

# Tuples Objects and References

Department of Computer Sciences  
University of Wisconsin-Madison

# 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

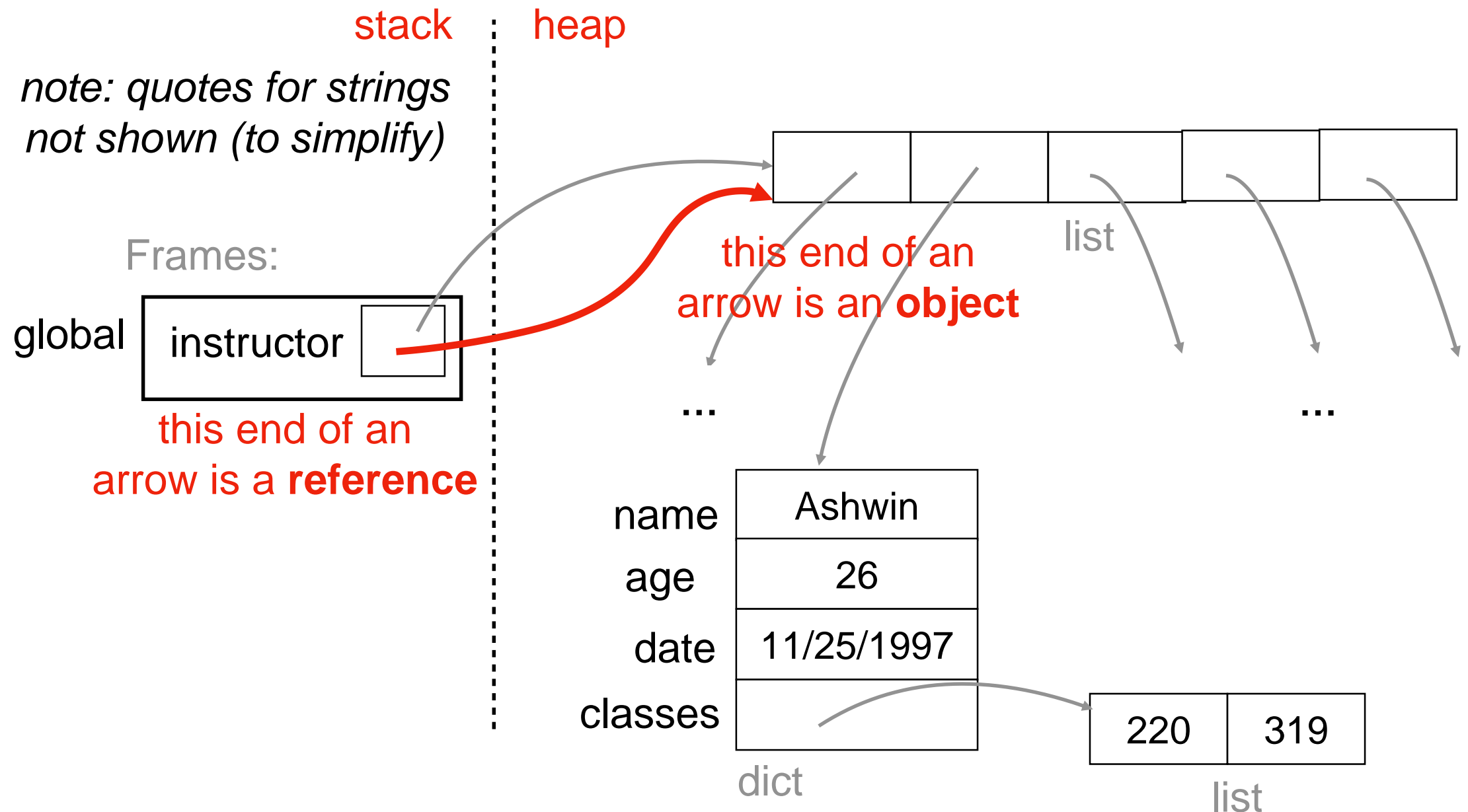
2

list

3

dict

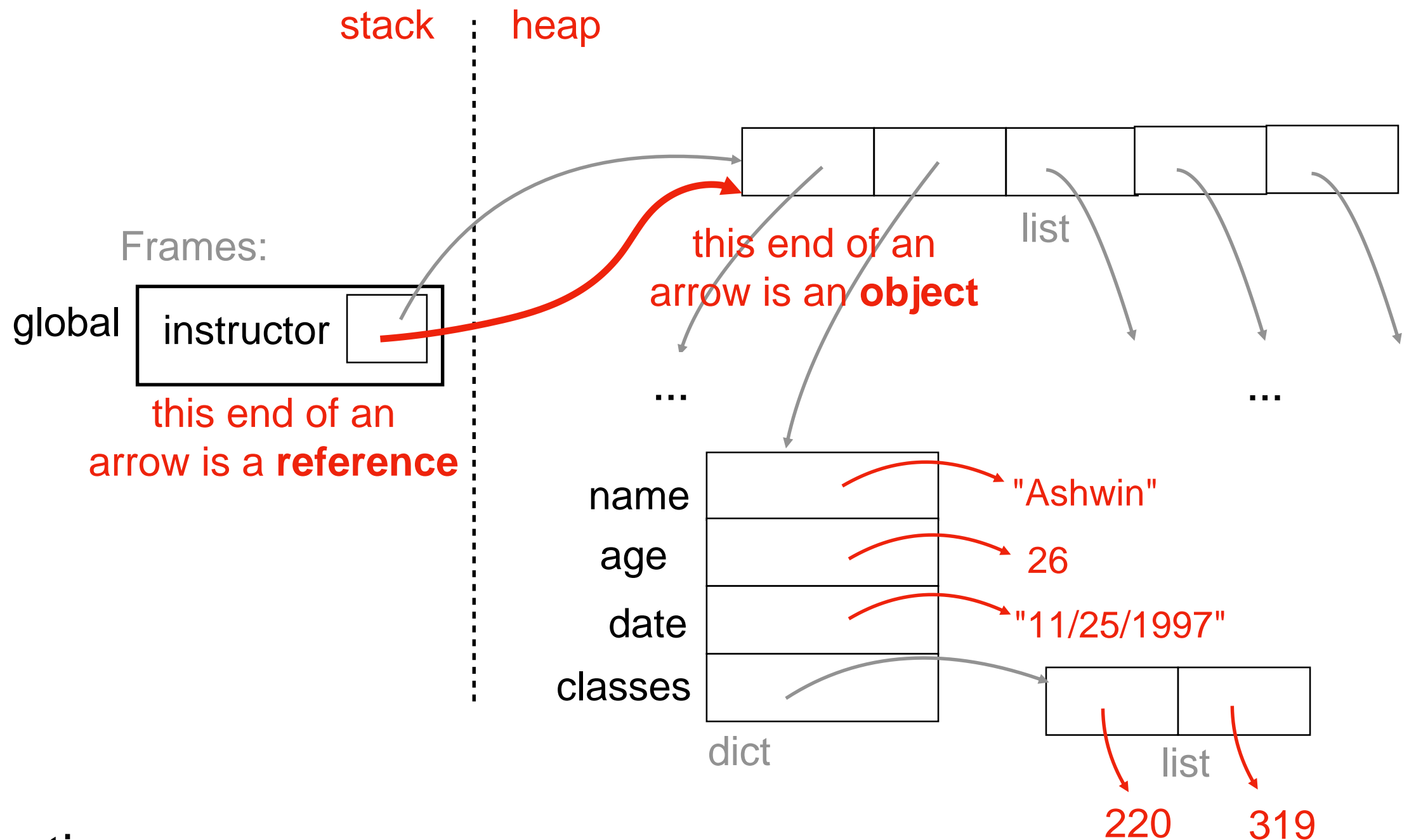
# Objects and References



## 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

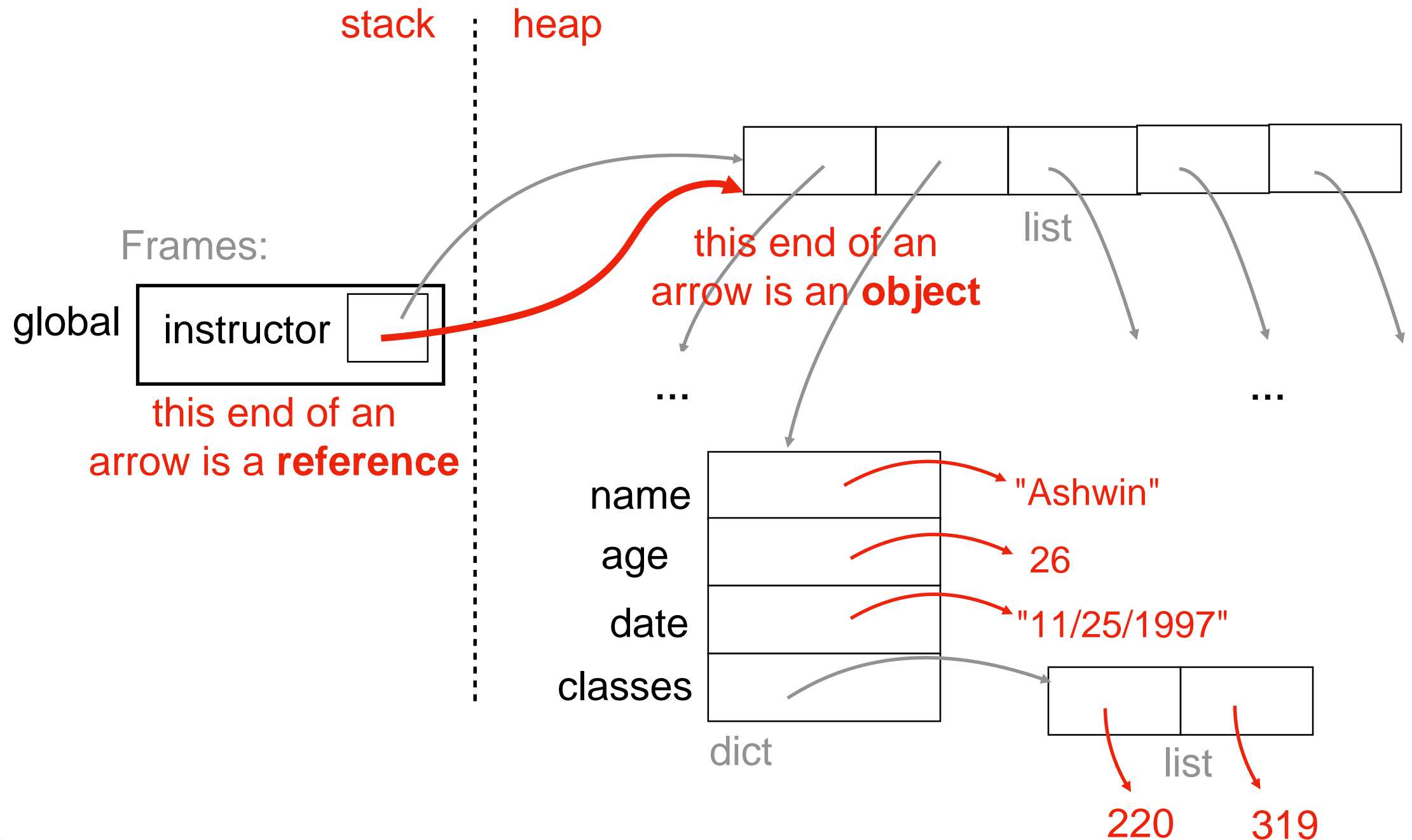
# Objects and References



## 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 copy objects to create new objects?

# Today's Outline



let's evolve our mental  
model of state!

## References

- **Mental Model for State (v2)**
- examples and bugs: accidental argument modification

## New Types of Objects

- tuple
- namedtuple

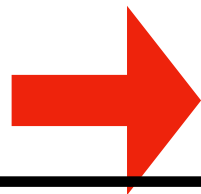
## Motivation for objects and references

- why do we need this new mental model?

