Lists, Tuples, Dictionaries

Now that we're familiar with the scalar types, let's move on to various collections.

In this section we're going to cover a few new types:

- list
- dict
- tuple
- set
- frozenset

As well as the concept of **mutability**.

Lists

Many languages only have one built-in complex type, commonly referred to as an array. You are probably familiar with syntax like `[1, 2, 3, 4]`.

In Python, `[1,2,3,4]` defines what we call a `list`. A list can also be declared as `list(1, 2, 3, 4)`.

The simplest list is the empty list `[]`.

(It is worth noting that the empty list evaluates to `False` when used in a boolean context.)

There are a few things we typically want to do with lists:

- retrieve (or set) an item at a specific position via indexing
- check for membership (e.g. does b_list contain the number 2?)
- concatenate two lists together
- get the length of the list
- add an item to the list
- remove an item from the list

Indexing

Like most languages we can also use `ourlist[idx]` to get and set an item at a specific index:

```
>>> words = ['green', 'blue', 'red']
>>> words[0]
'green'
>>> words[2] = 'orange'
>>> words
['green', 'blue', 'orange']
```

Unlike a lot of other languages, the indexing syntax in Python is quite powerful.

You can get a range of elements using what is called a **slice**:

```
>>> words = ['cat', 'dog', 'snake', 'fish', 'elephant']
>>> words[1:4]
['dog', 'snake', 'fish']
```

This gets the elements starting at 1 and ending at 3. Note that the range is inclusive on the lower end only. v: latest
seems odd but is quite helpful when doing math on indices.)

You can also use negative indices, which count from the back of the list:

```
>>> words = ['cat', 'dog', 'snake', 'fish', 'elephant']
>>> words[-1]
['elephant']
>>> words[1:-1]     # trims off first and last
['dog', 'snake', 'fish']
```

You can specify a third part of the **slice** called the stride. By default the stride is one, meaning it steps through from *start* to *end* one at a time.

Let's try changing that:

```
>>> words = ['cat', 'dog', 'snake', 'fish', 'elephant']
>>> words[0:5:2]
['cat', 'snake', 'elephant']
```

That took every other element.

You can omit any part of the slice and it'll default to *start*, *end* and *1*. So let's look at one last example:

```
>>> words = ['cat', 'dog', 'snake', 'fish', 'elephant']
>>> words[::-1]
['elephant', 'fish', 'snake', 'dog', 'cat']
```

Checking For Membership with `in`

The first thing we're going to do is to check if an item is a member of a list:

```
>>> a_list = [1, 2, 3, 4]
>>> 4 in a_list
True
>>> 'five' in a_list
False
```

This is actually a new operator, `in` checks for containment in a collection. It can also be used on strings:

```
>>> 'hello' in 'hello world'
True
```

Concatenating Lists

Let's try `+`:

```
>>> a_list = [1, 2, 3, 4]
>>> b_list = ['cinco', 'six', 'VII']
>>> a_list + b_list
[1, 2, 3, 4, 'cinco', 'six', 'VII']
```

`len`

There's a special builtin function to get the length of a `list`:

```
>>> len([1, 2, 3, 4])
4
```

v: latest

Appending & Removing Items

Sometimes you just want to append a single item. There isn't an operator for this but we'll use a method of the list:

```
>>> a_list = [1, 2, 3, 4]
>>> a_list.append(5)
>>> a_list
[1, 2, 3, 4, 5]
```

And we'll use `.pop()` to remove the first item:

```
>>> a_list.pop()
1
>>> a_list
[2, 3, 4, 5]
```

Pop by default removes the first item and returns it. You can also pass an argument to `pop` to remove a different item.

You can also remove items by value using `.remove()`:

```
>>> a_list = [1, 2, 3, 4]
>>> a_list.remove(2)
>>> a_list
[1, 3, 4]
>>> a_list = [2, 2, 2, 2]
>>> a_list.remove(2)        # will only remove the first occurence
>>> a_list
[2, 2, 2]
```

There are plenty of other methods on `list` to explore.

See the documentation on common sequence operations and mutable sequence types to see the other methods and builtins that operate on lists.

Mutability in Python

Lists are the first Python type that we've seen that is "mutable." A mutable type means that the inner value can change, not just what the variable references. This can be a confusing concept so let's dive a bit into it:

When we assign a scalar value like `x = 7` we are essentially saying that `x` now has the value `7` until we say otherwise. If we then say `y = x` we are saying `y` is `7` too. If we change `x` later, `y` does not also change. Here's a demonstration:

```
>>> x = 7
>>> y = x
>>> print('x =', x, '     y =', y)
x = 7      y = 7

>>> x = 8
>>> print('x =', x, '     y =', y)
x = 8      y = 7

>>> y = 9
>>> print('x =', x, '     y =', y)
x = 8      y = 9
```

Notice that x and y change independently of one another. This is also true if we do it with any of the other from Part 1 (str, float, bool, etc.)

v: latest

We can change what **value** a variable points to as many times as we like: `x = 7`, `x = "hello"`, `x = 3.14`, etc.

