61A Lecture 22
Announcements
Lists
Lists in Environment Diagrams
Lists in Environment Diagrams

Assume that before each example below we execute:

\[
s = [2, 3] \\
t = [5, 6]
\]
Lists in Environment Diagrams

Assume that before each example below we execute:
s = [2, 3]
t = [5, 6]
Lists in Environment Diagrams

Assume that before each example below we execute:

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
</table>


Lists in Environment Diagrams

Assume that before each example below we execute:

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
</table>
### Lists in Environment Diagrams

Assume that before each example below we execute:

```python
s = [2, 3]
t = [5, 6]
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>append</code></td>
<td>adds one element to a list</td>
<td></td>
</tr>
</tbody>
</table>
### Lists in Environment Diagrams

Assume that before each example below we execute:

```
s = [2, 3]
t = [5, 6]
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong></td>
<td><code>s.append(t)</code></td>
<td><code>t = 0</code></td>
</tr>
<tr>
<td>adds one element to a list</td>
<td>s.append(t)</td>
<td>t = 0</td>
</tr>
</tbody>
</table>
Lists in Environment Diagrams

Assume that before each example below we execute:

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong> adds one element to a list</td>
<td>( s.append(t) )  ( t = 0 )</td>
<td>( s \rightarrow [2, 3, [5, 6]] )  ( t \rightarrow 0 )</td>
</tr>
</tbody>
</table>
Lists in Environment Diagrams

Assume that before each example below we execute:
\[ s = [2, 3] \]
\[ t = [5, 6] \]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong></td>
<td>s.append(t)</td>
<td>( s \rightarrow [2, 3, [5, 6]] )</td>
</tr>
<tr>
<td></td>
<td>( t = 0 )</td>
<td>( t \rightarrow 0 )</td>
</tr>
</tbody>
</table>

**extend** adds all elements in one list to another list
Lists in Environment Diagrams

Assume that before each example below we execute:
s = [2, 3]
t = [5, 6]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong> adds one element to a list</td>
<td>s.append(t)</td>
<td>s → [2, 3, [5, 6]]</td>
</tr>
<tr>
<td></td>
<td>t = 0</td>
<td>t → 0</td>
</tr>
<tr>
<td><strong>extend</strong> adds all elements in one list to another list</td>
<td>s.extend(t)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t[1] = 0</td>
<td></td>
</tr>
</tbody>
</table>
Lists in Environment Diagrams

Assume that before each example below we execute:
s = [2, 3]
t = [5, 6]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
</table>
| `append` adds one element to a list | s.append(t)  
t = 0 | s → [2, 3, [5, 6]]  
t → 0 |
| `extend` adds all elements in one list to another list | s.extend(t)  
t[1] = 0 | s → [2, 3, 5, 6]  
t → [5, 0] |
Lists in Environment Diagrams

Assume that before each example below we execute:

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
</table>
| append adds one element to a list | s.append(t)  
|                    | t = 0         | \[2, 3, [5, 6]\] \[5, 6\] |
| extend adds all elements in one list to another list | s.extend(t)  
|                    | t[1] = 0      | \[2, 3, 5, 6\] \[5, 0\] |
| addition & slicing create new lists containing existing elements | | |
Lists in Environment Diagrams

Assume that before each example below we execute:

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong> adds one element to a list</td>
<td>( s.append(t) ) ( t = 0 )</td>
<td>( s \rightarrow [2, 3, [5, 6]] ) ( t \rightarrow 0 )</td>
</tr>
<tr>
<td><strong>extend</strong> adds all elements in one list to another list</td>
<td>( s.extend(t) ) ( t[1] = 0 )</td>
<td>( s \rightarrow [2, 3, 5, 6] ) ( t \rightarrow [5, 0] )</td>
</tr>
<tr>
<td><strong>addition &amp; slicing</strong> create new lists containing existing elements</td>
<td>( a = s + [t] ) ( b = a[1:] ) ( a[1] = 9 ) ( b[1][1] = 0 )</td>
<td></td>
</tr>
</tbody>
</table>
Lists in Environment Diagrams

