Lists and Tuples

What we will cover...

1. List operations
2. Tuples
3. Destructuring

List indexing

Sometimes we want to get retrieve a single element from a list.

We can do this if we know the **index** of the element.

Python indices start from `0`!

We can also use negative indexing to index from the end of the list.

```python
my_list = ['foo', 'bar', 'baz']

my_list[0] # 'foo'
my_list[1] # 'bar'
my_list[2] # 'baz'

my_list[-1] # 'baz'
my_list[-2] # 'bar'
```

List indexing

Instead of selecting a single element from a list, you can select a **contiguous range** of elements.

This is done with the followign form `[start:end]`.

If `start` is omitted, the range starts at the first element. If `end` is omitted, the range goes until the last element.

`start` is inclusive. `end` is exclusive.

```
my_list = ['foo', 'bar', 'baz', 'qux']

my_list[1:] # ['bar', 'baz', 'qux]
my_list[:1] # ['foo']
my_list[1:2] # ['bar']
my_list[1:3] # ['bar', 'baz]
```

List indexing

We can also mutate a list inplace.

For example, we can change an element of a list by setting a new value via its index.

```
my_list = ['foo', 'bar', 'baz']

my_list[1] = 'qux'

my_list
# ['foo', 'qux', 'baz']
```

IndexError

Lists raise an `IndexError` exception when you try to get or set a value at an index that does not exist.

```python
my_list = ['foo', 'bar', 'baz']

print(my_list[4]) # IndexError

my_list[4] = 'qux'  # IndexError
```

List membership

We can check whether or not an element exists somewhere in our list with the `in` operator.

The opposite can be performed with the `not in` operator!

```
my_list = ['foo', 'bar', 'baz']

'foo' in my_list # True
'foo' not in my_list # False
'qux' in my_list # False
```

List concatenation

We can also concatenate two lists together just like we concatenated strings together, with the `+` operator.

Note: this means we can also use the `+=` operator with lists!

```
list_a = ['foo', 'bar']
list_b = ['baz', 'qux']

list_a + list_b
# ['foo', 'bar', 'baz', 'qux']
```

Tuples

Another kind of iterable in python is the `tuple` data type.

Tuples are created with parentheses `()`.

But can also be created without any parentheses! The use of just a comma implies a tuple.

```
tuple_a = ('foo', 1)

tuple_b = 'foo', 1

tuple_a == tuple_b # True
```

Tuples

Elements in the tuple are also accessed via the index (like lists).

Tuples can be iterated over with a for loop, just like lists.

So what's the difference between a tuple and a list???

```
tuple_a = ('foo', 1)

tuple_a[0] # 'foo'
tuple_a[1] # 1

for el in tuple_a:
    print(el)
```

Tuples vs. lists

The biggest technical differences between a list and a tuple is that tuples have a fixed length and can't be mutated in-place.

However, lists can be used most places that a tuple is used, so the following rules can help you decide when to use a tuple and when to use a list:

```
LIST: Potentially many elements, unknown number of elements, relatively homogenous elements.
TUPLE: Few elements, fixed number of elements, completely heterogeneous elements.
```

Tuples vs. lists

The name comes from here: double, triple, quadruple, quintuple, sextuple, septuple, octuple.

Which gives a hint that they should be of fixed length! Because of this, we rarely iterate over them in a for loop like lists.

Instead, we often use **destructuring**.

Destructuring tuples

Because they have a fixed length, we often **destructure** tuples!

Destructuring is the act of taking apart an advanced data structure and assigning underlying values to variables.

Note: technically we can destructure lists like this, but we need to know the length in advance, which is often not the case with lists!

```
tuple_a = ('foo', 1)

name, score = tuple_a

name # 'foo'
score # 1
```

Lists of tuples

Lists of tuples are a convenient combination.

They can be destructured in a for loop!

```python
grades = [('Foo', 9.5),
          ('Bar', 6.3),
          ('Baz', 8.8)]

for student, grade in grades:
    print(student + ' got a: ', grade)
```

Review

1. List operations
2. Tuples
3. Destructuring