Lists

Lists are quite literally lists of items. Think about all the lists you might have used before such as a to-do list and a shopping list. A list in Python simply contains items. These items can be pretty much anything that has some sort of value attached to them. You can store raw values, variables, or both, and you can mix-and-match types, which means your list can contain multiple types of values at any given time.

The syntax for a list is just square brackets with every item separated by a comma:

In [ ]:
```
# Example 7-0: List Syntax

example_list = [1, 2, "Hi"]
```

Lists have quite a few things you can do to them which makes them very powerful data structures to use, which include but are not limited to:

- Add (append) items
- Remove items
- Access specific items
- Merge lists
- Find their lengths
- Loop through lists

These seven are probably the most common list operations in ENGR 102. They will also be used when coding in general, not just this course. As I have said, lists are extremely powerful, so this module will be extremely important to understand and learn well in order to get good grades in the course and to code well should you continue coding past this course.

Accessing Specific Items

Items in a list do have an order. Their location is determined by something called an "index", similar to the index section of a textbook. The way it is generally done in Python (and most other programming languages) is what is called zero-indexing: the first item is at location zero, the second is at location one, etc. The way you retrieve an item at a certain index is with square brackets.

In [ ]:
```
# Example 7-1: Indexing a list (valid and invalid)

names = ["Anthony", "Sam"]

print(names[0])
print(names[2])
```

```
Anthony
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[12], line 6
      3 names = ["Anthony", "Sam"]
      5 print(names[0])
----> 6 print(names[2])

IndexError: list index out of range
```

As you can see, `names` has two items (`"Anthony"` and `"Sam"`) at indices zero and one respectively. Accessing an index that does not exist will cause an error as seen in Example 7-1.

Modifying a List

Lists are what we call mutable: you can directly change their value. Values like ints and strings are immutable: you can't directly change the value of an int or a string, you can only assign a new int or string to the variable holding the value. The most obvious way to modify a list is to append (add) or remove items from a list.

Appending to a list requires using `.append(item)` . `item` can be anything you want. If it can be stored in a variable, it can be stored in a list.

In [ ]:
```
# Example 7-2: Appending to a list

names = ["Anthony"]
names.append("Sam")

print(names)
```

```
['Anthony', 'Sam']
```

There is an important things to notice about Example 2. One is that there is no variable assignment. I did not write `names = names.append("Sam")` . There is no need to do this because lists are mutable: just calling the `append()` function correctly will add the name `"Sam"` to the list for me. Once we learn more about functions, you will realize that trying to assign the result of `append()` to the variable will actually *remove* the list.

In [ ]:
```
# Example 7-3: Variable Assignment

names = ["Anthony"]
names = names.append("Sam")

print(names)
```

```
None
```

As seen above, `names` is now `None`, which is the data type for nothing. Remember that assignments will take the value of the right side and put it into the variable on the left side of the assignment operator. `.append()` doesn't give you a value which is why the resulting output is `None`. Do note that not all changes to a variable require an assignment: sometimes you can just directly change the variable if the data type is mutable. There's no trick to helping you with this. You just need to know what the operation you're doing actually does and whether it gives you a new value (needs an assignment) or directly modifies the value of your variable (does not want an assignment).

Merging lists is as simple as literally adding lists together. As this is addition, the newly created list needs to be stored somewhere.

In [ ]:
```
# Example 7-4: Adding lists

names = ["Anthony"]
more_names = ["Sam"]

all_names = names + names + more_names

print(all_names)
```

```
['Anthony', 'Anthony', 'Sam']
```

You add two lists and it creates a larger list. As I've done with Example 4, you can add more than two at a time as long they're all lists. Python gets very unhappy if they are not all of the same type (which is typical behavior).

You can also insert into an index using `.insert(index, item)`.