Assume that before each example below we execute:
\[
s = [2, 3] \\
t = [5, 6]
\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
</table>
| **append** adds one element to a list | s.append(t)
  t = 0 |
|                         | s → [2, 3, [5, 6]] |
|                         | t → 0              |
| **extend** adds all elements in one list to another list | s.extend(t)
  t[1] = 0 |
|                         | s → [2, 3, 5, 6]   |
|                         | t → [5, 0]         |
| **addition & slicing** create new lists containing existing elements | a = s + [t]
  b = a[1:]
  a[1] = 9
  b[1][1] = 0 |
|                         |                   |

Global
Lists in Environment Diagrams

Assume that before each example below we execute:

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
</table>
| append adds one element to a list | `s.append(t)`  
 `t = 0` | `s \rightarrow [2, 3, [5, 6]]`  
 `t \rightarrow 0` |
| extend adds all elements in one list to another list | `s.extend(t)`  
 `t[1] = 0` | `s \rightarrow [2, 3, 5, 6]`  
 `t \rightarrow [5, 0]` |
| addition & slicing create new lists containing existing elements | `a = s + [t]`  
 `b = a[1:]`  
 `a[1] = 9`  
 `b[1][1] = 0` | }
Lists in Environment Diagrams

Assume that before each example below we execute:
s = [2, 3]
t = [5, 6]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>append adds one element to a list</td>
<td>s.append(t) t = 0</td>
<td>s → [2, 3, [5, 6]] t → 0</td>
</tr>
<tr>
<td>extend adds all elements in one list to another list</td>
<td>s.extend(t) t[1] = 0</td>
<td>s → [2, 3, 5, 6] t → [5, 0]</td>
</tr>
<tr>
<td>addition &amp; slicing create new lists containing existing elements</td>
<td>a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0</td>
<td></td>
</tr>
</tbody>
</table>
Lists in Environment Diagrams

Assume that before each example below we execute:

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
</table>
| **append** adds one element to a list | \( s.append(t) \)
  \( t = 0 \) | \( s \rightarrow [2, 3, [5, 6]] \)
  \( t \rightarrow 0 \) |
| **extend** adds all elements in one list to another list | \( s.extend(t) \)
  \( t[1] = 0 \) | \( s \rightarrow [2, 3, 5, 6] \)
  \( t \rightarrow [5, 0] \) |
| **addition & slicing** create new lists containing existing elements | \( a = s + [t] \)
  \( b = a[1:] \)
  \( a[1] = 9 \)
  \( b[1][1] = 0 \) | |
Lists in Environment Diagrams

Assume that before each example below we execute:
s = [2, 3]
t = [5, 6]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong> adds one element to a list</td>
<td>s.append(t) \n t = 0</td>
<td>s → [2, 3, [5, 6]] \n t → 0</td>
</tr>
<tr>
<td><strong>extend</strong> adds all elements in one list to another list</td>
<td>s.extend(t) \n t[1] = 0</td>
<td>s → [2, 3, 5, 6] \n t → [5, 0]</td>
</tr>
<tr>
<td><strong>addition &amp; slicing</strong> create new lists containing existing elements</td>
<td>a = s + [t] \n b = a[1:]</td>
<td>a[1] = 9 \n b[1][1] = 0</td>
</tr>
</tbody>
</table>
Lists in Environment Diagrams

Assume that before each example below we execute:
\[ s = [2, 3] \]
\[ t = [5, 6] \]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>append adds one element to a list</td>
<td>s.append(t) t = 0</td>
<td>[2, 3, [5, 6]] t = 0</td>
</tr>
<tr>
<td>extend adds all elements in one list to another list</td>
<td>s.extend(t) t[1] = 0</td>
<td>[2, 3, 5, 6] t = [5, 0]</td>
</tr>
<tr>
<td>addition &amp; slicing create new lists containing existing elements</td>
<td>a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0</td>
<td></td>
</tr>
</tbody>
</table>
Lists in Environment Diagrams

Assume that before each example below we execute:
s = [2, 3]
t = [5, 6]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
</table>
| append adds one element to a list | s.append(t)
  t = 0               | s = [2, 3, [5, 6]]
  t = 0               |
| extend adds all elements in one list to another list | s.extend(t)
  t[1] = 0         | s = [2, 3, 5, 6]
  t = [5, 0]         |
| addition & slicing create new lists containing existing elements | a = s + [t]
  b = a[1:]         | a[1] = 9
  b[1][1] = 0       |

