Lists as a Mutable Data Type
The swap function

- Consider the following “integer swap” function:

```python
def swapInts(a, b):
    temp = a
    a = b
    b = temp
```

- Let us call this function as follows:

```python
x = 5
y = 10
swapInts(x, y)
```

- What are values of variables `x` and `y` now?
This is not unexpected!

- The fact that $x$ and $y$ remain unchanged is not unexpected.
- Recall that when the function `swapInts` is called, the parameter $a$ is a local variable that takes the value of $x$ (which is 5).
- Similarly, the parameter $b$ is a local variable that takes on the value of $y$ (which is 10).
- The variables $a$ and $b$ are swapped in `swapInts`.
- However, nothing happens to $x$ and $y$ since these and the variables $a$ and $b$ are distinct.
Let us now try swapping string elements

Consider the code for `swap` that was part of `selectionSort`:

```python
def swap(L, i, j):
    temp = L[i]
    L[i] = L[j]
    L[j] = temp
```

What happens when we call it as follows?

```python
s = "hello"
swap(s, 1, 2)
```
This is a key difference between strings and lists

- Both lists and strings allow the *access* of elements via an index. In other words, we can look at \( L[i] \) or \( s[i] \).
- However, we can *assign* to list elements via an index, but not to string elements.
- **Example:**
  
  ```python
  s = "hello"
  s[2] = "p"
  ```
  produces an error saying *str* object cannot support assignment.
In-place operations

- Say \( \text{L} = [1, 2, 3] \).
- \( \text{L}[2] = 10 \) and \( \text{L}.\text{append}(17) \) are examples of *in-place* list operations.
- These operations modify the list \( \text{L} \) onto which they are applied. They do not create a new list.
- In this sense, \( \text{L}.\text{append}(17) \) and \( \text{L} + [17] \) are very different from each other.
- \( \text{L} + [17] \) does not modify \( \text{L} \) 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.
Lists support many other in-place operations

- Try these operations!
  - `L.append(10)`
  - `L.extend([1, 2, 3])`
  - `L.insert(2, "hello")`
  - `L.remove("hello")`
  - `L.sort()`
  - `L.reverse()`

- None of these work on strings.

- Look at Section 5.6.4 on “Mutable Sequence Types” in Python v.2.7.3 documentation.
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, $L$ points to the list $\[3, 4, 5\]$. But no! $L$ points to something that in turn points to $\[3, 4, 5\]$.

In programming language terminology, we say $L$ points to a *reference* to $\[3, 4, 5\]$. 
L (is a sticky note) → Reference (address) (pointer) → [3, 4, 5]

Indirection
Consider the example:
```python
>>> L = [3,4,5]
>>> LL = L
>>> L.append(6)
>>> LL [3, 4, 5, 6]
```

Notice how when 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 points to a reference that points to the same list as L.
Another Example using List Assignment

```python
>>> 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):
    L[0] = 7
    return sum(L)

# main program
J = [3, 4, 5]
print test(J)
print J

• When you run this and print J, you will see that J has become [7, 4, 5].

• When J is sent in as argument to test, L is given a copy of J.

• But, since J is pointing to a reference to a list, L ends up pointing to a copy of the reference, but to the same physical list.

• This provides another way of communicating between a main program and functions (and between functions).