16. More On Lists

Topics:
- References
- Alias
- More on Slicing
- Merging Sorted Lists

Comparing Lists

You can use == to compare two lists

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

Comparing Lists

You can use == to compare two lists

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

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
```

Comparing Lists

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

Aliasing

This:
```
x = [10, 20, 30, 40]
y = x
```

Results in this:
```
x --> 0 --> 10
     1 --> 20
     2 --> 30
     3 --> 40
```

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

Unpredictable
**Aliasing**

$$x \rightarrow 0 \rightarrow 10$$
$$y \rightarrow 2 \rightarrow 30$$

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**

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

---

**Tracking Changes**

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

---

**The is Operator**

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

- $x$ -- $0 \rightarrow 10$
- $y$ -- $0 \rightarrow 10$

Even though the two lists have the same component values, $x$ and $y$ do not refer to the same object.

---

**The is Operator**

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

- $x$ -- $0 \rightarrow 10$
- $y$ -- $2 \rightarrow 30$

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

- \( x = [10, 20, 30, 40] \)
  - \( y = \text{list}(x) \)

Slices Create new Objects

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

Careful!

- \( x = [10, 20, 30, 40] \)
  - \( y = x \)
  - \( y = x[1:] \)
**Careful!**

\[
\begin{align*}
  x &= [10,20,30,40] \\
  y &= x \\
  y &= x[1:] \\
\end{align*}
\]

**Void Functions**

\[
\begin{align*}
  x &= [40,20,10,30] \\
  y &= x.sort() \\
  y &= \text{None} \\
\end{align*}
\]

**Void Functions**

\[
\begin{align*}
  x &= [40,20,10,30] \\
  y &= x.sort() \\
  y &= \text{None} \\
\end{align*}
\]

**Void Functions**

\[
\begin{align*}
  x &= [40,20,10,30] \\
  y &= list(x) \\
  y.sort() \\
  y &= \text{None} \\
\end{align*}
\]

**Understanding Function Calls**

```python
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...

Parameter x initially refers to the same object as u

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

1 2 3 4 is printed

Some Inadvertent Errors

```python
>>> x = [10,20,30]
>>> y = [11,21,31]
>>> z = x+y
>>> print z
[10,20,30,11,21,31]
```
Some Inadvertent Errors

```python
>>> x = [10,20,30]
>>> y = 3*x
>>> print y
[10,20,30,10,20,30,10,20,3]
```

List Comprehensions

```python
>>> x = [i for i in range(5)]
>>> print x
[0,1,2,3,4]
```

List Comprehensions

```python
>>> x = [1 for i in range(5)]
>>> print x
[1,1,1,1,1]
```

List Comprehensions

```python
>>> x = [math.sqrt(i) for i in range(5)]
>>> print x
[0,1,1.414,1.732,2.0]
```

Quickly: Lists of Strings

```python
>>> x = ['Maine','Vermont','New York']
>>> a = x[1]
>>> print a
'Vermont'
>>> c = a[2]
>>> print c
'r'
>>> x[1][2]
'r'
```

Next Problem

**Merging Two Sorted Arrays**

*Into a Single Sorted Array*
Example

x→12 33 35 45
y→15 42 55 65 75
z→12 15 33 35 42 45 55 65 75

x and y are input
They are sorted
z is the output

Merging Two Sorted Lists

x→12 33 35 45
y→15 42 55 65 75
z→[]

ix:0
iy:0

ix and iy keep track of where we are in x and y

Merge

x→12 33 35 45
y→15 42 55 65 75
z→12

ix:0
iy:0

Do we pick from x? x[ix] <= y[iy] ???

No. So update iy

Merge

x→12 33 35 45
y→15 42 55 65 75
z→12 15

ix:1
iy:0

Do we pick from x? x[ix] <= y[iy] ???