In [ ]:
```
# Example 7-5: Inserting into a list

names = ["Anthony", "Sam"]
names.insert(2, "Luke")
names.insert(0, "Luna")

print(names)
```

```
['Luna', 'Anthony', 'Sam', 'Luke']
```

Although there is no value at index two, Python understands that you want to insert the value to the end of the list (as index two comes right after index one, which was the current end index at the time). Also note that all affected elements move to the right (and never to the left).

Removing from a list is weird though. There's actually three different ways to remove an item from a list and they all have different behaviors:

- `.remove(item)`: directly removes `item` regardless of where it is
- `del list[index]`: removes the item from `list` at index `index`
- `.pop(index)`: removes the item from the list at index `index`, defaults to last item if `index` is not given; also returns the item removed

All three will get the job done, but choosing which one depends on what information you have. If you know exactly what item you want to remove, you should use `.remove()`. If you only know where the item is (you only know the index), then use `del`. However, if you'd like to save the removed item for further processing, use `.pop()`. If you just want to remove the last item in the list, use `.pop()`.

In [ ]:
```
# Example 7-6: Using .remove()

names = ["Anthony", "Sam"]
names.remove("Sam")

print(names)
```

```
['Anthony']
```

In [ ]:
```
# Example 7-7: Using del

names = ["Anthony", "Sam"]
del names[1]

print(names)
```

```
['Anthony']
```

In [ ]:
```python
# Example 7-8: Using .pop()

names = ["Anthony", "Sam", "Luke"]

last_name = names.pop()
first_name = names.pop(0)

print(names)
print("Last name in the list:", last_name)
print("First name in the list:", first_name)
```

```
['Sam']
Last name in the list: Luke
First name in the list: Anthony
```

Built-in Functions

There are quite the number of built-in functions that you can use for lists. The functions I believe you will have the most use of in this course are the following:

- `len()`
- `max()` and `min()`
- `sum()`

Don't worry, these functions do exactly as they seem: `len()` gives you the length of the list, `max()` gives you the maximum value in the list, `min()` gives you the minimum value in the list, and `sum()` gives you the sum of all the items in the list.

In [ ]:
```python
# Example 7-9: Using the built-in functions

numbers = [1, 5, 7, 4]

print(len(numbers), max(numbers), min(numbers), sum(numbers))
```

```
4 7 1 17
```

Looping through a List

There are actually two ways to loop through a list: using `range()` and using the list itself. Using `range()` is probably the easiest way, going through index zero to the very last valid index. The easiest way to do this is to use `range(len(list))` because `len(list)` will be excluded and the last index of any non-empty list is the length minus one.

In [ ]:
```python
# Example 7-10: Looping through a list with range()

numbers = [5, 6, 7, 8]

for i in range(len(numbers)):
    print(numbers[i])
```

```
5
6
7
8
```

In Example 7-10, you can see that `i` holds in the indicies and the loop goes through the indicies and prints out each number in order. You can also just directly loop through the list though.

In [ ]:
```python
# Example 7-11: Looping through the list directly

numbers = [5, 6, 7, 8]

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

```
5
6
7
8
```

Now with Example 7-11, you can see that `number` actually holds the contents of the list, and the loop goes from left to right of the list. If you don't care about the index, then directly loop through the list. However, if the index is useful for whatever reason, then use `range()`.

Replacing Items

Instead of modifying the list by adding or removing, you can just replace items in the list. You can treat each *existing* index as a variable and just assign a new value.

In [ ]:
```python
# Example 7-12: Replacing an item
```

```
names = ["Anthony", "Sam"]
names[0] = "Luke"

print(names)
```

```
['Luke', 'Sam']
```

Now like I said earlier, it must be a pre-existing index. In other words, you must be replacing a value at its index, not just inserting a new element. If you try to assign to an index that has no value, you will get an error.

In [ ]:

```
# Example 7-13: Replacing... nothing?

names = ["Anthony", "Sam"]
names[2] = "Luke"

print(names)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[2], line 4
      1 # Example 12: Replacing... nothing?
      3 names = ["Anthony", "Sam"]
----> 4 names[2] = "Luke"
      6 print(names)

IndexError: list assignment index out of range
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js