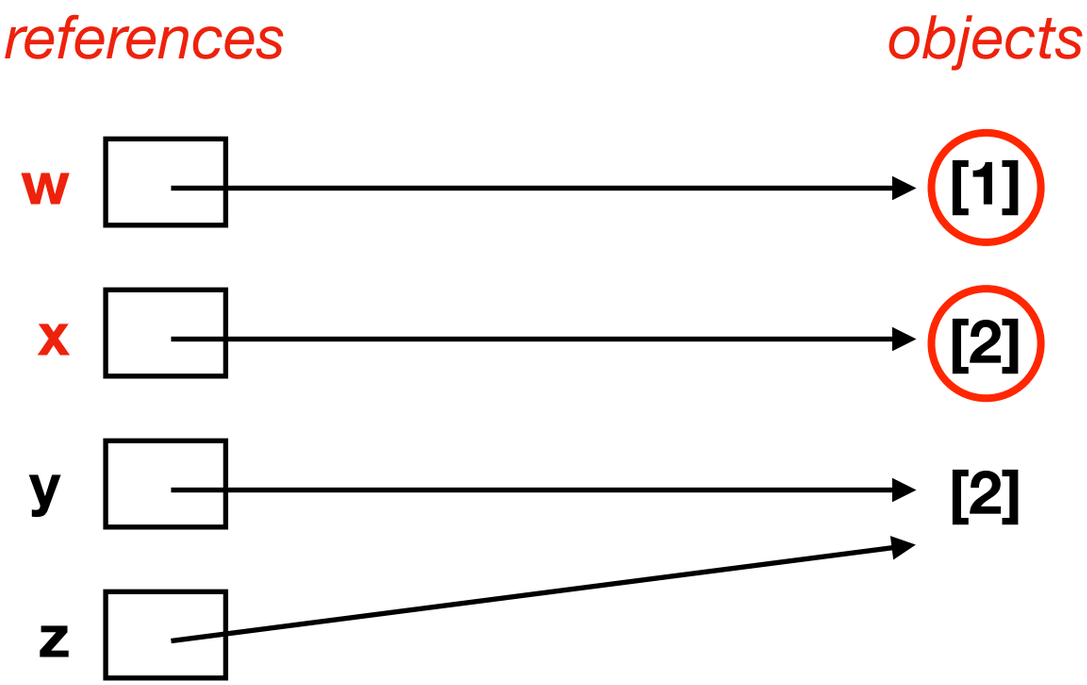
== and is

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

w == x

False

State:

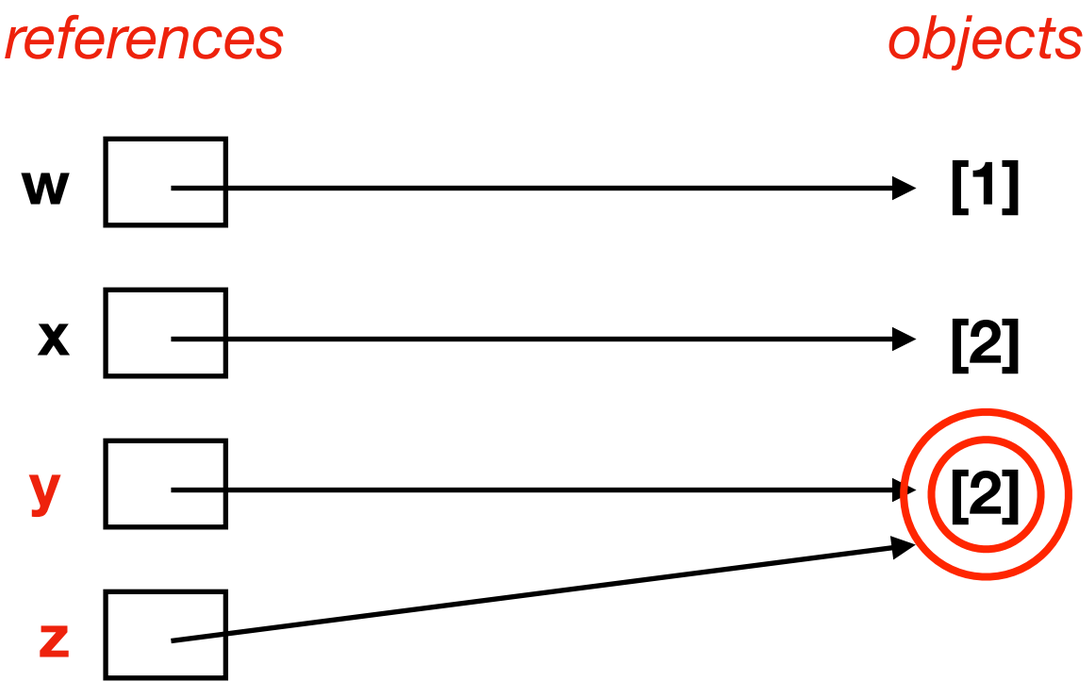


== and is

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

y == z

State:



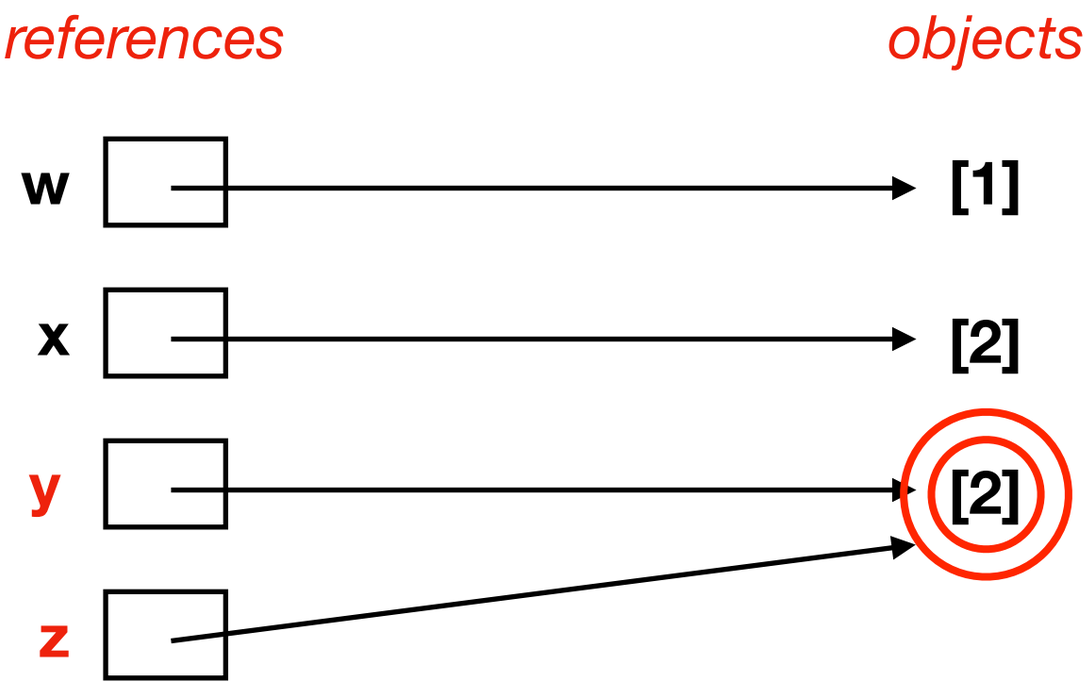
== and is

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

y == z

True

State:

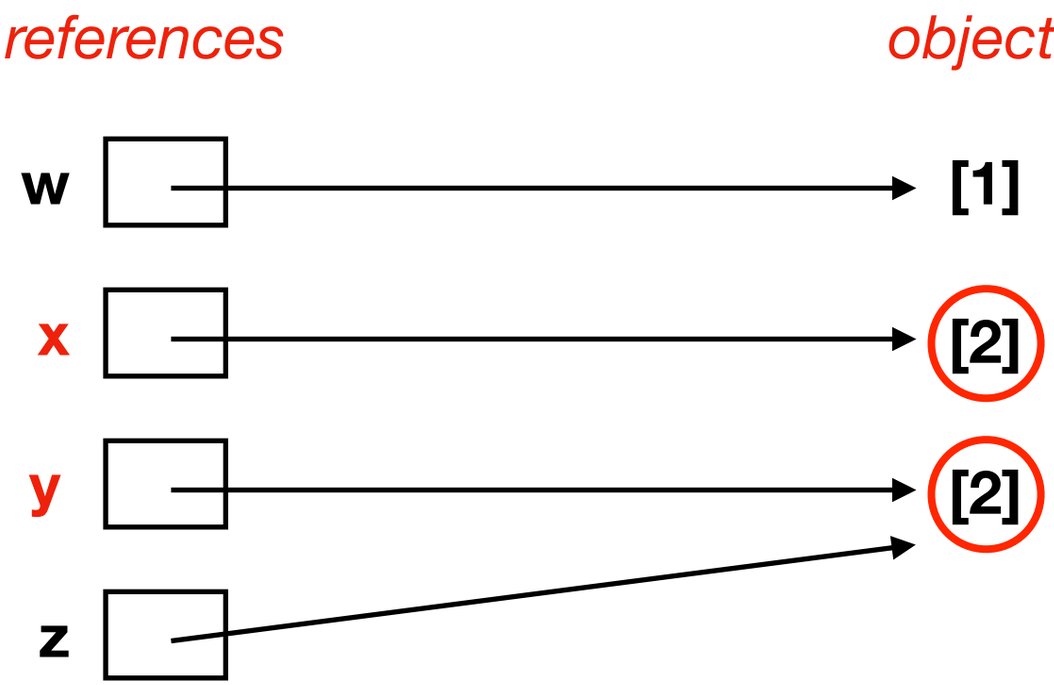


== and is

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

x == y

State:



== and is

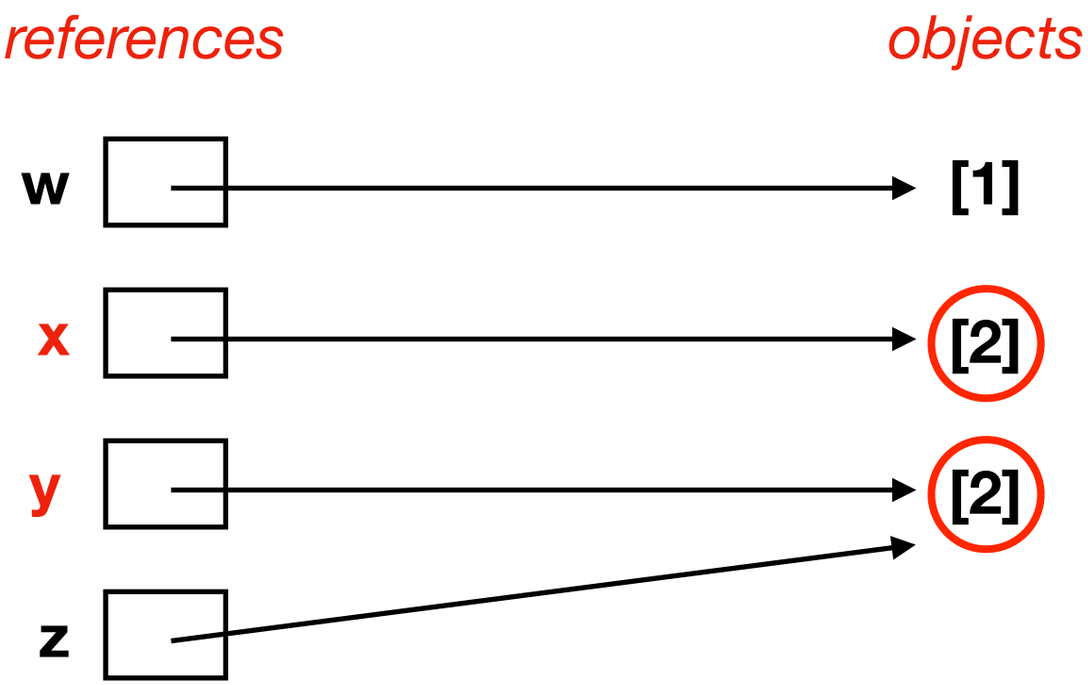
```
w = [1]
x = [2]
y = [2]
z = y
```

x == y

True

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

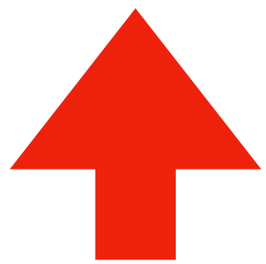
State:



== and is

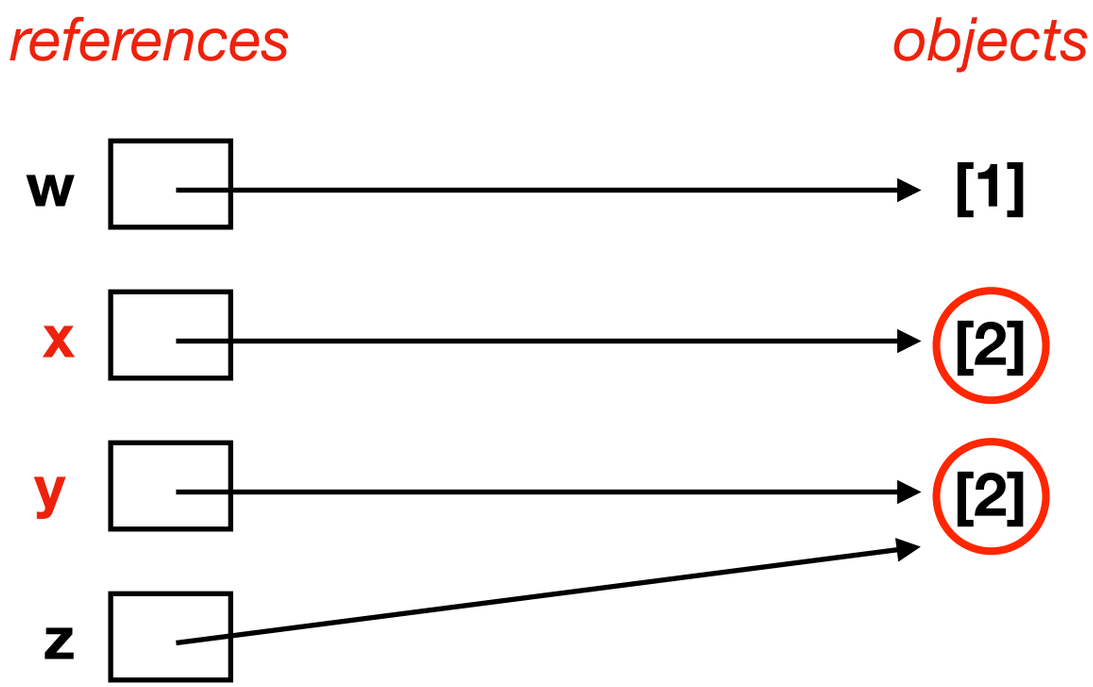
```
w = [1]
x = [2]
y = [2]
z = y
```

x is y



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

State:



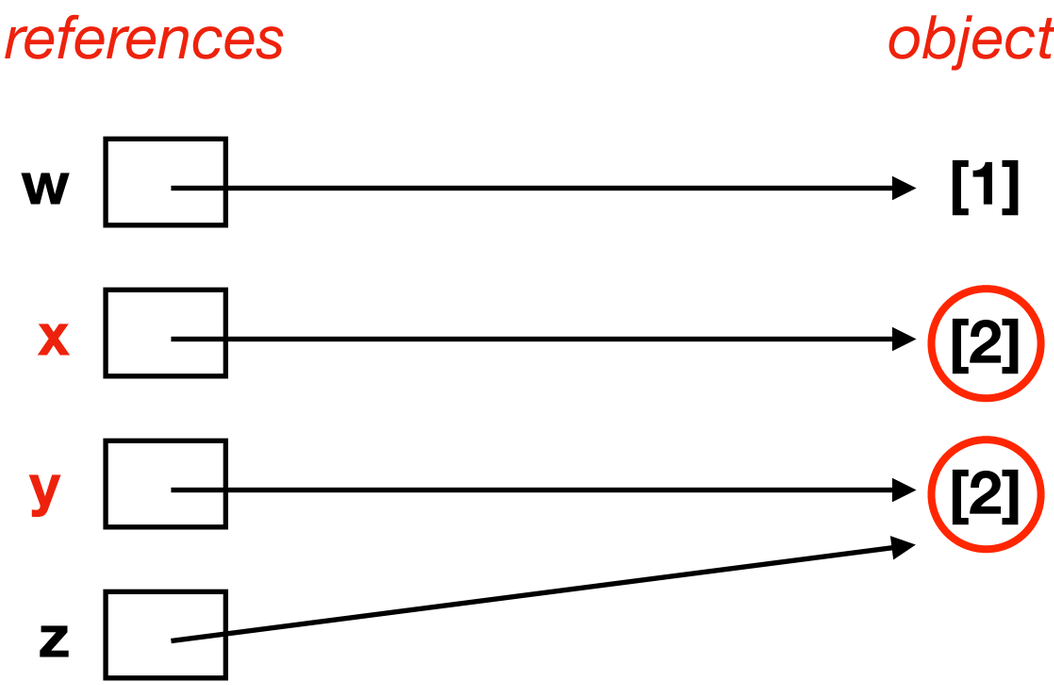
== and is

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

x is y

False

State:

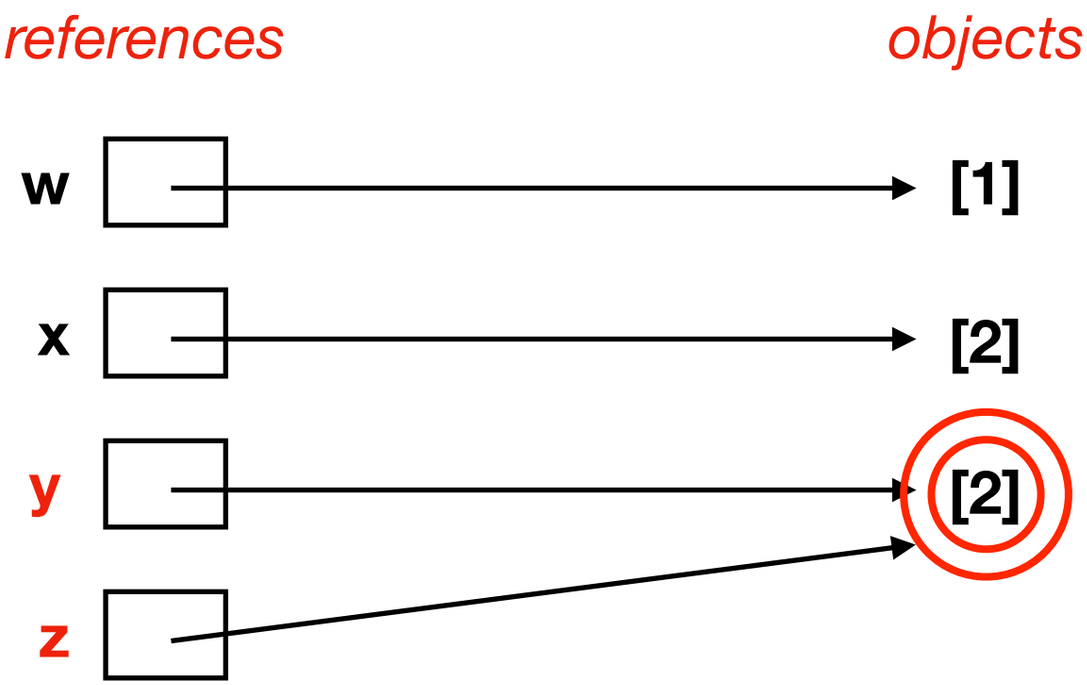


== and is

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

y is z

State:



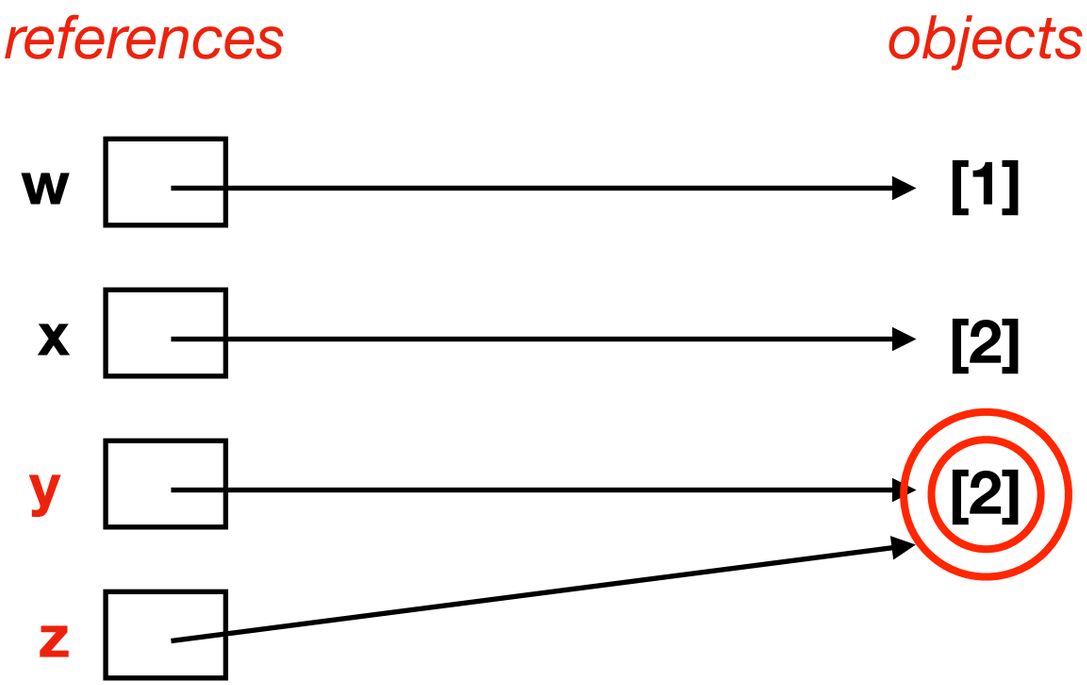
== and is

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

y is z

True

State:



== and is

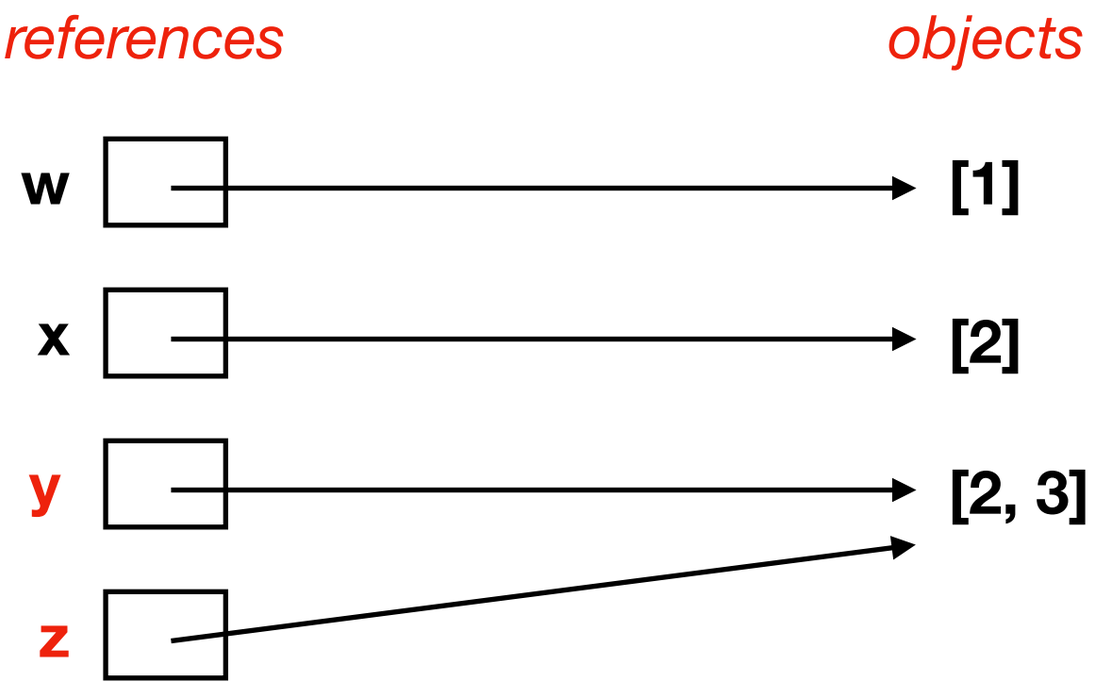
```
w = [1]
x = [2]
y = [2]
z = y
y.append(3)
```

y is z

True

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

State:



== and is

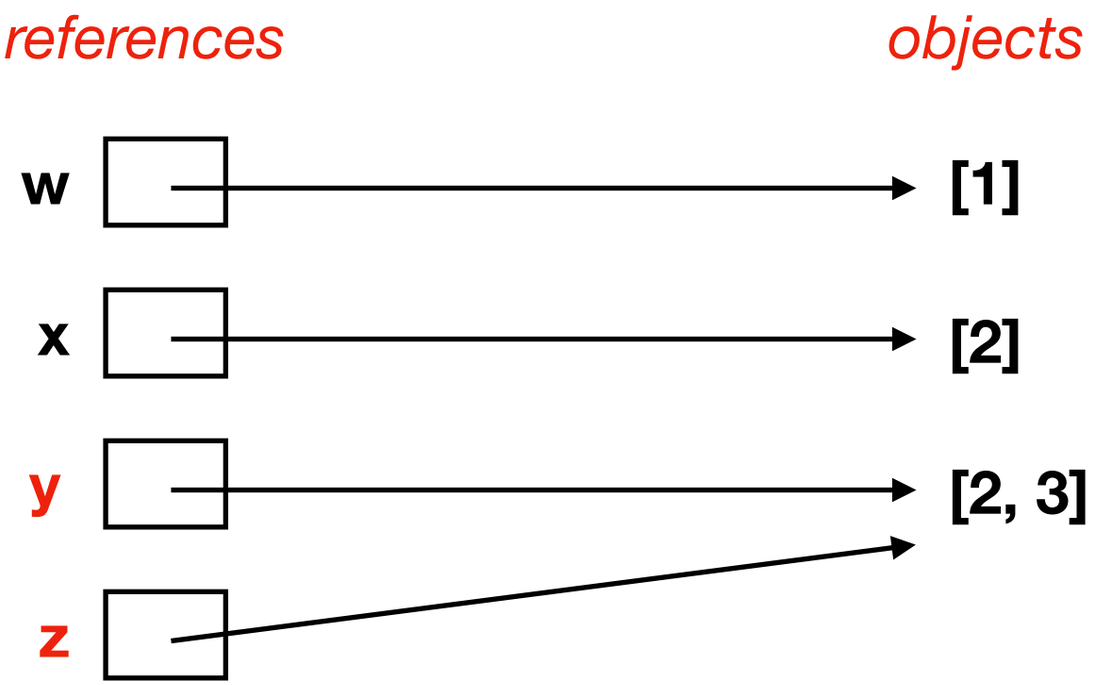
```
w = [1]
x = [2]
y = [2]
z = y
y.append(3)
print(z) # [2,3]
```

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?