# 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:

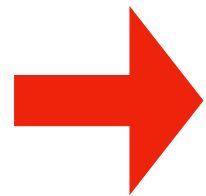
**x** hello

**y** hello world

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

# Mental Model for State (v2)

**Code:**



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

---

**State:**

*references*

*objects*

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



# Mental Model for State (v2)

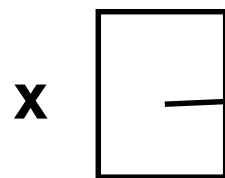
**Code:**

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

---

**State:**

*references*



*objects*

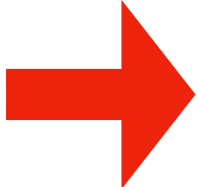
"hello"

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

# Mental Model for State (v2)

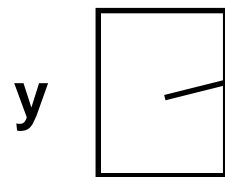
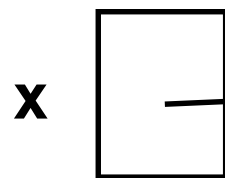
**Code:**

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



**State:**

*references*



*objects*

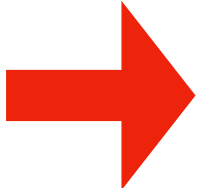
"hello"



# Mental Model for State (v2)

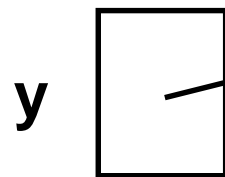
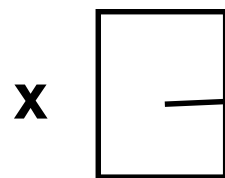
**Code:**

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



**State:**

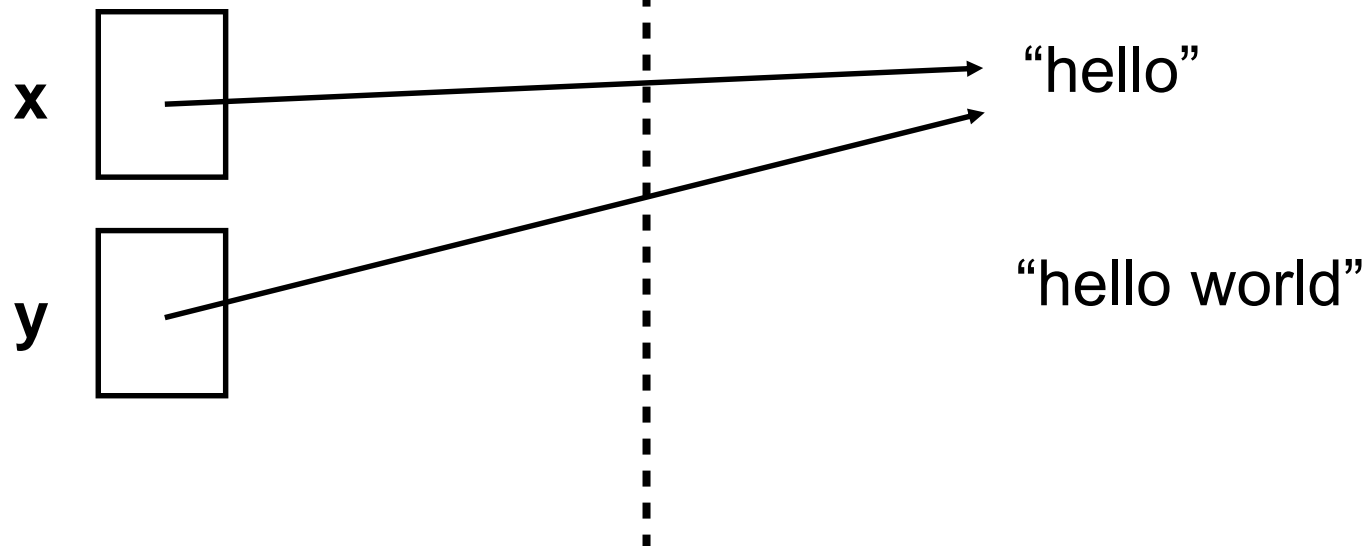
*references*



*objects*

"hello"

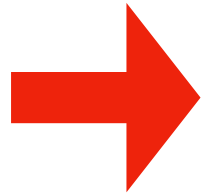
"hello world"



# Mental Model for State (v2)

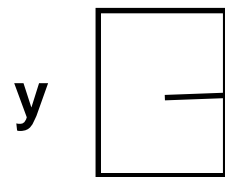
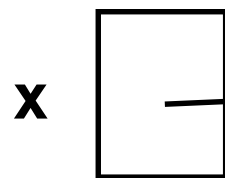
**Code:**

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



**State:**

*references*



*objects*

"hello"

"hello world"



