Tuples

Lists work well for storing collections of items that can change throughout the life of a program. The ability to modify lists is particularly important when you're working with a list of users on a website or a list of characters in a game. However, sometimes you'll want to create a list of items that cannot change. Tuples allow you to do just that. Python refers to values that cannot change as *immutable*, and an immutable list is called a *tuple*.

Defining a Tuple

A tuple looks just like a list except you use parentheses instead of square brackets. Once you define a tuple, you can access individual elements by using each item's index, just as you would for a list.

For example, if we have a rectangle that should always be a certain size, we can ensure that its size doesn't change by putting the dimensions into a tuple:

dimensions.py

```
❶ dimensions = (200, 50)
❷ print(dimensions[0])
  print(dimensions[1])
```

We define the tuple `dimensions` at ❶, using parentheses instead of square brackets. At ❷ we print each element in the tuple individually, using the same syntax we've been using to access elements in a list:

```
200
50
```

Let's see what happens if we try to change one of the items in the tuple `dimensions`:

```
dimensions = (200, 50)
❶ dimensions[0] = 250
```

The code at ❶ tries to change the value of the first dimension, but Python returns a type error. Basically, because we're trying to alter a tuple, which can't be done to that type of object, Python tells us we can't assign a new value to an item in a tuple:

```
Traceback (most recent call last):
  File "dimensions.py", line 2, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

This is beneficial because we want Python to raise an error when a line of code tries to change the dimensions of the rectangle.

NOTE *Tuples are technically defined by the presence of a comma; the parentheses make them look neater and more readable. If you want to define a tuple with one element, you need to include a trailing comma:*

```
my_t = (3,)
```

It doesn't often make sense to build a tuple with one element, but this can happen when tuples are generated automatically.

Looping Through All Values in a Tuple

You can loop over all the values in a tuple using a for loop, just as you did with a list:

```
dimensions = (200, 50)
for dimension in dimensions:
    print(dimension)
```

Python returns all the elements in the tuple, just as it would for a list:

```
200
50
```

Writing over a Tuple

Although you can't modify a tuple, you can assign a new value to a variable that represents a tuple. So if we wanted to change our dimensions, we could redefine the entire tuple:

```
❶ dimensions = (200, 50)
  print("Original dimensions:")
  for dimension in dimensions:
      print(dimension)

❷ dimensions = (400, 100)
❸ print("\nModified dimensions:")
  for dimension in dimensions:
      print(dimension)
```

The lines starting at ❶ define the original tuple and print the initial dimensions. At ❷, we associate a new tuple with the variable `dimensions`. We then print the new dimensions at ❸. Python doesn't raise any errors this time, because reassigning a variable is valid:

```
Original dimensions:
200
50

Modified dimensions:
400
100
```

When compared with lists, tuples are simple data structures. Use them when you want to store a set of values that should not be changed throughout the life of a program.