Global

```
Global
  s
  t
```

```
Global
  list
  0 2
  1 3
```

```
Global
  list
  0 5
  1 6
```

```
Global
  list
  0 3
  1
```

```
Global
  list
  0 2
  1 9
  2
```

```
Global
  list
  0 1
```

```
Global
  list
  0
  1
```
Lists in Environment Diagrams

Assume that before each example below we execute:
s = [2, 3]
t = [5, 6]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong> adds one element to a list</td>
<td>s.append(t) t = 0</td>
<td>s → [2, 3, [5, 6]] t → 0</td>
</tr>
<tr>
<td><strong>extend</strong> adds all elements in one list to another list</td>
<td>s.extend(t) t[1] = 0</td>
<td>s → [2, 3, 5, 6] t → [5, 0]</td>
</tr>
<tr>
<td><strong>addition &amp; slicing</strong> create new lists containing existing elements</td>
<td>a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0</td>
<td></td>
</tr>
</tbody>
</table>
Lists in Environment Diagrams

Assume that before each example below we execute:
s = \[2, 3\]
t = \[5, 6\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>append adds one element to a list</td>
<td>s.append(t)</td>
<td>s → [2, 3, [5, 6]] t → 0</td>
</tr>
<tr>
<td>extend adds all elements in one list to another list</td>
<td>s.extend(t)</td>
<td>s → [2, 3, 5, 6] t → [5, 0]</td>
</tr>
<tr>
<td>addition &amp; slicing create new lists containing existing elements</td>
<td>a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0</td>
<td>s → [2, 3] t → [5, 0] a → [2, 9, [5, 0]] b → [3, [5, 0]]</td>
</tr>
</tbody>
</table>
Lists in Environment Diagrams

Assume that before each example below we execute:
s = [2, 3]
t = [5, 6]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong> adds one element to a list</td>
<td>s.append(t) t = 0</td>
<td>s → [2, 3, [5, 6]] t → 0</td>
</tr>
<tr>
<td><strong>extend</strong> adds all elements in one list to another list</td>
<td>s.extend(t) t[1] = 0</td>
<td>s → [2, 3, 5, 6] t → [5, 0]</td>
</tr>
<tr>
<td><strong>addition &amp; slicing</strong> create new lists containing existing elements</td>
<td>a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0</td>
<td>s → [2, 3] t → [5, 0] a → [2, 9, [5, 0]] b → [3, [5, 0]]</td>
</tr>
</tbody>
</table>

The list function also creates a new list containing existing elements.
Lists in Environment Diagrams

Assume that before each example below we execute:
s = [2, 3]
t = [5, 6]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>append adds one element to a list</td>
<td>s.append(t)</td>
<td>s → [2, 3, [5, 6]]</td>
</tr>
<tr>
<td></td>
<td>t = 0</td>
<td>t → 0</td>
</tr>
<tr>
<td>extend adds all elements in one list to another list</td>
<td>s.extend(t)</td>
<td>s → [2, 3, 5, 6]</td>
</tr>
<tr>
<td></td>
<td>t[1] = 0</td>
<td>t → [5, 0]</td>
</tr>
<tr>
<td>addition &amp; slicing create new lists containing existing elements</td>
<td>a = s + [t]</td>
<td>s → [2, 3]</td>
</tr>
<tr>
<td></td>
<td>b = a[1:]</td>
<td>t → [5, 0]</td>
</tr>
<tr>
<td></td>
<td>a[1] = 9</td>
<td>a → [2, 9, [5, 0]]</td>
</tr>
<tr>
<td></td>
<td>b[1][1] = 0</td>
<td>b → [3, [5, 0]]</td>
</tr>
<tr>
<td>The list function also creates a new list containing existing elements</td>
<td>t = list(s)</td>
<td>s[1] = 0</td>
</tr>
<tr>
<td></td>
<td>s[1] = 0</td>
<td></td>
</tr>
</tbody>
</table>
Lists in Environment Diagrams