# 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

# Imagine a hidden y = x statement

# How PythonTutor renders immutable types is configurable...

## Code:

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

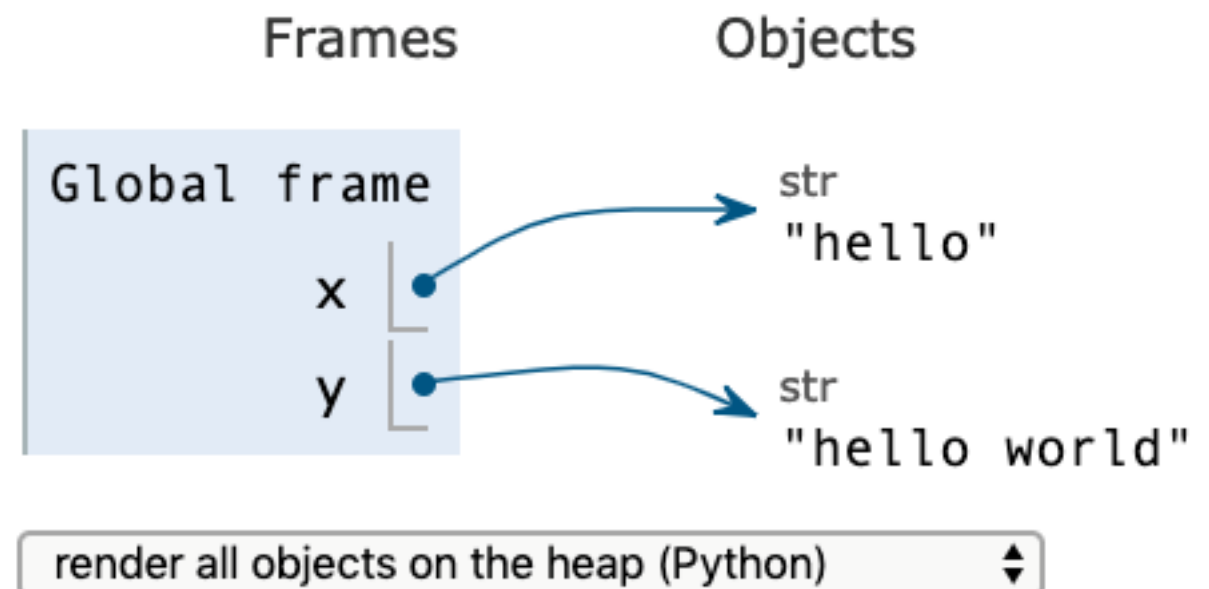
v1



Objects

inline primitives but don't nest objects [default] ⬆⬇⬆

v2



# Today's Outline

## References

- Mental Model for State (v2)
- examples and bugs: accidental argument modification

## New Types of Objects

- tuple
- namedtuple

## Motivation for objects and references

- why do we need this new mental model?

# References and Arguments/Parameters

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

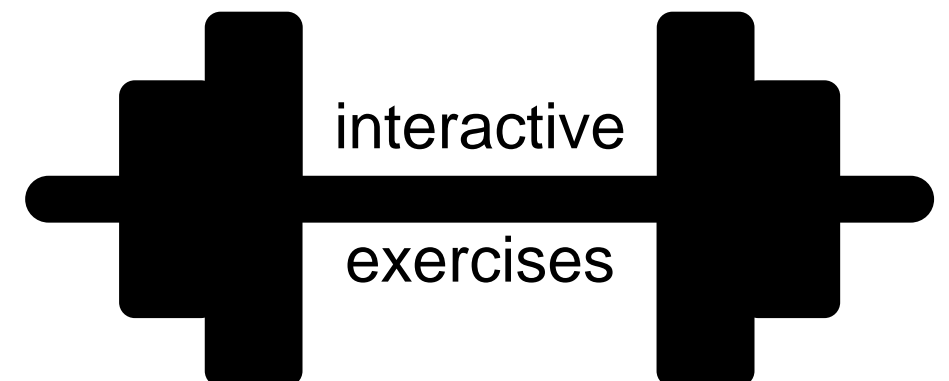
Thinking carefully about a few examples will prevent many debugging headaches...



# Example 1: reassign parameter

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

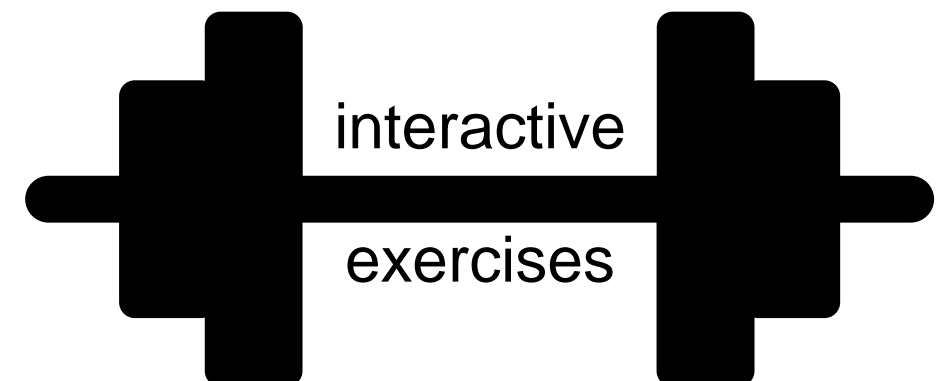
```
num = 10  
f(num)  
print("after:", num)
```



# Example 2: modify list via param

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

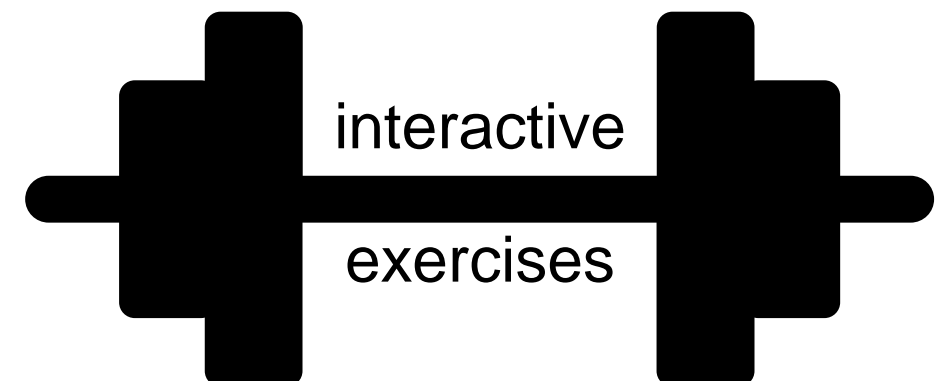
```
words = ['hello', 'world']  
f(words)  
print("after:", words)
```



