[220 / 319]

Objects + References

Department of Computer Sciences
University of Wisconsin-Madison

Readings:
Chapter 10 & 12 of Think Python
Chapter 11.1 to 11.7 of Python for Everybody
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

note: quotes for strings not shown (to simplify)
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...
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

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:

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

State:

```
|   |   |
n| x |   |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>
```

*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:

\[
\begin{array}{|c|}
\hline
x & hello \\
\hline
y & \\
\hline
\end{array}
\]
Mental Model for State (v1)

Code:

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

State:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>hello</td>
</tr>
<tr>
<td>y</td>
<td>hello</td>
</tr>
</tbody>
</table>
Mental Model for State (v1)

Code:

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

State:

<table>
<thead>
<tr>
<th>x</th>
<th>hello</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>hello world</td>
</tr>
</tbody>
</table>
Mental Model for State (v1)

Code:

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

State:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>hello</td>
</tr>
<tr>
<td>y</td>
<td>hello world</td>
</tr>
</tbody>
</table>

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

Issues
- incorrect for mutable types
- ignores performance
Mental Model for State (v2)

Code:

```python
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:

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

State:

- **references**: `x` points to "hello"
- **objects**: "hello"

*any box with an arrow is a reference (variables are one kind of reference)*
Mental Model for State (v2)

Code:

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

State:

- **references**
  - `x`
  - `y`

- **objects**
  - "hello"
Mental Model for State (v2)

Code:

\[
\begin{align*}
x &= \text{"hello"} \\
y &= x \\
y &+= \text{" world"}
\end{align*}
\]

State:

- **references**
  - \(x\)
  - \(y\)

- **objects**
  - \(\text{"hello"}\)
  - \(\text{"hello world"}\)
Mental Model for State (v2)

Code:

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

State:

- **references**
  - `x` references "hello"
  - `y` references "hello world"

- **objects**
  - "hello"
  - "hello world"
Mental Model for State (v2)

Code:

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

State:

- **references**
  - `x`
  - `y`

- **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
```
How PythonTutor renders immutable types is configurable...

**Code:**

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

**Frames**

**Objects**

**Global frame**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>&quot;hello&quot;</td>
</tr>
<tr>
<td>y</td>
<td>&quot;hello world&quot;</td>
</tr>
</tbody>
</table>

**v1**

- inline primitives but don't nest objects [default]

**v2**

- render all objects on the heap (Python)
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)
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
Tuple Sequence

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

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**

\[
\text{nums\_list} = [200, 100, 300] \\
\text{nums\_tuple} = (200, 100, 300)
\]

\[
\begin{align*}
\text{x} &= \text{nums\_list}[2] \\
\text{x} &= \text{nums\_tuple}[2]
\end{align*}
\]

both put 300 in x

Like a list
- for loop, **indexing**, slicing, other methods

Unlike a list:
- immutable (like a string)
**Tuple Sequence**

```python
tuple_sequence = [(200, 100, 300)]
tuple_sequence[0] = 99
```

**As 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

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)

Crashes!
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
Example: location -> building mapping

```python
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

<table>
<thead>
<tr>
<th>type of parenthesis</th>
<th>uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>parentheses: ( and )</td>
<td>specifying order: (1+2) * 3</td>
</tr>
<tr>
<td></td>
<td>function invocation: f()</td>
</tr>
<tr>
<td></td>
<td>tuple: (1, 2, 3)</td>
</tr>
<tr>
<td>brackets: [ and ]</td>
<td>list creation: s = [1, 2, 3]</td>
</tr>
<tr>
<td></td>
<td>sequence indexing: s[-1]</td>
</tr>
<tr>
<td></td>
<td>sequence slicing: s[1:-2]</td>
</tr>
<tr>
<td></td>
<td>dict lookup: d[&quot;one&quot;]</td>
</tr>
<tr>
<td>braces: { and }</td>
<td>dict creation: d = {&quot;one&quot;:1, &quot;two&quot;:2}</td>
</tr>
<tr>
<td></td>
<td>set creation: {1, 2, 3}</td>
</tr>
</tbody>
</table>
A note on parenthetical characters

<table>
<thead>
<tr>
<th>type of parenthesis</th>
<th>uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>parentheses: ( and )</td>
<td>specifying order: ((1+2) * 3)</td>
</tr>
<tr>
<td></td>
<td>function invocation: (f())</td>
</tr>
<tr>
<td></td>
<td>tuple: ((1, 2, 3))</td>
</tr>
<tr>
<td></td>
<td>list creation: ([1, 2, 3])</td>
</tr>
<tr>
<td>brackets: [ and ]</td>
<td>sequence indexing: (s[-1])</td>
</tr>
<tr>
<td></td>
<td>sequence slicing: (s[1:-2])</td>
</tr>
<tr>
<td></td>
<td>dict lookup: (d[&quot;one&quot;])</td>
</tr>
<tr>
<td>braces: { and }</td>
<td>dict creation: {&quot;one&quot;:1, &quot;two&quot;:2}</td>
</tr>
<tr>
<td></td>
<td>set creation: ({1, 2, 3})</td>
</tr>
</tbody>
</table>
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?

```
people=[
    {
    "Fname": "Alice", "Iname": "Anderson", "age": 30},
    {
    "fname": "Bob", "lname": "Baker", "age": 31},
]

p = people[0]
print("Hello " + p["fname"] + " " + p["lname"])
```

```
people=[
    ("Alice", "Anderson", 30),
    ("Bob", "Baker", 31),
]

p = people[1]
print("Hello " + p[1] + " " + p[2])
```
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
people=[
    {"fname": "Alice", "lname": "Anderson", "age": 30},
    {"fname": "Bob", "lname": "Baker", "age": 31},
]

p = people[0]
print("Hello ", p["fname"] + " ", p["lname"])

people=[
    ("Alice", "Anderson", 30),
    ("Bob", "Baker", 31),
]

p = people[1]
print("Hello ", p[0] + " ", p[1])

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)
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("Alice", "Anderson", 30)

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("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")

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)

print("crashes immediately (good!)")
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

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:

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

State:

```
<table>
<thead>
<tr>
<th>references</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;this string is millions of ...&quot;</td>
</tr>
</tbody>
</table>
```
Reason 1: Performance

Code:

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

State:

- **references**
  - `x`
- **objects**
  - "this string is millions of ..."
  - `y`
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:

```
references
alice

objects
name
score
age
"Alice"
10
30
dict
```
Reason 2: Centralized Updates

```python
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:

```
alice references

bob objects

name: "Alice"
score: 10
age: 30

name: "Bob"
score: 8
age: 25
```
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"])
Reason 2: Centralized Updates

code:
```python
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:

- **References**
  - alice
  - bob
  - winner

- **Objects**
  - name: "Alice"
  - score: 10
  - age: 30
  - name: "Bob"
  - score: 8
  - age: 25

Diagram:

- alice references objects alice and bob
- bob references objects alice
- winner references objects alice
Reason 2: Centralized Updates

```python
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:

```
<table>
<thead>
<tr>
<th>references</th>
<th>objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>alice</td>
<td>name: &quot;Alice&quot;, score: 10, age: 31</td>
</tr>
<tr>
<td>bob</td>
<td>name: &quot;Bob&quot;, score: 8, age: 25</td>
</tr>
<tr>
<td>winner</td>
<td>dict: Alice: 10, Bob: 8</td>
</tr>
</tbody>
</table>
```

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