Assume that before each example below we execute:
s = [2, 3]
t = [5, 6]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong> adds one element to a list</td>
<td>s.append(t) t = 0</td>
<td>s → [2, 3, [5, 6]] t → 0</td>
</tr>
<tr>
<td><strong>extend</strong> adds all elements in one list to another list</td>
<td>s.extend(t) t[1] = 0</td>
<td>s → [2, 3, 5, 6] t → [5, 0]</td>
</tr>
<tr>
<td><strong>addition &amp; slicing</strong> create new lists containing existing elements</td>
<td>a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0</td>
<td>s → [2, 3] t → [5, 0] a → [2, 9, [5, 0]] b → [3, [5, 0]]</td>
</tr>
<tr>
<td>The <strong>list</strong> function also creates a new list containing existing elements</td>
<td>t = list(s) s[1] = 0</td>
<td>s → [2, 0] t → [2, 3]</td>
</tr>
</tbody>
</table>
### Lists in Environment Diagrams

**Assume that before each example below we execute:**

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong> adds one element to a list</td>
<td><code>s.append(t)</code> ( t = 0 )</td>
<td>( s \to [2, 3, [5, 6]] ) ( t \to 0 )</td>
</tr>
<tr>
<td><strong>extend</strong> adds all elements in one list to another list</td>
<td><code>s.extend(t)</code> ( t[1] = 0 )</td>
<td>( s \to [2, 3, 5, 6] ) ( t \to [5, 0] )</td>
</tr>
<tr>
<td><strong>addition &amp; slicing</strong> create new lists containing existing elements</td>
<td><code>a = s + [t]</code> ( b = a[1:] ) ( a[1] = 9 ) ( b[1][1] = 0 )</td>
<td>( s \to [2, 3] ) ( t \to [5, 0] ) ( a \to [2, 9, [5, 0]] ) ( b \to [3, [5, 0]] )</td>
</tr>
<tr>
<td>The <strong>list</strong> function also creates a new list containing existing elements</td>
<td><code>t = list(s)</code> ( s[1] = 0 )</td>
<td>( s \to [2, 0] ) ( t \to [2, 3] )</td>
</tr>
<tr>
<td><strong>slice assignment</strong> replaces a slice with new values</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Environment Diagram](image)
Lists in Environment Diagrams

Assume that before each example below we execute:
s = [2, 3]
t = [5, 6]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>append</strong> adds one element to a list</td>
<td>s.append(t) t = 0</td>
<td>s → [2, 3, [5, 6]] t → 0</td>
</tr>
<tr>
<td><strong>extend</strong> adds all elements in one list to another list</td>
<td>s.extend(t) t[1] = 0</td>
<td>s → [2, 3, 5, 6] t → [5, 0]</td>
</tr>
<tr>
<td><strong>addition &amp; slicing</strong> create new lists containing existing elements</td>
<td>a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0</td>
<td>s → [2, 3] t → [5, 0] a → [2, 9, [5, 0]] b → [3, [5, 0]]</td>
</tr>
<tr>
<td>The list function also creates a new list containing existing elements</td>
<td>t = list(s) s[1] = 0</td>
<td>s → [2, 0] t → [2, 3]</td>
</tr>
<tr>
<td><strong>slice assignment</strong> replaces a slice with new values</td>
<td>s[0:0] = t s[3:] = t t[1] = 0</td>
<td></td>
</tr>
</tbody>
</table>
Lists in Environment Diagrams

Assume that before each example below we execute:
\[ s = [2, 3] \]
\[ t = [5, 6] \]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>append adds one element to a list</td>
<td><code>s.append(t)</code> t = 0</td>
<td>[2, 3, 5, 6] t → 0</td>
</tr>
<tr>
<td>extend adds all elements in one list to another list</td>
<td><code>s.extend(t)</code> t[1] = 0</td>
<td>[2, 3, 5, 6] t → [5, 0]</td>
</tr>
<tr>
<td>addition &amp; slicing create new lists containing existing elements</td>
<td><code>a = s + [t]</code> <code>b = a[1:]</code> a[1] = 9 b[1][1] = 0</td>
<td>[2, 3] t → [5, 0] [2, 9, 5, 0] t → [3, [5, 0]]</td>
</tr>
<tr>
<td>The list function also creates a new list containing existing elements</td>
<td><code>t = list(s)</code> s[1] = 0</td>
<td>[2, 0] t → [2, 3]</td>
</tr>
<tr>
<td>slice assignment replaces a slice with new values</td>
<td><code>s[0:0] = t</code> s[3:] = t t[1] = 0</td>
<td>[5, 6, 2, 5, 6] t → [5, 0]</td>
</tr>
</tbody>
</table>
Lists in Lists in Lists in Environment Diagrams

t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)