# Example 3: reassign new list to param

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

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

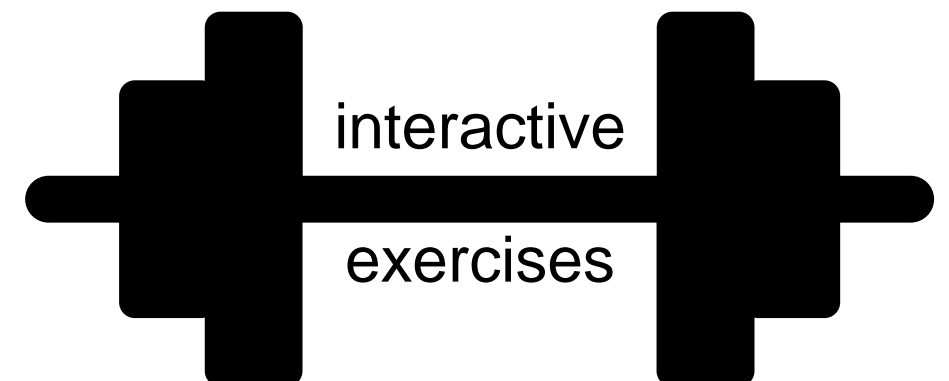


# Example 4: in-place sort

```
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))
```

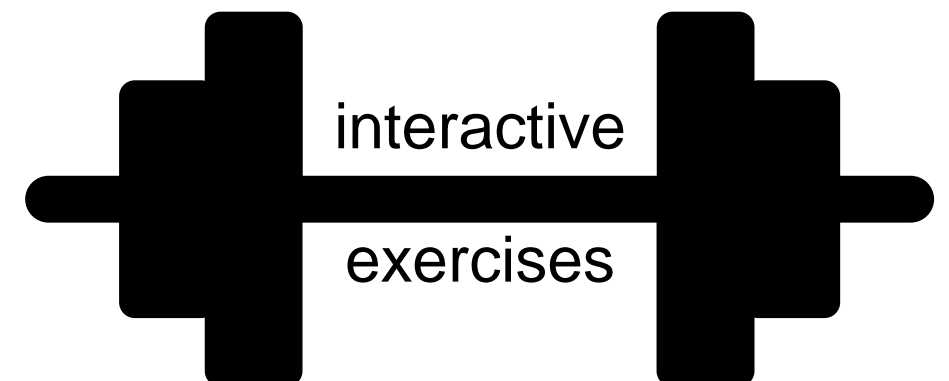


# Example 5: sorted sort

```
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

## References

- Mental Model for State (v2)
- examples and bugs: accidental argument modification

## New Types of Objects

- **tuple**
- namedtuple

## Motivation for objects and references

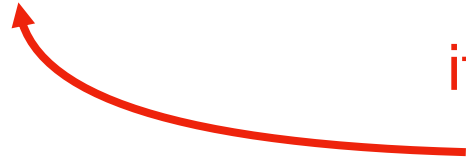
- why do we need this new mental model?

# 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

## References

- Mental Model for State (v2)
- examples and bugs: accidental argument modification

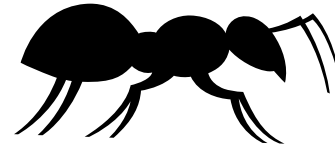
## New Types of Objects

- tuple
- **namedtuple**

## Motivation for objects and references

- why do we need this new mental model?

# See any bugs?



1

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

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

3

```
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
```

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
```

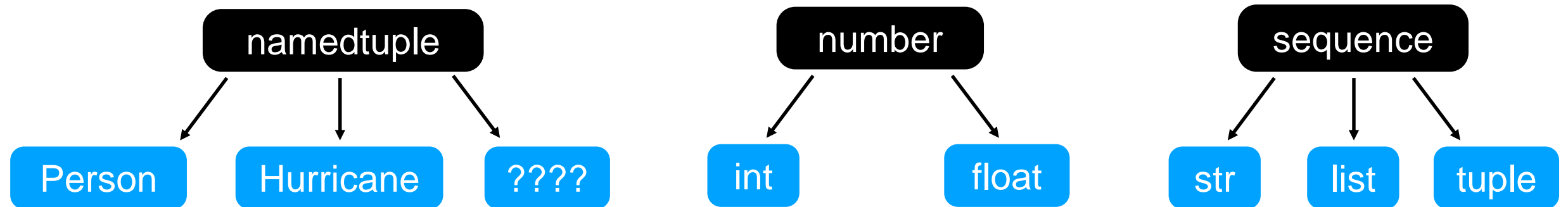
need to import this data struct

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)
```

```
from collections import namedtuple
```

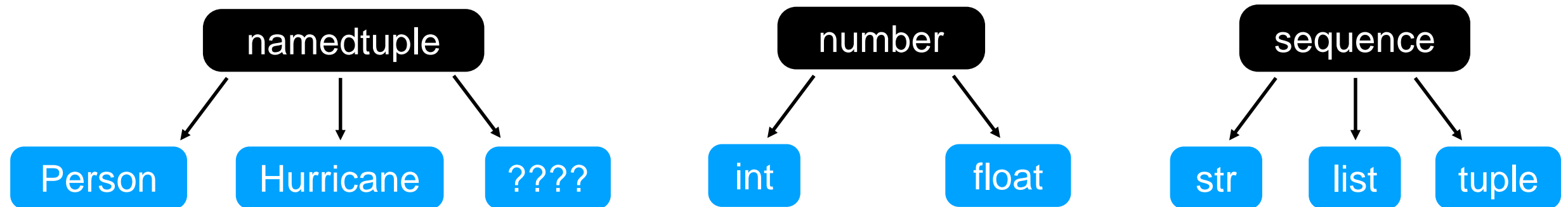
need to import this data struct

name of that type

creates a new type!

name of that type

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



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

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

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

```
from collections import namedtuple
```

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

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



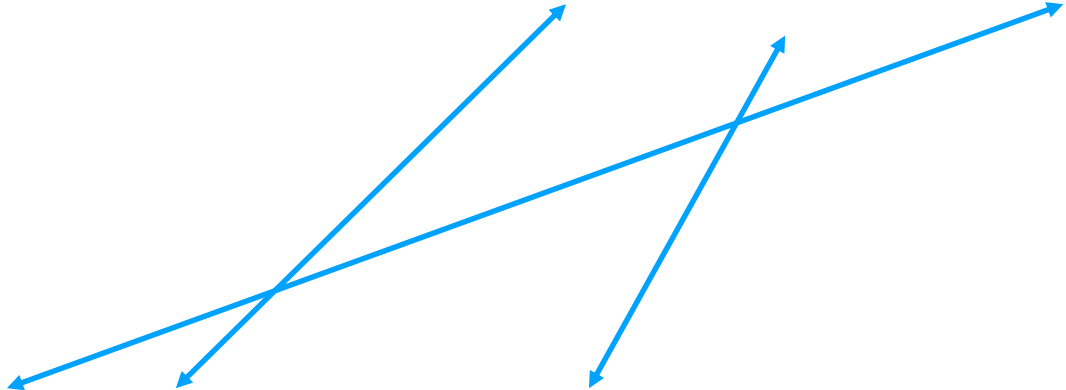
can use either **positional** or keyword arguments to create a Person

```
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")
```

crashes  
immediately  
(good!)

