Objects

• Objects represent information
• They consist of data and behavior, bundled together to create abstractions
• Objects can represent things, but also properties, interactions, & processes
• A type of object is called a class; classes are first-class values in Python

• Object-oriented programming:
  • A metaphor for organizing large programs
  • Special syntax that can improve the composition of programs
  • In Python, every value is an object
  • All objects have attributes
  • A lot of data manipulation happens through object methods
  • Functions do one thing; objects do many related things

Example: Strings

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

- Layout was chosen to support sorting by character code
- Rows indexed 2-5 are a useful 6-bit (64 element) subset
- Control characters were designed for transmission

latin capital letter A

die face 6

representing strings: the unicode standard

- 137,994 characters in Unicode 12.1
- 138 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order
- A canonical name for every character
Identity Operators

Some Objects Can Change

Identity

- `is` evaluates to `True` if both `<exp1>` and `<exp2>` evaluate to the same object

Equality

- `==` evaluates to `True` if both `<exp1>` and `<exp2>` evaluate to equal values

Identical objects are always equal values

(Memo)

Mutation Can Happen Within a Function Call

A function can change the value of any object in its scope

```python
four = [1, 2, 3, 4]
def mystery(s):
    s.pop()  # `s` is a list
    return s
mystery(four)
```

(Tuples)

Tuples

Immutable values are protected from mutation

```python
>>> turtle = (1, 2, 3)
>>> turtle  # Next lecture: more can change turtle’s binding
(1, 2, 3)
```

The value of an expression can change because of changes in names or objects

```python
>>> x = 2
>>> x += 1
```

Name change:

```python
>>> x = 3
>>> x += 2
```

Object notation:

```python
>>> x = x + x
```

An immutable sequence may still change if it contains a mutable value as an element

```python
>>> a = (1, 2, 3)
>>> a[1] = 4
```

Sameness and Change

- As long as we never modify objects, a compound object is just the totality of its pieces
- A rational number is just its numerator and denominator
- This view is no longer valid in the presence of change
- A compound data object has an "identity" in addition to the pieces of which it is composed
- A list is still "the same" list even if we change its contents
- Conversely, we could have two lists that happen to have the same contents, but are different

```python
>>> a = [10]
>>> b = a
>>> a == b
True
>>> a.append(20)
>>> a
[10, 20]
>>> b
[10, 20]
>>> a == b
False
```

Mutable Default Arguments are Dangerous

A default argument value is part of a function value, not generated by a call

```python
>>> def f(u=[]):
...     u.append(3)
...     return len(u)
...f()  # Each time the function is called, `u` is bound to the same value!
```

(Mutable Default Arguments are Dangerous)
Lists in Environment Diagrams

Assume that before each example below we execute:

\( t = [5, 6] \)
\( s = [2, 3] \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complex</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append all existing elements to a list</td>
<td><code>t.append(s)</code></td>
<td><code>t = [5, 6, 2, 3]</code></td>
</tr>
<tr>
<td>Extend all existing elements in another list</td>
<td><code>s.extend(t)</code></td>
<td><code>s = [2, 3, 5, 6]</code></td>
</tr>
<tr>
<td>Insertion &amp; slicing create new lists containing existing elements &amp; modifies the last element</td>
<td><code>b[1][1] = 0</code></td>
<td><code>b = [3, 5, 0]</code></td>
</tr>
<tr>
<td>Slice assignment replaces a slice with new values</td>
<td><code>s[0:0] = t</code></td>
<td><code>s = [2, 3, 5, 6]</code></td>
</tr>
</tbody>
</table>

Assume that before each example below we execute:

\( t = [5, 6] \)
\( s = [2, 3] \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complex</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append all existing elements to a list</td>
<td><code>t.append(s)</code></td>
<td><code>t = [5, 6, 2, 3]</code></td>
</tr>
<tr>
<td>Extend all existing elements in another list</td>
<td><code>s.extend(t)</code></td>
<td><code>s = [2, 3, 5, 6]</code></td>
</tr>
<tr>
<td>Insertion &amp; slicing create new lists containing existing elements &amp; modifies the last element</td>
<td><code>b[1][1] = 0</code></td>
<td><code>b = [3, 5, 0]</code></td>
</tr>
<tr>
<td>Slice assignment replaces a slice with new values</td>
<td><code>s[0:0] = t</code></td>
<td><code>s = [2, 3, 5, 6]</code></td>
</tr>
</tbody>
</table>

Assume that before each example below we execute:

\( t = [5, 6] \)
\( s = [2, 3] \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complex</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append all existing elements to a list</td>
<td><code>t.append(s)</code></td>
<td><code>t = [5, 6, 2, 3]</code></td>
</tr>
<tr>
<td>Extend all existing elements in another list</td>
<td><code>s.extend(t)</code></td>
<td><code>s = [2, 3, 5, 6]</code></td>
</tr>
<tr>
<td>Insertion &amp; slicing create new lists containing existing elements &amp; modifies the last element</td>
<td><code>b[1][1] = 0</code></td>
<td><code>b = [3, 5, 0]</code></td>
</tr>
<tr>
<td>Slice assignment replaces a slice with new values</td>
<td><code>s[0:0] = t</code></td>
<td><code>s = [2, 3, 5, 6]</code></td>
</tr>
</tbody>
</table>

Assume that before each example below we execute:

\( t = [5, 6] \)
\( s = [2, 3] \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complex</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append all existing elements to a list</td>
<td><code>t.append(s)</code></td>
<td><code>t = [5, 6, 2, 3]</code></td>
</tr>
<tr>
<td>Extend all existing elements in another list</td>
<td><code>s.extend(t)</code></td>
<td><code>s = [2, 3, 5, 6]</code></td>
</tr>
<tr>
<td>Insertion &amp; slicing create new lists containing existing elements &amp; modifies the last element</td>
<td><code>b[1][1] = 0</code></td>
<td><code>b = [3, 5, 0]</code></td>
</tr>
<tr>
<td>Slice assignment replaces a slice with new values</td>
<td><code>s[0:0] = t</code></td>
<td><code>s = [2, 3, 5, 6]</code></td>
</tr>
</tbody>
</table>

Assume that before each example below we execute:

\( t = [5, 6] \)
\( s = [2, 3] \)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complex</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append all existing elements to a list</td>
<td><code>t.append(s)</code></td>
<td><code>t = [5, 6, 2, 3]</code></td>
</tr>
<tr>
<td>Extend all existing elements in another list</td>
<td><code>s.extend(t)</code></td>
<td><code>s = [2, 3, 5, 6]</code></td>
</tr>
<tr>
<td>Insertion &amp; slicing create new lists containing existing elements &amp; modifies the last element</td>
<td><code>b[1][1] = 0</code></td>
<td><code>b = [3, 5, 0]</code></td>
</tr>
<tr>
<td>Slice assignment replaces a slice with new values</td>
<td><code>s[0:0] = t</code></td>
<td><code>s = [2, 3, 5, 6]</code></td>
</tr>
</tbody>
</table>