---

t = [[1, 2], [3, 4]]
t[0].append(t[1:2])
Lists in Lists in Lists in Environment Diagrams

t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)

\[
\begin{array}{c}
\text{Global} \\
\text{t} \quad \text{list} \\
| & | & |
\end{array}
\]

0 1 2 3

\[
\begin{array}{c}
\text{t} \quad \text{t[0].append(t[1:2])} \\
[1, 2], [3, 4] \\
\end{array}
\]
Lists in Lists in Lists in Environment Diagrams

t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)

[t] evaluates to:

Global

t

list

0 1 2 3

list

0

[t] evaluates to:

t = [[1, 2], [3, 4]]
t[0].append(t[1:2])
Lists in Lists in Lists in Environment Diagrams

t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)

[t] evaluates to:

```python
t = [[1, 2], [3, 4]]
t[0].append(t[1:2])
```
Lists in Lists in Lists in Environment Diagrams

t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)

t = [[1, 2], [3, 4]]
t[0].append(t[1:2])
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)

Global          list
  t               1 2 3


t = [[1, 2], [3, 4]]
t[0].append(t[1:2])
Lists in Lists in Lists in Environment Diagrams

```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```

```
t = [[1, 2], [3, 4]]
t[0].append(t[1:2])
```
Lists in Lists in Lists in Environment Diagrams

```python
# Example 1

t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)

# Example 2

t = [[1, 2], [3, 4]]
t[0].append(t[1:2])
```

![Environment Diagram](image-url)
Lists in Lists in Lists in Environment Diagrams

$t = [1, 2, 3]$
t[1:3] = [t]
t.extend(t)

$[1, [...], 1, [...]]$

$t = [[1, 2], [3, 4]]$
t[0].append(t[1:2])
Lists in Lists in Lists in Environment Diagrams

t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)

```
[1, [...], 1, [...]]
```

t = [[1, 2], [3, 4]]
t[0].append(t[1:2])

```
[[1, 2, 3, 4], [...], [...]]
```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)

[1, [...], 1, [...]]

t = [[1, 2], [3, 4]]
t[0].append(t[1:2])
Lists in Lists in Lists in Environment Diagrams

t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)

Global
     t

list

1 2 3

1

1 2 3

[1, [...], 1, [...]]

t = [[1, 2], [3, 4]]
t[0].append(t[1:2])

Global
     t

list

0 1

0

1

2 2

0

[[1, 2, [[3, 4]]], [3, 4]]
Objects
Land Owners

Instance attributes are found before class attributes; class attributes are inherited
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'

def __init__(self):
    self.elf = Worker

def work(self):
    return self.greeting + ', I work'

def __repr__(self):
    return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'

def work(self):
    print(Worker.work(self))
    return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```
**Land Owners**

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

>>> Worker().work()

>>> jack

>>> jack.work()

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

>>> jack = Worker()
>>> john = Bourgeoisie()
>>> jack.greeting = 'Maam'
```

```python
>>> john.work()  # Uncomment to run this line

>>> john.elf.work(john)  # Uncomment to run this line
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()  
<class Worker>  
greeting: 'Sir'

>>> jack

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work() <class Worker>
greeting: 'Sir'

>>> jack
<class Bourgeoisie>
greeting: 'Peon'

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
```

```python
>>> Worker().work()
<class Worker>
greeting: 'Sir'

>>> jack
<class Bourgeoisie>
greeting: 'Peon'

>>> jack.work()

>>> jack.elf.work(john)
```
Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.greeting'

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
<class Worker>
greeting: 'Sir'

