Lists in Python

The List is Python's most useful and flexible sequence type. A list is similar to a tuple, but it is mutable: Method calls and assignment statements can change the contents of a list.

We can introduce many list modification operations through an example that illustrates the history of playing cards (drastically simplified). Comments in the examples describe the effect of each method invocation.

Playing cards were invented in China, perhaps around the 9th century. An early deck had three suits, which corresponded to denominations of money.

```
>>> chinese_suits = ['coin', 'string', 'myriad']  # A
list literal
>>> suits = chinese_suits                         # Two
names refer to the same list
```

As cards migrated to Europe (perhaps through Egypt), only the suit of coins remained in Spanish decks (*oro*).

```
>>> suits.pop()             # Remove and return the final
element
'myriad'
>>> suits.remove('string')  # Remove the first element
that equals the argument
```

Three more suits were added (they evolved in name and design over time),

```
>>> suits.append('cup')              # Add an element to
the end
>>> suits.extend(['sword', 'club'])  # Add all elements
of a list to the end
```

and Italians called swords *spades*.

```
>>> suits[2] = 'spade'  # Replace an element
```

giving the suits of a traditional Italian deck of cards.

```
>>> suits
['coin', 'cup', 'spade', 'club']
```

The French variant that we use today in the U.S. changes the first two:

```
>>> suits[0:2] = ['heart', 'diamond']  # Replace a slice
>>> suits
['heart', 'diamond', 'spade', 'club']
```

Methods also exist for inserting, sorting, and reversing lists. All of these *mutation operations* change the value of the list; they do not create new list objects.

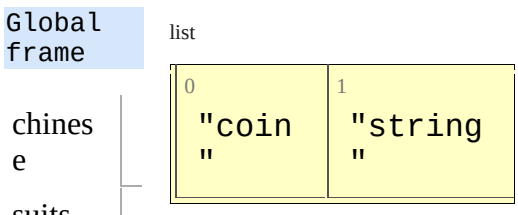
Sharing and Identity. Because we have been changing a single list rather than creating new lists, the object bound to the name `chinese_suits` has also changed, because it is the same list object that was bound to `suits`!

```
>>> chinese_suits  # This name co-refers with "suits" to the same list
['heart', 'diamond', 'spade', 'club']
```

This behavior is new. Previously, if a name did not appear in a statement, then its value would not be affected by that statement. With mutable data, methods called on one name can affect another name at the same time.

The environment diagram for this example shows how the value bound to `chinese` is changed by statements involving only `suits`. Step through each line of the following example to observe these changes.

```
1  chinese = ['coin', 'string', 'myriad']
2  suits = chinese
3  suits.pop()
4  suits.remove('string')
```



```
5  suits.append('cup')
6  suits.extend(['sword', 'club'])
7  suits[2] = 'spade'
8  suits[0:2] = ['heart', 'diamond']
```

Edit code

< Back Step 4 of 8 Forward >

Lists can be copied using the `list` constructor function. Changes to one list do not affect another, unless they share structure.

```
>>> nest = list(suits)  # Bind "nest" to a second list with the same elements
>>> nest[0] = suits     # Create a nested list
```

According to this environment, changing the list referenced by `suits` will affect the nested list that is the first element of `nest`, but not the other elements.

```
>>> suits.insert(2, 'Joker')  # Insert an element at index 2, shifting the rest
>>> nest
[['heart', 'diamond', 'Joker', 'spade', 'club'], 'diamond', 'spade', 'club']
```

And likewise, undoing this change in the first element of `nest` will change `suit` as well.

```
>>> nest[0].pop(2)
'Joker'
>>> suits
['heart', 'diamond', 'spade', 'club']
```

Stepping through this example line by line will show the representation of a nested list.

```
1  suits = ['heart', 'diamond', 'spade', 'club']
2  nest = list(suits)
```

```
3  nest[0] = suits
4  suits.insert(2, 'Joker')
5  j = nest[0].pop(2)
```

Edit code

< Back Step 1 of 5 Forward >

Because two lists may have the same contents but in fact be different lists, we require a means to test whether two objects are the same. Python includes two comparison operators, called `is` and `is not`, that test whether two expressions in fact evaluate to the identical object. Two objects are identical if they are equal in their current value, and any change to one will always be reflected in the other. Identity is a stronger condition than equality.

```
>>> suits is nest[0]
True
>>> suits is ['heart', 'diamond', 'spade', 'club']
False
>>> suits == ['heart', 'diamond', 'spade', 'club']
True
```

The final two comparisons illustrate the difference between `is` and `==`. The former checks for identity, while the latter checks for the equality of contents.

List comprehensions. A list comprehension uses an extended syntax for creating lists, analogous to the syntax of generator expressions.

For example, the `unicodedata` module tracks the official names of every character in the Unicode alphabet. We can look up the characters corresponding to names, including those for card suits.

```
>>> from unicodedata import lookup
>>> [lookup('WHITE ' + s.upper() + ' SUIT') for s in
suits]
['♡', '♢', '♤', '♧']
```

List comprehensions reinforce the paradigm of data processing using the conventional interface of sequences, as `list` is a sequence data type.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/objects.html#lists