```
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)
```



The diagram consists of two blue arrows. The first arrow originates from the `p.fname` attribute access in the `print` statement and points diagonally upwards to the `fname="Alice"` argument in the `Person` constructor. The second arrow originates from the `p.lname` attribute access in the `print` statement and points diagonally upwards to the `lname="Anderson"` argument in the `Person` constructor.

# Today's Outline

## New Types of Objects

- tuple
- namedtuple

## References

- motivation
- bugs: accidental argument modification



# Today's Outline

## References

- Mental Model for State (v2)
- examples and bugs: accidental argument modification

## New Types of Objects

- tuple
- namedtuple

## Motivation for objects and references


- why do we need this new mental model?

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:**

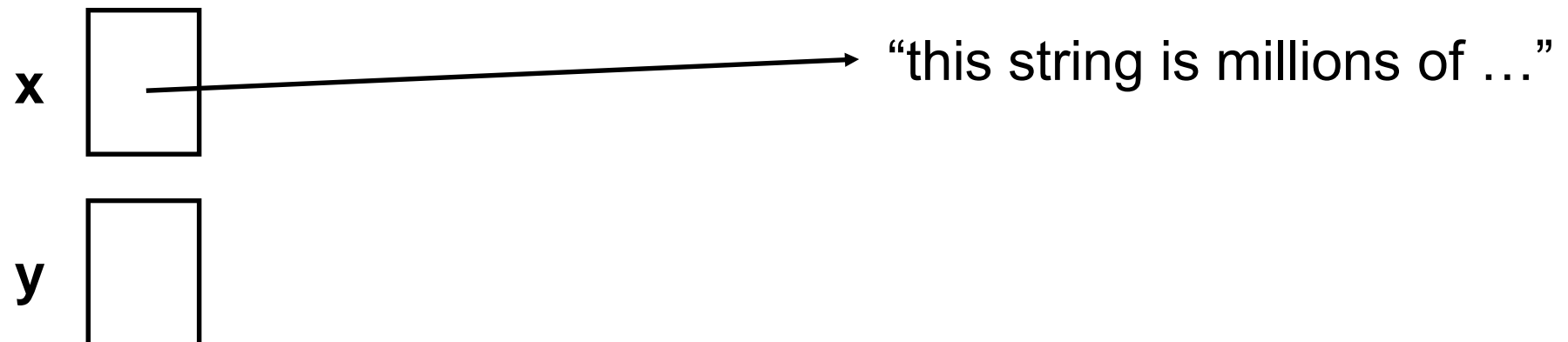
 `x = "this string is millions of characters..."`  
`y = x # this is fast!`

---

**State:**

*references*

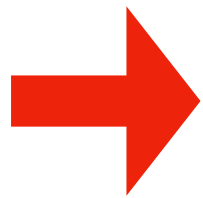
*objects*



# Reason 1: Performance

## Code:

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

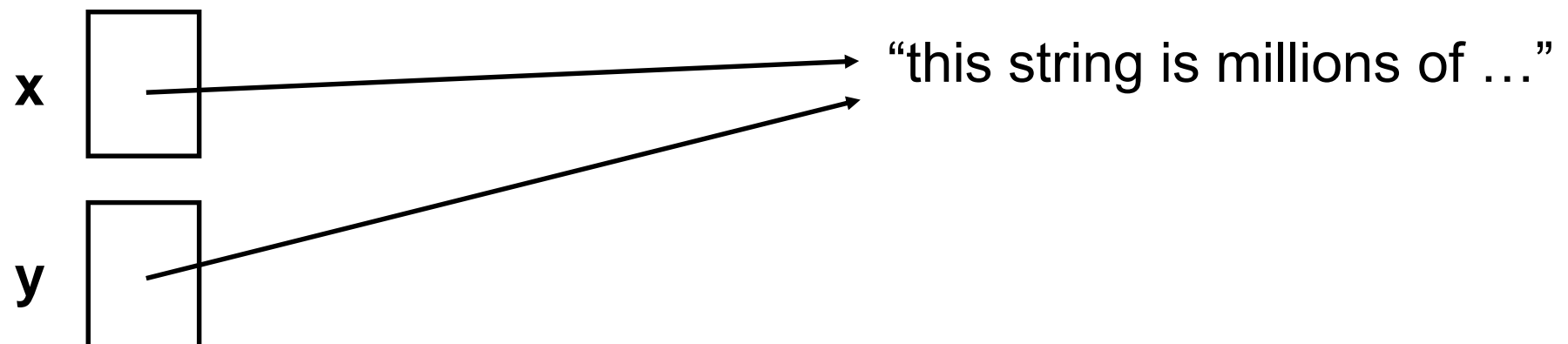


---

## State:

*references*

*objects*



# Reason 2: Centralized Updates

alice = {"name": "Alice", "score": 10, "age": 30}

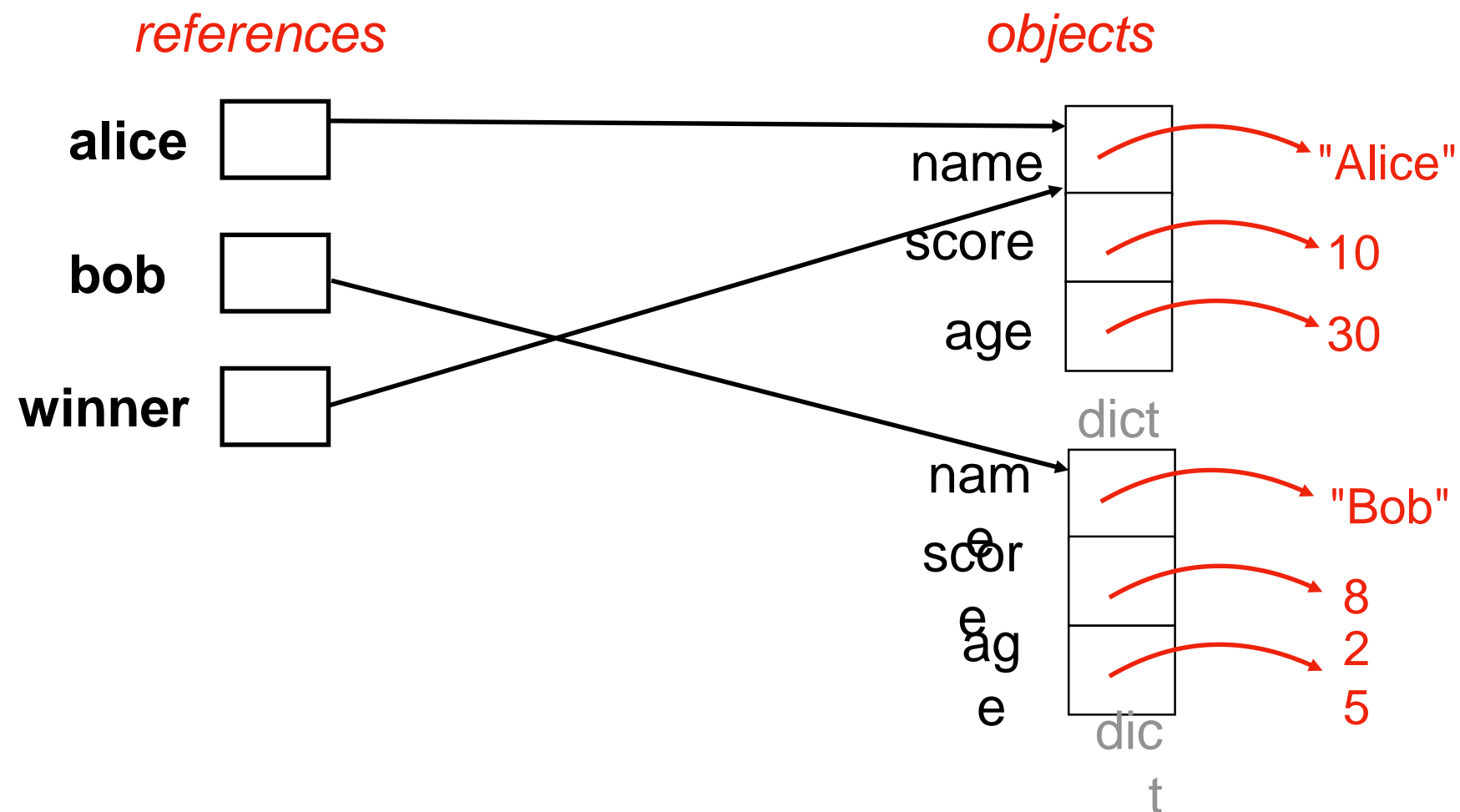
bob = {"name": "Bob", "score": 8, "age": 25}

winner = alice

alice["age"] += 1

print("Winner age:", winner["age"])

State:

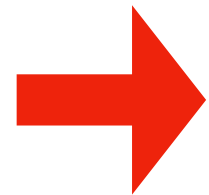


# Reason 2: Centralized Updates

