Lists

Sequential containers store data items in a specified order. Think elements of a vector, names in a list of people that you want to invite to your birthday party. The most fundamental Python data type for this is called a `list`. Later in the course we will learn about containers that are more appropriate (and faster) for numerical data that come from NumPy.

Creating lists

In Python, lists are created by putting things inside of square brackets (`[]`).

```
some_primes = [1,2,3,5,7,11]
print(some_primes)
```

Lists can hold objects of any type:

```
many_types = [1, 55.5, "Am I in a list?", True]
print(many_types)
```

We can get the length of the list with the `len()` functions:

```
len(many_types)
```

Accessing Elements

Elements of a list are accessed using square brackets after a variable.

```
myList = [5, 2.3, 'hello']
```

The first element of a list is at index `0`:

```
myList[0]
```

```
myList[2]
```

Attempting to access an element out of bounds will produce an error:

```
myList[3]
```

We can index to `-1` to get the object at the end of the list:

```
myList[-1]
```

Likewise, we can index backwards using negative numbers:

```
myList[-3]
```

Slicing

A sub-list may be accessed using slice syntax. Let's start with the list:

```
many_types = [1, 55.5, "Am I in a list?", True, "the end"]
```

Let's look at a sub-list:

```
many_types[2:4]
```

The `[2:4]` is called a slice and returns a list with elements at indices 2 and 3 from the original list.

It is easy to slice from an index to the end:

```
many_types[2:]
```

It is also easy to slice from the beginning to a specified index:

```
many_types[:3]
```

Adding and multiplying

The + operator concatenates (or combines) lists:

```
myList = [5, 2.3, 'hello']
mySecondList = ['a', '3']
concatList = myList + mySecondList
print(concatList)
```

The * operator can be used to repeat lists:

```
myList = ['hello', 'world']
print(myList * 2)
print(2 * myList)
```

Lists are mutable

Lists are mutable, this means that individual elements can be changed.

```
# start with a list
my_list = ['a', 43, 1.234]
# assign a new value to index 0
my_list[0] = -3
# inspect the list
print(my_list)
```

We can also assign to a slice

```
x = 2
my_list[1:3] = [x, 2.3]
print(my_list)
```

Copying a list

Let's attempt to copy a list referenced by variable `a` to another variable `b`:

```
a = ['a', 'b', 'c']
b = a # attempt to copy a to b
b[1] = 1 # now we want to change an element in b
print(b)
print(a)
```

That's interesting! Modifying `b` caused `a` to change.

Let's look at this example in Python Tutor.

Ok, let's try a different technique.

```
a = ['a', 'b', 'c']
b = a       # first attempt to copy a to b
c = list(a) # use the list constructor
b[1] = 1    # now we want to change an element in b
print("a: ", a)
print("b: ", b)
```

```
print("c: ", c)
print("id(a): ", id(a))
print("id(b): ", id(b))
print("id(c): ", id(c))
```

Again, let's try this example in Python Tutor.

A list can be copied with `b = list(a)` or `b = a[:]`. The second option is a slice including all elements.

Python's data model

Variables in Python are actually a reference to an object in memory. Assignment with the `=` operator sets the variable to refer to an object. Here is a simple example to demonstrate this property:

```
a = [1,2,3,4]
b = a
b[1] = 55
print(b)
print(a)
```

In this example, we assigned `a` to `b` via `b = a`. This did not copy the data, it only copied the reference to the list object in memory. Thus modifying the list through `b` also changes the data that you will see when accessing from `a`. You can inspect object ids in Python with:

```
print("id(a): ", id(a))
print("id(b): ", id(b))
```

Those numbers refer to memory addresses.

Copying data (more generally)

The `copy` function in the `copy` module is a more generic way to copy a list:

```
import copy
a = [5,2,7,0,'abc']
b = copy.copy(a)
b[4] = 'xyz'
print(b)
print(a)
```

Note that elements in a list are also references to memory location. For example if your list contains a list, this will happen when using `copy.copy()`:

```
a = [2, 'string', [1,2,3]]
b = copy.copy(a)
b[2][0] = 55
print(b)
print(a)
```

Here, the element for the sub-list `[55, 2, 3]` is actually a memory reference. So, when we copy the outer list, only references for the contained objects are copied. Thus in this case modifying the copy (`b`) modifies the original (`a`). Thus, we may need the function `copy.deepcopy()`:

```
a = [2, 'string', [1,2,3]]
b = copy.deepcopy(a)
b[2][0] = 99
print(b)
print(a)
```

Sorting lists

Sorting Python lists is very easy. Let's randomly shuffle a list and then sort it.

```
import random
my_list = list(range(10))
print(my_list)
random.shuffle(my_list)
print(my_list)
```

Note that the `random.shuffle()` function shuffles the list in-place. It does not create a new list.

We can use the `sorted()` function to return a new sorted list:

```
sorted_list = sorted(my_list)
print("my_list:", my_list)
print("sorted_list:", sorted_list)
```

The list `sort()` method sorts the list in place:

```
my_list.sort()
print("my_list:", my_list)
```

List operations

In the following summary, `s` is a list and `x` is an element.

- `x in s`: `True` if an item of `s` is equal to `x`, else `False`
- `x not in s`: `False` if an item of `s` is equal to `x`, else `True`
- `s + t`: the concatenation of `s` and `t`
- `s * n` or `n * s`: equivalent to adding `s` to itself `n` times
- `s[i]`: `i`th item of `s`, origin `0`
- `s[i:j]`: slice of `s` from `i` to `j`
- `s[i:j:k]`: slice of s from `i` to `j` with step `k`
- `len(s)`: length of `s`
- `min(s)`: smallest item of `s`
- `max(s)`: largest item of `s`
- `s.index(x)`: index of the first occurrence of `x` in `s`
- `s.count(x)`: total number of occurrences of `x` in `s`
- `s[i] = x`: item `i` of `s` is replaced by `x`
- `s[i:j] = t`: slice of `s` from `i` to `j` is replaced by the contents of the `t`
- `del s[i:j]`: same as `s[i:j] = []`
- `s[i:j:k] = t`: the elements of `s[i:j:k]` are replaced by those of `t`
- `del s[i:j:k]`: removes the elements of `s[i:j:k]` from the list
- `s.append(x)`: appends `x` to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
- `s.clear()`: removes all items from `s` (same as `del s[:]`)
- `s.copy()`: creates a shallow copy of `s` (same as `s[:]`)
- `s.extend(t) or s += t`: extends `s` with the contents of `t` (for the most part the same as `s[len(s):len(s)] = t`)
- `s *= n`: updates `s` with its contents repeated `n` times
- `s.insert(i, x)`: inserts `x` into `s` at the index given by `i` (same as `s[i:i] = [x]`)
- `s.pop([i])`: retrieves the item at `i` and also removes it from `s`
- `s.remove(x)`: remove the first item from `s` where `s[i] == x`
- `s.reverse()`: reverses the items of `s` in place
- `s.sort()`: sorts the items of `s` in place

Also have a look at `help(list)`.

Resources

- Python tutorial section on `list`
- Python reference section on sequences