```
def append_val(val, list_1 = []):
    List_1.append(val)
    print(list_1)

append_val(3)
append_val(4)
```

You might think that the second call to `append_val` would print the list `[4]` because it would have appended `4` to the empty list. In fact, it will print `[3, 4]`. This happens because, at function definition time, a new object of type `list` is created, with an initial value of the empty list. Each time `append_val` is invoked without supplying a value for the formal parameter `list_1`, the object created at function definition is bound to `list_1`, mutated, and then printed. So, the second call to `append_val` mutates and then prints a list that was already mutated by the first call to that function.

When we append one list to another, e.g., `Techs.append(Ivys)`, the original structure is maintained. The result is a list that contains a list. Suppose we do not want to maintain this structure, but want to add the elements of one list into another list. We can do that by using list concatenation (using the `+` operator) or the `extend` method, e.g.,

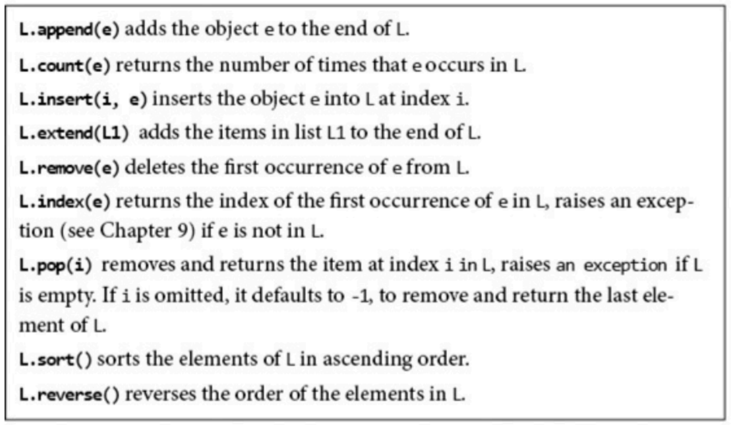
```
L1 = [1,2,3]
L2 = [4,5,6]
L3 = L1 + L2
print('L3 =', L3)
L1.extend(L2)
print('L1 =', L1)
L1.append(L2)
print('L1 =', L1)
```

will print

```
L3 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6, [4, 5, 6]]
```

Notice that the operator `+` does not have a side effect. It creates a new list and returns it. In contrast, `extend` and `append` each mutate `L1`.

Figure 5-4 briefly describes some of the methods associated with lists. Note that all of these except `count` and `index` mutate the list.



`L.append(e)` adds the object `e` to the end of `L`.

`L.count(e)` returns the number of times that `e` occurs in `L`.

`L.insert(i, e)` inserts the object `e` into `L` at index `i`.

`L.extend(L1)` adds the items in list `L1` to the end of `L`.

`L.remove(e)` deletes the first occurrence of `e` from `L`.

`L.index(e)` returns the index of the first occurrence of `e` in `L`, raises an exception (see Chapter 9) if `e` is not in `L`.

`L.pop(i)` removes and returns the item at index `i` in `L`, raises an exception if `L` is empty. If `i` is omitted, it defaults to `-1`, to remove and return the last element of `L`.

`L.sort()` sorts the elements of `L` in ascending order.

`L.reverse()` reverses the order of the elements in `L`.

Figure 5-4 Common methods associated with lists

5.3.1 Cloning

It is usually prudent to avoid mutating a list over which one is iterating. Consider the code

```
def remove_dups(L1, L2):
    """Assumes that L1 and L2 are lists.
       Removes any element from L1 that also occurs in L2"""
    for e1 in L1:
        if e1 in L2:
            L1.remove(e1)
L1 = [1,2,3,4]
L2 = [1,2,5,6]
Remove_dups(L1, L2)
print('L1 =', L1)
```

You might be surprised to discover that this prints

```
L1 = [2, 3, 4]
```

During a `for` loop, Python keeps track of where it is in the list using an internal counter that is incremented at the end of each iteration. When the value of the counter reaches the current length of the list, the loop terminates. This works as you might expect if the list is not mutated within the loop, but can have surprising consequences if the list is mutated. In this case, the hidden counter starts out at `0`, discovers that `L1[0]` is in `L2`, and removes it—reducing the length of `L1` to `3`. The counter is then incremented to `1`, and the code proceeds to check if the value of `L1[1]` is in `L2`. Notice that this is not the original value of `L1[1]` (i.e., `2`), but rather the current value of `L1[1]` (i.e., `3`). As you can see, it is possible to figure out what happens when the list is modified within the loop. However, it is not easy. And what happens is likely to be unintentional, as in this example.

One way to avoid this kind of problem is to use slicing to **clone**[35] (i.e., make a copy of) the list and write `for e1 in L1[:]`. Notice that writing

```
new_L1 = L1
for e1 in new_L1:
```

would not solve the problem. It would not create a copy of `L1`, but would merely introduce a new name for the existing list.

Slicing is not the only way to clone lists in Python. The expression `L.copy()` has the same value as `L[:]`. Both slicing and `copy` perform what is known as a **shallow copy**. A shallow copy creates a new list and then inserts the objects (not copies of the objects) of the list to be copied into the new list. The code

```
L = [2]
L1 = [L]
L2 = L1[:]
L2 = copy.deepcopy(L1)
L.append(3)
print(f'L1 = {L1}, L2 = {L2}')
```

prints `L1 = [[2, 3]] L2 = [[2, 3]]` because both `L1` and `L2` contain the object that was bound to `L` in the first assignment statement.

If the list to be copied contains mutable objects that you also want to copy, import the standard library module `copy` and use the function `copy.deepcopy` to make a **deep copy**. The method `deepcopy` creates a new list and then inserts copies of the objects in the list to be copied into the new list. If we replace the third line in the above code by `L2 = copy.deepcopy(L1)`, it will print `L1 = [[2, 3]], L2 = [[2]]`, because `L1` would not contain the object to which `L` is bound.

Understanding `copy.deepcopy` is tricky if the elements of a list are lists containing lists (or any mutable type). Consider

```
L1 = [2]
L2 = [[L1]]
L3 = copy.deepcopy(L2)
L1.append(3)
```

The value of `L3` will be `[[[2]]]` because `copy.deepcopy` creates a new object not only for the list `[L1]`, but also for the list `L1`. I.e., it makes copies all the way to the bottom—most of the time. Why "most of the time?" The code

```
L1 = [2]
L1.append(L1)
```

creates a list that contains itself. An attempt to make copies all the way to the bottom would never terminate. To avoid this problem, `copy.deepcopy` makes exactly one copy of each object, and then uses that copy for each instance of the object. This matters even when lists do not contain themselves. For example,

```
L1 = [2]
L2 = [L1, L1]
L3 = copy.deepcopy(L2)
L3[0].append(3)
print(L3)
```

prints `[[2, 3], [2, 3]]` because `copy.deepcopy` makes one copy of `L1` and uses it both times `L1` occurs in `L2`.

5.3.2 List Comprehension

List comprehension provides a concise way to apply an operation to the sequence values provided by iterating over an iterable value. It creates a new list in which each element is the result of applying a given operation to a value from an iterable (e.g., the elements in another list). It is an expression of the form

```
[expr for elem in iterable if test]
```

Evaluating the expression is equivalent to invoking the function