>>> jack
<class Bourgeoisie>
greeting: 'Peon'

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)

jack <Worker>
elf: 

john <Bourgeoisie>
elf: 
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
<class Worker>
greeting: 'Sir'

>>> jack
<class Bourgeoisie>
greeting: 'Peon'

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)

jack <Worker>
elf: 
greeting: 'Maam'

john <Bourgeoisie>
elf: 

Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.greeting'

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
<class Worker>
greeting: 'Sir'

>>> jack
<class Bourgeoisie>
greeting: 'Peon'

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)

jack <Worker>
elf: 
greeting: 'Maam'

john <Bourgeoisie>
elf: 
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.greeting'

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
<class Worker>
greeting: 'Sir'

>>> jack.work()
<class Bourgeoisie>
greeting: 'Peon'

>>> john.work()

>>> john.elf.work(john)

jack <Worker>
elf: john <Bourgeoisie>
egreeting: 'Maam'

john <Bourgeoisie>
egreeting: 'Maam'
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
greeting = 'Sir'
def __init__(self):
    self.elf = Worker
def work(self):
    return self.greeting + ', I work'
def __repr__(self):
    return 'Bourgeoisie.greeting'

class Bourgeoisie(Worker):
greeting = 'Peon'
def work(self):
    print(Worker.work(self))
    return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
<class Worker>
greeting: 'Sir'

>>> jack.work()
<class Bourgeoisie>
greeting: 'Peon'

>>> john.work()
<class Bourgeoisie>
greeting: 'Maam'

>>> john.elf.work(john)
<_class Worker>
elf: 

greeting: 'Maam'

john <Bourgeoisie>
elf: 

---
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.greeting'

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)

<class Worker>
greeting: 'Sir'

<class Bourgeoisie>
greeting: 'Peon'

greeting:
'Maam'

e: 

e:

john <Bourgeoisie>

e: 

Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.' + self.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()

>>> john.work()

>>> john.elf.work(john)
```

---

<box>

```python
<class Worker>
greeting: 'Sir'

<class Bourgeoisie>
greeting: 'Peon'

greeting: 'Maam'

ejohn <Bourgeoisie>
elf: john

ejack <Worker>
greeting: 'Maam'

ejack <Worker>
elf: 
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting
class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'
jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'
>>> Worker().work()
'Sir, I work'
>>> jack
Peon
>>> jack.work()    
'Maam, I work'
>>> john.work()    
'I gather wealth'
>>> john.elf.work(john)
```

- jack <Worker>
  - greeting: 'Sir'
- john <Bourgeoisie>
  - elf: 
    - greeting: 'Maam'
  - elf: 
    - greeting: 'Peon'
```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()

>>> john.elf.work(john)

jack <Worker>
greeting: 'Sir'
elf:  

john <Bourgeoisie>
greeting: 'Maam'
elf:  

<class Worker>
greeting: 'Sir'
elf:  

class Bourgeoisie(greeting: 'Peon')
elf:  

<class Bourgeoisie><class Worker>
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
'Peon, I work
'I gather wealth'

>>> john.elf.work(john)

<class Worker>
greeting: 'Sir'
<class Bourgeoisie>
greeting: 'Peon'
jack <Worker>
elf: john <Bourgeoisie>
greeting: 'Maam'
john <Bourgeoisie>
elf: 

Instance attributes are found before class attributes; class attributes are inherited

```python
class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return 'Bourgeoisie.

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
'Peon, I work
'I gather wealth'

>>> john.elf.work(john)

<class Worker>
greeting: 'Sir'

ejack <Worker>
elf: 
greeting: 'Peon'

<class Bourgeoisie>
greeting: 'Peon'

john <Bourgeoisie>
elf:
```

Land Owners

```
Land Owners

Instance attributes are found before class attributes; class attributes are inherited

class Worker:
    greeting = 'Sir'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', I work'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Peon'
    def work(self):
        print(Worker.work(self))
        return 'I gather wealth'

jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Maam'

>>> Worker().work()
'Sir, I work'

>>> jack
Peon

>>> jack.work()
'Maam, I work'

>>> john.work()
'Peon, I work'

>>> john.elf.work(john)
'Peon, I work'

<class Worker>
    greeting: 'Sir'

<class Bourgeoisie>
    greeting: 'Peon'

jack <Worker>
    elf: john <Bourgeoisie>
    greeting: 'Maam'

john <Bourgeoisie>
    elf: ___
Linked Lists
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link.

The rest of a linked list can contain the linked list as a sub-list.

