15. Lists are Objects

Topics:

References
Alias
More on Slicing
Comparing Lists

You can use `==` to compare two lists

```python
>>> x = [10, 20, 30, 40]
>>> y = [10, 20, 30, 40]
>>> x == y
True
```
## Comparing Lists

You can use `==` to compare two lists:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
</tr>
</tbody>
</table>

The Boolean expression `x==y` is True because `x` and `y` have the same length and identical values in each element.
Comparing Lists

You can use `==` to compare two lists

```python
>>> x = [1,2,3]
>>> y = [1.0,2.0,3.0]
>>> x==y
True
```

If there are ints and floats, convert everything to float then compare
Comparing Lists

Do not use <, <=, >, >= to compare two lists

>>> x = [10, 20, 30, 40]
>>> y = [11, 21, 31, 41]
>>> x<y
True
>>> y<x
True

Unpredictable
**Aliasing**

This:

\[
x = [10, 20, 30, 40]
\]
\[
y = x
\]

Results in this:

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>y</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>
Aliasing

Things to say:

x and y are variables that refer to the same list object.

The object is aliased because it has more than one name.
Tracking Changes

\[ x = [10, 20, 30, 40] \]
\[ y = x \]
\[ y = [1, 2, 3] \]

\[ x \rightarrow \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>---</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>---</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>---</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>---</td>
<td>40</td>
</tr>
</tbody>
</table>
Tracking Changes

\[ \text{x} = [10, 20, 30, 40] \]
\[ \text{y} = \text{x} \]
\[ \text{y} = [1, 2, 3] \]
Tracking Changes

\[ x = [10, 20, 30, 40] \]
\[ y = x \]
\[ y = [1, 2, 3] \]

\[ x \rightarrow \]
\[ 0 \rightarrow 10 \]
\[ 1 \rightarrow 20 \]
\[ 2 \rightarrow 30 \]
\[ 3 \rightarrow 40 \]

\[ y \rightarrow \]
\[ 0 \rightarrow 1 \]
\[ 1 \rightarrow 2 \]
\[ 2 \rightarrow 3 \]
The is Operator

```python
>>> x = [10, 20, 30, 40]
>>> y = [10, 20, 30, 40]
>>> x is y
False
```

Even though the two lists have the same component values, `x` and `y` do not refer to the same object.
The is Operator

```python
>>> x = [10, 20, 30, 40]
>>> y = x
>>> x is y
True
```

$x$ and $y$ refer to the same object

---

```
x --
y-->
```

```
0  -->  10
1  -->  20
2  -->  30
3  -->  40
```

$x$ and $y$ refer to the same object
Making a Copy of a List

\[ x = [10, 20, 30, 40] \]
\[ y = \text{list}(x) \]
Making a Copy of a List

\[ x = [10, 20, 30, 40] \]

\[ y = \text{list}(x) \]
Slices Create new Objects

\[ x = [10, 20, 30, 40] \]
\[ y = x[1:] \]
Slices Create New Objects

\[ x = [10, 20, 30, 40] \]
\[ y = x[1:] \]

\[ x \rightarrow \]

\[
\begin{array}{c|c}
0 & 10 \\
1 & 20 \\
2 & 30 \\
3 & 40 \\
\end{array}
\]

\[ y \rightarrow \]

\[
\begin{array}{c|c}
0 & 20 \\
1 & 30 \\
2 & 40 \\
\end{array}
\]
Careful!

\[
x = [10, 20, 30, 40]
\]

\[
y = x
\]

\[
y = x[1:]
\]
Careful!

\[ x = [10, 20, 30, 40] \]

\[ y = x \]

\[ y = x[1:] \]
Careful!

\[
x = [10, 20, 30, 40]
\]

\[
y = x
\]

\[
y = x[1:]
\]
Void Functions

\[ x = [40, 20, 10, 30] \]
\[ y = x.sort() \]

\[
\begin{array}{c|c}
0 & 40 \\
1 & 20 \\
2 & 10 \\
3 & 30 \\
\end{array}
\]
Void Functions

\[
x = [40, 20, 10, 30]
\]

\[
y = x.sort()
\]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
</tr>
</tbody>
</table>

Void Functions return None, a special type.
Void Functions

\[ x = [40, 20, 10, 30] \]

\[ y = \text{list}(x) \]
\[ y.\text{sort}() \]

Void Functions return None, a special type
Void Functions

\[
x = [40, 20, 10, 30]
y = list(x)
y.sort()
\]

Void Functions return None, a special type
def f(x):
    x = x[1:]
    print x

if __name__ == '__main__':
    u = [1,2,3,4]
    f(u)
    print u

Looks like f deletes the 0-th character in x
Understanding Function Calls

```python
def f(x):
    x = x[1:]
    print x

if __name__ blabla
    u = [1,2,3,4]
    f(u)
    print u
```