Yes. So update ix

ix:1
iy:0
Merge

\[ \begin{array}{c}
x \rightarrow 12 \ 33 \ 35 \ 45 \\
y \rightarrow 15 \ 42 \ 55 \ 65 \ 75 \\
z \rightarrow 12 \ 15
\end{array} \]

\text{Do we pick from } x? \quad x[ix] \leq y[iy] \quad ???

Yes. So update \( ix \)

Merge

\[ \begin{array}{c}
x \rightarrow 12 \ 33 \ 35 \ 45 \\
y \rightarrow 15 \ 42 \ 55 \ 65 \ 75 \\
z \rightarrow 12 \ 15 \ 33
\end{array} \]

\text{Do we pick from } x? \quad x[ix] \leq y[iy] \quad ???

Yes. So update \( ix \)

Merge

\[ \begin{array}{c}
x \rightarrow 12 \ 33 \ 35 \ 45 \\
y \rightarrow 15 \ 42 \ 55 \ 65 \ 75 \\
z \rightarrow 12 \ 15 \ 33 \ 35
\end{array} \]

\text{Do we pick from } x? \quad x[ix] \leq y[iy] \quad ???

No. So update \( iy \)...
Merge

x→12 33 35 45

y→15 42 55 65 75

z→12 15 33 35 42

Do we pick from x? x[ix] <= y[iy]? ??

Yes. So update ix.

Merge

x→12 33 35 45

y→15 42 55 65 75

z→12 15 33 35 42 45

Done with x. Pick from y

Merge

x→12 33 35 45

y→15 42 55 65 75

z→12 15 33 35 42 45

Done with x. Pick from y

Merge

x→12 33 35 45

y→15 42 55 65 75

z→12 15 33 35 42 45 55

So update iy

Merge

x→12 33 35 45

y→15 42 55 65 75

z→12 15 33 35 42 45 55 65

So update iy.
The Python Implementation...

```python
def Merge(x, y):
    n = len(x); m = len(y);
    ix = 0; iy = 0; z = []
    for iz in range(n+m):
        if ix>=n:
            z.append(y[iy]); iy+=1
        elif iy>=m:
            z.append(x[ix]); ix+=1
        elif x[ix] <= y[iy]:
            z.append(x[ix]); ix+=1
        else:
            z.append(y[iy]); iy+=1
    return z
```

x-list exhausted  y-list exhausted  x-value smaller  y-value smaller

x-list exhausted  y-list exhausted  x-value smaller  y-value smaller

4

ix: 4
iy: 4

Update iy
def Merge(x, y):
    u = list(x)  # Make copies of the incoming lists
    v = list(y)
    z = []
    while len(u) > 0 and len(v) > 0:
        if u[0] <= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
        z.extend(u)
        z.extend(v)
    return z

Implementation Using Pop

def Merge(x, y):
    u = list(x)  # Build z up via repeated appending
    v = list(y)
    z = []
    while len(u) > 0 and len(v) > 0:
        if u[0] <= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
        z.extend(u)
        z.extend(v)
    return z

Implementation Using Pop

def Merge(x, y):
    u = list(x)  # Every 'pop' reduces the length by 1. The loop shuts down when one of u or v is exhausted
    v = list(y)
    z = []
    while len(u) > 0 and len(v) > 0:
        if u[0] <= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
        z.extend(u)
        z.extend(v)
    return z

Implementation Using Pop

def Merge(x, y):
    u = list(x)  # Add what is left in u. OK if u is the empty list
    v = list(y)
    z = []
    while len(u) > 0 and len(v) > 0:
        if u[0] <= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
        z.extend(u)
        z.extend(v)
    return z

Implementation Using Pop

def Merge(x, y):
    u = list(x)  # Add what is left in v. OK if v is the empty list
    v = list(y)
    z = []
    while len(u) > 0 and len(v) > 0:
        if u[0] <= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
        z.extend(u)
        z.extend(v)
    return z

Implementation Using Pop