```python
>>> s = Link(1, Link(2, Link(3)))
```
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

>>> s = Link(1, Link(2, Link(3)))
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link.

The rest of a linked list can contain the linked list as a sub-list.

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
2
```

Note: The actual environment diagram is much more complicated.
Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```python
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
2
```

Note: The actual environment diagram is much more complicated.
Trees
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

*Problem*: Implement `morse` so that `decode` works correctly
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.}
```

A: 📤 🔹
B: 📤 📤 📤 🔹
C: 📤 📤 📤 📤 🔹
D: 📤 📤 🔹
E: 📤
...
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter."

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']

    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.}
def decode(signals, tree):
    """Decode signals into a letter."

    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```python
def morse(code):
    ...
```

```
>>> t = morse(abcde)
>>> [decode(s, t) for s in ['-..', '.', '-.--', '.-', '-..', '.']]
['d', 'e', 'c', 'a', 'd', 'e']
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
def decode(signals, tree):
    """Decode signals into a letter.""
    tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```bash
>>> t = morse(abcde)
>>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
['d', 'e', 'c', 'a', 'd', 'e']
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.}
def decode(signals, tree):
    """Decode signals into a letter."
    tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter.
    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```python
def morse(code):
    ...
```

```python
def decode('.', t)
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly.

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter.""
    tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```python
>>> t = morse(abcde)
```

```python
>>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
['d', 'e', 'c', 'a', 'd', 'e']
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem:** Implement `morse` so that `decode` works correctly

```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter."

    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label

def morse(code):
    ....
```

```python
>>> t = morse(abcde)
>>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
['d', 'e', 'c', 'a', 'd', 'e']
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
def decode(signals, tree):
    """Decode signals into a letter."
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
        assert len(leaves) == 1
        return leaves[0].label

def morse(code):
    ....

>>> t = morse(abcde)
>>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
['d', 'e', 'c', 'a', 'd', 'e']
```

A: 🍈 🍈
B: 🍈 🍈 🍈 🍈
C: 🍈 🍈 🍈 🍈 🍈
D: 🍈 🍈 🍈
E: 🍈
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
def decode(signals, tree):
    """Decode signals into a letter."
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
        assert len(leaves) == 1
        return leaves[0].label
def morse(code):
    ...
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}

def decode(signals, tree):
    """Decode signals into a letter."

    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
        assert len(leaves) == 1
        return leaves[0].label

    return decode('.', t)
```

```python
>>> t = morse(abcde)
>>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
['d', 'e', 'c', 'a', 'd', 'e']
```
Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly.

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
def decode(signals, tree):
    """Decode signals into a letter."
    tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```python
>>> t = morse(abcde)
>>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
['d', 'e', 'c', 'a', 'd', 'e']
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly.

```python
code = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
def decode(signals, tree):
    """Decode signals into a letter."
    tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```python
def morse(code):
    ....
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals

Problem: Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.}
def decode(signals, tree):
    """Decode signals into a letter."

    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
        assert len(leaves) == 1
        return leaves[0].label
```

```python
>>> t = morse(abcde)
>>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
['d', 'e', 'c', 'a', 'd', 'e']
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly.

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
def decode(signals, tree):
    """Decode signals into a letter."
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
        assert len(leaves) == 1
    return leaves[0].label
```

```python
def morse(code):
    ....
```

```python
>>> t = morse(abcde)
>>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
['d', 'e', 'c', 'a', 'd', 'e']
```
Morse Code

Morse code is a signaling protocol that transmits messages by sequences of signals.

**Problem:** Implement `morse` so that `decode` works correctly

```python
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
def decode(signals, tree):
    """Decode signals into a letter."
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
        leaves = [b for b in tree.branches if b.is_leaf()]
        assert len(leaves) == 1
        return leaves[0].label
```

(Demo)