Follow the red dot and watch for impact…
def f(x):
    x = x[1:]
    print x

if __name__ blabla
    u = [1, 2, 3, 4]
    f(u)
    print u

Parameter x initially refers to the same object as u
def f(x):
    x = x[1:]
    print x

if __name__ == "__main__":
    u = [1,2,3,4]
    f(u)
    print u

x[1:] creates a new object and x will refer to it
Understanding Function Calls

def f(x):
    x = x[1:]
    print x

if __name__ blabla
    u = [1,2,3,4]
    f(u)
    print u

2 3 4 is printed
Understanding Function Calls

def f(x):
    x = x[1:]
    print x

if __name__ == 'blabla':
    u = [1,2,3,4]
    f(u)
    print u

1 2 3 4 is printed
Example: The Perfect Shuffle

Permuting the items in a list comes up a lot.

Here is a famous example called the perfect shuffle:

Before: 10 20 30 40 50 60 70 80

After: 10 50 20 60 30 70 40 80
Executing the Perfect Shuffle

The given list:

10 20 30 40 50 60 70 80

Cut it in half:

10 20 30 40

50 60 70 80

The Re-assemble Process:

Alternately choose from the “half” lists.
Executing the Perfect Shuffle

The given list:

10 20 30 40 50 60 70 80

Cut it in half:

10 20 30 40

50 60 70 80

The Re-assemble Process:

10

Alternately choose from the “half” lists.
Executing the Perfect Shuffle

The given list:

10 20 30 40 50 60 70 80

Cut it in half:

10 20 30 40 50 60 70 80

The Re-assemble Process:

10 50

Alternately choose from the “half” lists.
Executing the Perfect Shuffle

The given list:

10 20 30 40 50 60 70 80

Cut it in half:

10 20 30 40 50 60 70 80

The Re-assemble Process:

10 50 20

Alternately choose from the “half” lists.
Executing the Perfect Shuffle

The given list:

10 20 30 40 50 60 70 80

Cut it in half:

10 20 30 40
50 60 70 80

The Re-assemble Process:

10 50 20 60

Alternately choose from the “half” lists.
Executing the Perfect Shuffle

The given list:

10 20 30 40 50 60 70 80

Cut it in half:

10 20 30 40
50 60 70 80

The Re-assemble Process:

10 50 20 60 30

Alternately choose from the “half” lists.
Executing the Perfect Shuffle

The given list:

10 20 30 40 50 60 70 80

Cut it in half:

10 20 30 40 50 60 70 80

The Re-assemble Process:

10 50 20 60 30 70

Alternately choose from the “half” lists.
Executing the Perfect Shuffle

The given list:

```
10 20 30 40 50 60 70 80
```

Cut it in half:

```
10 20 30 40
50 60 70 80
```

The Re-assemble Process:

```
10 50 20 60 30 70 40
```
Executing the Perfect Shuffle

The given list:

\[10 \ 20 \ 30 \ 40 \ 50 \ 60 \ 70 \ 80\]

Cut it in half:

\[\begin{array}{c c c}
10 & 20 & 30 \\
40 & 50 & 60
\end{array}\]

\[\begin{array}{c c c}
50 & 60 & 70 \\
80 & & \\
\end{array}\]

The Re-assemble Process:

\[10 \ 50 \ 20 \ 60 \ 30 \ 70 \ 40 \ 80\]

Alternatively choose from the “half” lists.
def PF1(x):
    n = len(x)
    m = n/2
    top = list(x[:m])
    bot = list(x[m:]):
    for k in range(m):
        x[2*k] = top[k]
        x[2*k+1] = bot[k]

This is a Void function. It returns None. However, it permutes the values in the list referenced by x according to the perfect shuffle.
def PF2(x):
    n = len(x)
    m = n/2
    y = []
    for k in range(m):
        y.append(x[k])
        y.append(x[k+m])
    return y

This is a fruitful function. It returns a reference to a list that is the perfect shuffle of the list referenced by x.
Perfect Shuffle Cycles

Question:

Given a length-$n$ list $x$ where $n$ is even, how many perfect shuffle updates are required before we cycle back to the original $x$?
Perfect Shuffle Cycles

Solution Using the Void function PF1:

```python
# Assume x0 is a given list
x = list(x0)
PF1(x)
numPFs = 1
while x!=x0:
    PF1(x)
    numPFs+=1
print numPFs
```
Perfect Shuffle Cycles

Solution Using the Fruitful function PF2:

# Assume x0 is a given list
x = PF2(x0)
numPFs = 1
while x!=x0:
    x = PF2(x)
    numPFs+=1
print numPFs
Sample Outputs

<table>
<thead>
<tr>
<th>n</th>
<th>numPFs</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>8</td>
</tr>
<tr>
<td>444</td>
<td>442</td>
</tr>
<tr>
<td>1000</td>
<td>36</td>
</tr>
<tr>
<td>10000</td>
<td>300</td>
</tr>
<tr>
<td>100000</td>
<td>540</td>
</tr>
</tbody>
</table>