Programming with Python

Storing Multiple Values in Lists

Learning Objectives

- Explain what a list is.
- Create and index lists of simple values.

Just as a `for` loop is a way to do operations many times, a list is a way to store many values. Unlike NumPy arrays, lists are built into the language (so we don't have to load a library to use them). We create a list by putting values inside square brackets:

```
odds = [1, 3, 5, 7]
print('odds are:', odds)
```

```
odds are: [1, 3, 5, 7]
```

We select individual elements from lists by indexing them:

```
print('first and last:', odds[0], odds[-1])
```

```
first and last: 1 7
```

and if we loop over a list, the loop variable is assigned elements one at a time:

```
for number in odds:
    print(number)
```

```
1
3
5
7
```

There is one important difference between lists and strings: we can change the values in a list, but we cannot change the characters in a string. For example:

```
names = ['Newton', 'Darwing', 'Turing'] # typo in Darwin's name
print('names is originally:', names)
names[1] = 'Darwin' # correct the name
print('final value of names:', names)
```

```
names is originally: ['Newton', 'Darwing', 'Turing']
final value of names: ['Newton', 'Darwin', 'Turing']
```

works, but:

```
name = 'Bell'
name[0] = 'b'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-220df48aeb2e> in <module>()
      1 name = 'Bell'
----> 2 name[0] = 'b'

TypeError: 'str' object does not support item assignment
```

does not.

Ch-Ch-Ch-Changes

Data which can be modified in place is called mutable, while data which cannot be modified is called immutable. Strings and numbers are immutable. This does not mean that variables with string or number values are constants, but when we want to change the value of a string or number variable, we can only replace the old value with a completely new value.

Lists and arrays, on the other hand, are mutable: we can modify them after they have been created. We can change individual elements, append new elements, or reorder the whole list. For some operations, like sorting, we can choose whether to use a function that modifies the data in place or a function that returns a modified copy and leaves the original unchanged.

Be careful when modifying data in place. If two variables refer to the same list, and you modify the list value, it will change for both variables! If you want variables with mutable values to be independent, you must make a copy of the value when you assign it.

Because of pitfalls like this, code which modifies data in place can be more difficult to understand. However, it is often far more efficient to modify a large data structure in place than to create a modified copy for every small change. You should consider both of these aspects when writing your code.

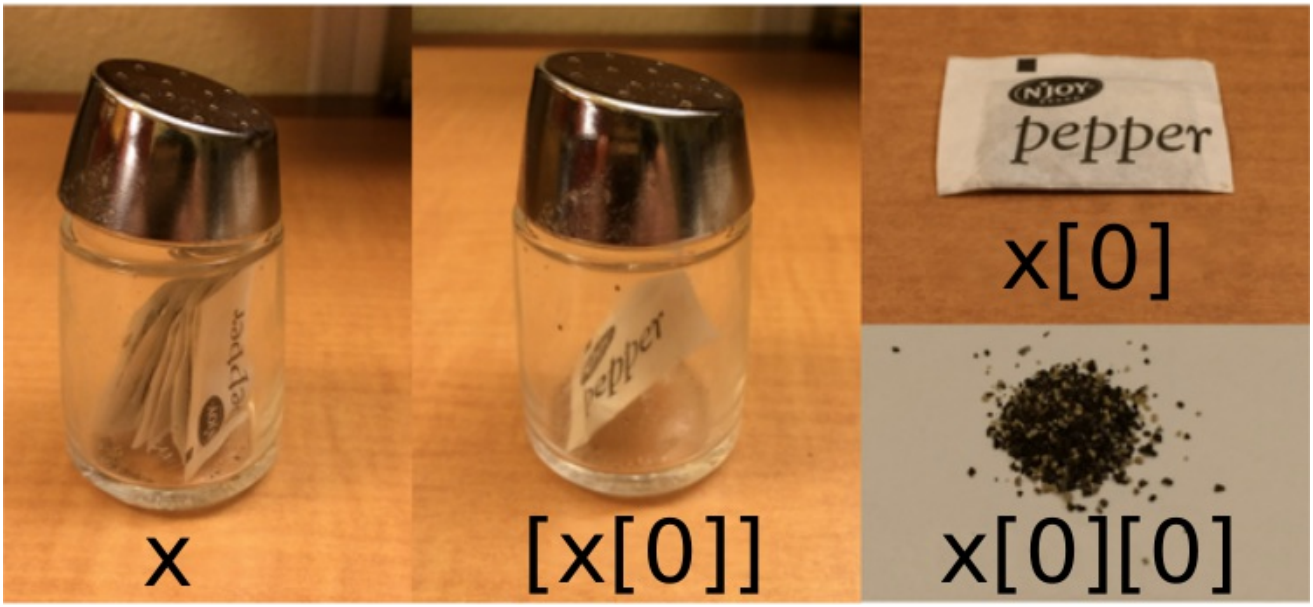
Nested Lists

Since lists can contain any Python variable, it can even contain other lists.

For example, we could represent the products in the shelves of a small grocery shop:

```
x = [['pepper', 'zucchini', 'onion'],
     ['cabbage', 'lettuce', 'garlic'],
     ['apple', 'pear', 'banana']]
```

Here is a visual example of how indexing a list of lists `x` works:



Using the previously declared list `x`, these would be the results of the index operations shown in the image:

```
print([x[0]])
```

```
[['pepper', 'zucchini', 'onion']]
```

```
print(x[0])
```

```
['pepper', 'zucchini', 'onion']
```

```
print(x[0][0])
```

```
'pepper'
```

Thanks to Hadley Wickham for the image above.

There are many ways to change the contents of lists besides assigning new values to individual elements:

```
odds.append(11)
print('odds after adding a value:', odds)
```

```
odds after adding a value: [1, 3, 5, 7, 11]
```

```
del odds[0]
print('odds after removing the first element:', odds)
```

```
odds after removing the first element: [3, 5, 7, 11]
```

```
odds.reverse()
print('odds after reversing:', odds)
```

```
odds after reversing: [11, 7, 5, 3]
```

While modifying in place, it is useful to remember that Python treats lists in a slightly counterintuitive way.

If we make a list and (attempt to) copy it then modify in place, we can cause all sorts of trouble:

```
odds = [1, 3, 5, 7]
primes = odds
primes += [2]
print('primes:', primes)
print('odds:', odds)
```

```
primes: [1, 3, 5, 7, 2]
odds: [1, 3, 5, 7, 2]
```

This is because Python stores a list in memory, and then can use multiple names to refer to the same list. If all we want to do is copy a (simple) list, we can use the `list` function, so we do not modify a list we did not mean to:

```
odds = [1, 3, 5, 7]
primes = list(odds)
primes += [2]
print('primes:', primes)
print('odds:', odds)
```

```
primes: [1, 3, 5, 7, 2]
odds: [1, 3, 5, 7]
```

This is different from how variables worked in lesson 1, and more similar to how a spreadsheet works.

Turn a string into a list

Use a for-loop to convert the string “hello” into a list of letters:

```
["h", "e", "l", "l", "o"]
```

Hint: You can create an empty list like this:

```
my_list = []
```

Tuples and exchanges

Explain what the overall effect of this code is:

```
left = 'L'
right = 'R'

temp = left
left = right
right = temp
```

Compare it to:

```
left, right = right, left
```

Do they always do the same thing? Which do you find easier to read?