Lists as a mutable type
The append and extend methods on lists

- Suppose we want to add an element 10 to the back of a list \( L \). Using what we have learned, we would use the code
  \[
  L = L + [10]
  \]
to do this.

- There is a more convenient and efficient way of accomplishing this:
  \[
  L.append(10)
  \]

- Example:
  ```python
  >>> L = [1, 25, "hello", -67]
  >>> L.append(25)
  >>> L [1, 25, 'hello', -67, 25]
  >>> L.extend([-1, -2])
  >>> L [1, 25, 'hello', -67, 25, -1, -2]
  ```
Differences between “+” and `append`, `extend` 

- Say `L = [1, 2, 3]`. 
- `L.append(17)` and `L.extend([12, 15])` are examples of *in-place* list operations. 
- These operations modify the list `L` onto which they are applied. They do not create a new list. 
- In this sense, `L.append(17)` and `L + [17]` are very different from each other. 
- `L + [17]` does not modify `L` and it evaluates to `[1, 2, 3, 17]`. 
- *Strings do not support any in-place operations*. You cannot modify a string – you have to create a new string.
Suppose $s = \text{“hello”}$

The \texttt{s.append(“hi”)} produces an error message.

For $s$ to take on value \texttt{“hellohi”} we have to use

\texttt{s = s + “hi”}
Lists support other in-place operations

- In addition to **append** and **extend**:
  - \( L[3] = 22 \)
    This assigns 22 to the slot in \( L \) indexed by 3. The previous value of \( L[3] \) is replaced by 22. \( L \) does not change in size.

  - \( L.insert(3, 22) \)
    This inserts 22 into slot in \( L \) indexed by 3, moving. Elements previously indexed 3, 4, 5, etc. are all moved to the right and have higher indices now.

**Example:**

\[
L = [0, 1, 2, 3, 4, 5, 6]
\]

\[
L.insert(3, 22)
\]

\[
L = [0, 1, 2, 22, 3, 4, 5, 6]
\]
Lists supports other in-place operations

Try these operations:

- **L.remove(22)**
  - Removes first occurrence of 22 from L. Elements that come after 22 are moved to the left. Length of L decreases by 1.
  - Causes an error if 22 is not in list; so the programmer has to be sure of this before using `remove`.

- **L.sort()**
- **L.reverse()**

Look at Python documentation: Section 5.6.4 on Mutable Sequence Types.
Mutable types

- Lists can support in-place operations and types of this sort in Python are called *mutable types*.
- None of the types we have encountered so far: *int*, *float*, *bool*, *string* are mutable.
- There are fundamental differences in behind-the-scenes implementation between Lists and these other types.
- These differences are important to learn about because they manifest themselves in many different settings.
Behind the Scenes

The difference between objects of type list and objects of other types is due to an important difference in implementation.

Consider the assignment: \( L = [3, 4, 5] \)

We might think that after this assignment, \( L \) is a “sticky note” onto the list \([3, 4, 5]\).

But no! \( L \) is a “sticky note” onto something that in turn points to \([3, 4, 5]\).

In programming language terminology, we say \( L \) is a “sticky note” to a reference to \([3, 4, 5]\).
L (a sticky note) → Reference (pointer) (address) → [2, 3, 4]
Consider the example:

L = [3, 4, 5]
LL = L
L.append(6)
LL
[3, 4, 5, 6]

Notice how when we modified L, the list LL also changed. This is not true for any of the data types we have seen so far.

After the assignment LL = L, LL is a “sticky note” to a reference that also points to the same exact list as L.
L → Reference 1 → [3, 4, 5] → LL → Reference 2
Another Example

$L = [3, 4, 5]$
$LCopy = L$
$M = [3, 4, 5]$

$L == LCopy, LCopy == M, M == L$
(True, True, True)

$L[0] = 9$
$L == LCopy, LCopy == M, M == L$
(True, False, False)
def test(L):
    x = L[0] + L[1] + L[2]
    L.append(10)
    return x

Now consider what happens when this function is called:
    M = [1, 2, 3, 4]
    test(M)
    6
    M
    [1, 2, 3, 4, 10]

This is a side-effect of the in-place operation L.append(10) performed inside the function.