

## 15. Lists are Objects

### Topics:

- References
- Alias
- More on Slicing

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

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

```
>>> 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:

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

## Aliasing

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

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]   x --> 0 ----> 10
y = x                1 ----> 20
y = [1,2,3]          2 ----> 30
                    3 ----> 40
```

## Tracking Changes

```
x = [10,20,30,40]   x --> 0 ----> 10
● y = x             1 ----> 20
y = [1,2,3]        2 ----> 30
                  3 ----> 40
```

## Tracking Changes

```
x = [10,20,30,40]   x --> 0 ----> 10
y = x               1 ----> 20
● y = [1,2,3]      2 ----> 30
                  3 ----> 40
y --> 0 ----> 1
      1 ----> 2
      2 ----> 3
```

## The is Operator

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

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

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

x and y refer to the same object

## Making a Copy of a List

```
● x = [10,20,30,40]   x --> 0 ----> 10  
  y = list(x)        1 ----> 20  
                    2 ----> 30  
                    3 ----> 40
```

## Making a Copy of a List

```
● x = [10,20,30,40]   x --> 0 ----> 10  
  y = list(x)        1 ----> 20  
                    2 ----> 30  
                    3 ----> 40  
  
y --> 0 ----> 10  
      1 ----> 20  
      2 ----> 30  
      3 ----> 40
```

## Slices Create new Objects

```
● x = [10,20,30,40]   x --> 0 ----> 10  
  y = x[1:]          1 ----> 20  
                    2 ----> 30  
                    3 ----> 40
```

## Slices Create New Objects

```
● x = [10,20,30,40]   x --> 0 ----> 10  
  y = x[1:]          1 ----> 20  
                    2 ----> 30  
                    3 ----> 40  
  
y --> 0 ----> 20  
      1 ----> 30  
      2 ----> 40
```

## Careful!

```
● x = [10,20,30,40]   x --> 0 ----> 10  
  y = x               1 ----> 20  
  y = x[1:]          2 ----> 30  
                    3 ----> 40
```

## Careful!

```
● x = [10,20,30,40]   x --> 0 ----> 10  
  y = x               1 ----> 20  
  y = x[1:]          2 ----> 30  
                    3 ----> 40
```

## Careful!

```
x = [10,20,30,40]   x --> 0 ----> 10
y = x              1 ----> 20
● y = x[1:]        2 ----> 30
                  3 ----> 40

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

## Void Functions

```
● x = [40,20,10,30] x --> 0 ----> 40
y = x.sort()       1 ----> 20
                  2 ----> 10
                  3 ----> 30

y -->
```

## Void Functions

```
x = [40,20,10,30]   x --> 0 ----> 10
● y = x.sort()      1 ----> 20
                  2 ----> 30
                  3 ----> 40

y --> None
```

Void Functions return None, a special type

## Void Functions

```
x = [40,20,10,30]   x --> 0 ----> 40
● y = list(x)       1 ----> 20
y.sort()            2 ----> 10
                  3 ----> 30

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

Void Functions return None, a special type

## Void Functions

```
x = [40,20,10,30]   x --> 0 ----> 40
y = list(x)         1 ----> 20
● y.sort()          2 ----> 10
                  3 ----> 30

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

Void Functions return None, a special type

## Understanding Function Calls

```
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

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

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

```
u --> 0 ---> 1
      1 ---> 2
      2 ---> 3
      3 ---> 4
```

Follow the red dot and watch for impact...

## Understanding Function Calls

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

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

```
u --> 0 ---> 1
      1 ---> 2
      2 ---> 3
      3 ---> 4
```

```
x
```

Parameter x initially refers to the same object as u

## Understanding Function Calls

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

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

```
u --> 0 ---> 1
      1 ---> 2
      2 ---> 3
      3 ---> 4
```

```
x --> 0 ---> 2
      1 ---> 3
      2 ---> 4
```

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
```

```
u --> 0 ---> 1
      1 ---> 2
      2 ---> 3
      3 ---> 4
```

```
x --> 0 ---> 2
      1 ---> 3
      2 ---> 4
```

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
```

```
u --> 0 ---> 1
      1 ---> 2
      2 ---> 3
      3 ---> 4
```

```
x --> 0 ---> 2
      1 ---> 3
      2 ---> 4
```

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

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 80

Alternately  
choose  
from the  
"half" lists.

## Implementation 1

```
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]
```

Make a copy  
of the top and  
bottom halves

They become the  
even-indexed and  
odd-indexed entries

This is a Void function. It returns None.  
However, it permutes the values in the list referenced  
by x according to the perfect shuffle.

## Implementation 2

```
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
```

Build y up through  
repeated appending

x[k] comes from the  
top half of the list,  
x[k+m] comes from  
the bottom half.

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:

```
# 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

n	numPFs
8	3
52	8
444	442
1000	36
10000	300
100000	540