```
alice = {"name": "Alice", "score": 10, "age": 30}
```

```
bob = {"name": "Bob", "score": 8, "age": 25}
```

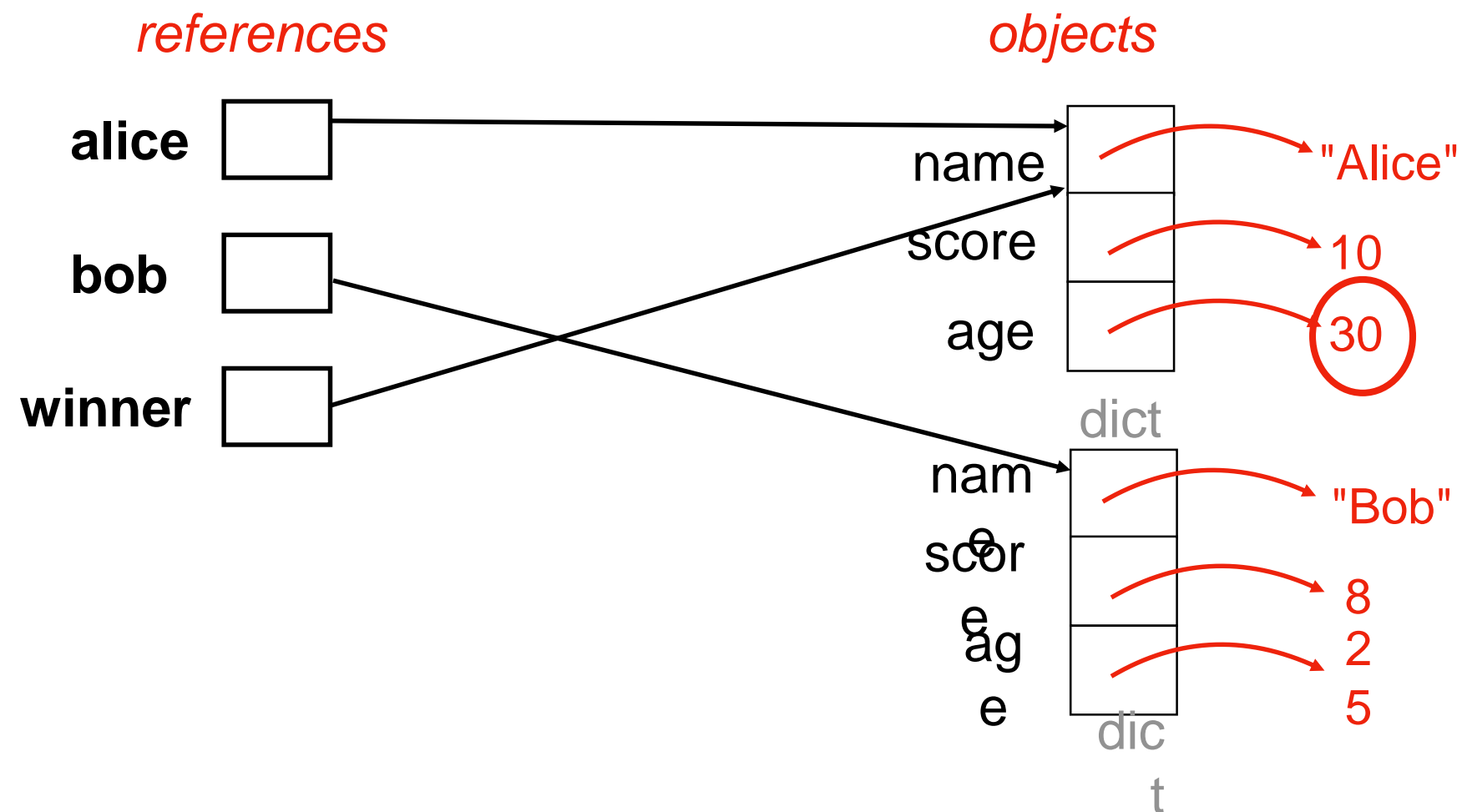
```
winner = alice
```



```
alice["age"] += 1
```

```
print("Winner age:", winner["age"])
```

State:



# Reason 2: Centralized Updates

```
alice = {"name": "Alice", "score": 10, "age": 30}
```

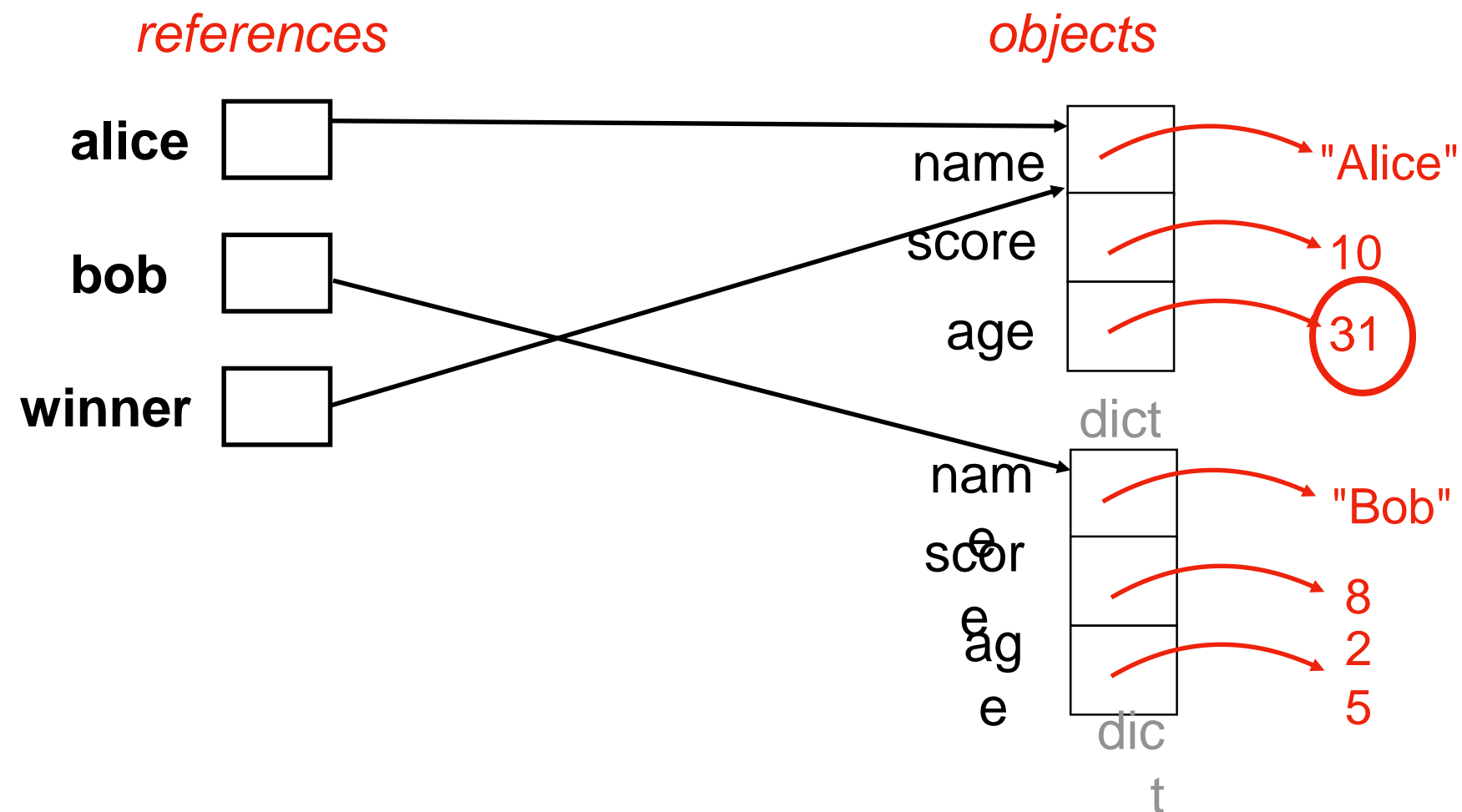
```
bob = {"name": "Bob", "score": 8, "age": 25}
```

```
winner = alice
```

➔ `alice["age"] += 1`  
`print("Winner age:", winner["age"])`

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

State:



# Conclusion

## New Types of Objects

- **tuple**: immutable equivalent as list
- **namedtuple**: make your own immutable types!
  - choose names, don't need to remember positions

## References

- **motivation**: faster and allows centralized update
- **gotchas**: mutating a parameter affects arguments