Lists offer a different option. If we set `x` to a list, we can also change the make-up of the list:

```
>>> x = [1, 2, 3]
>>> y = x
>>> print('x =', x, '        y =', y)
x = [1, 2, 3]        y = [1, 2, 3]

>>> x.append(4)
>>> print('x =', x, '     y =', y)
x = [1, 2, 3, 4]     y = [1, 2, 3, 4]
```

Wait.. what?! `y` changed when `x` changed?

What is the difference between this and the above? Why are `x` and `y` now linked?

Let's explore...

```
>>> x = [1, 2, 3]
>>> y = x
>>> z = x
>>> print('x=', x, '        y=', y, '        z=', z)
x= [1, 2, 3]        y= [1, 2, 3]        z= [1, 2, 3]


>>> # append 4 to x, it will also appear in y and z
>>> x.append(4)
>>> print('x=', x, '     y=', y, '     z=', z)
x= [1, 2, 3, 4]     y= [1, 2, 3, 4]     z= [1, 2, 3, 4]

>>> # ok at this point all three have changed, let's assign to x now
>>> x = [0, 0, 0]
>>> print('x=', x, '        y=', y, '     z=', z)
x= [0, 0, 0]        y= [1, 2, 3, 4]     z= [1, 2, 3, 4]

# wait... that time it only changed x?
```

If the above isn't making sense to you that's fine. Let's think about what we know though:

In our first example with the integers, any time we set `x =` or `y =` it didn't affect the other one.

In our second example, when we called `x.append` we know that `y` changed too.

In our third example, when we set `x = [0, 0, 0]` those changes didn't affect `y` or `z`.

The common thread here is the `=` operator. When we set `x` to a new value, whether it is an `int` or `list` (or anything else)- we are creating a new **value**. When we call a method like `x.append(4)` though, we are not creating a new **value** but instead modifying (mutating) the underlying array we created when we said `x = [1, 2, 3]`.

It can take some getting used to but there's a separation between the concept of a **value** and a **variable**.

Whenever we define a variable by typing an int, str, float, or list directly (what we call a **literal**) we're doing the same thing, regardless of what type we're using:

- `x = [1, 2, 3]` creates a new list with the **value** [1, 2, 3] and points `x` at it.
- `y = "hello"` creates a new string with the **value** "hello" and points `y` at it.

v: latest

When we assign from one variable to another the behavior varies slightly though:

- `x = y` when y is **immutable** means that `x` now has a copy of the **value** of `y`.

- `x = y` when y is **mutable** means that `x` is a **reference** to the same **value** as `x`

This is why we have the ability to modify our list in the above examples, but when we assign a new value to `x` the other values do not update.

x = 7

```
variables                 values

x   --------------------->  7
```

y = x

```
variables                 values

x   --------------------->  7
y   --------------------->  7
```

a = [1, 2, 3]

```
variables                 values

x   --------------------->  7
y   --------------------->  7
a   --------------------->  [1, 2, 3]
```

b = a

```
variables                 values

x   --------------------->  7
y   --------------------->  7
a   --------------------->  [1, 2, 3]
b   ----------/
```

a.append(4)

```
variables                 values

x   --------------------->  7
y   --------------------->  7
a   --------------------->  [1, 2, 3, 4]
b   ----------/
```

a = []

```
variables                 values

x   --------------------->  7
y   --------------------->  7
a   --------------------->  []
b   --------------------->  [1, 2, 3, 4]
```

b = None

```
variables                 values

x   --------------------->  7
y   --------------------->  7
a   --------------------->  []
```

v: latest

```
b   --------------------->  None
                            [1, 2, 3, 4] *will be deleted*
```

Tuples

We saw earlier that a list is an **ordered mutable** collection. There's also an ordered **immutable** collection.

In Python these are called tuples and look very similar to lists, but typically written with () instead of []:

```
a_list = [1, 'two', 3.0]
a_tuple = (1, 'two', 3.0)
```

Similar to how we used `list` before, you can also create a `tuple` via `tuple(1,2,3)`.

The difference being that tuples are immutable. This means no assignment, append, insert, pop, etc. Everything else works as it did with lists: indexing, getting the length, checking membership, etc.

Like lists, all of the common sequence operations are available.

Another thing to note is that strictly speaking, the comma is what makes the tuple, not the parentheses. In practice it is a good idea to include the parentheses for clarity and because they are needed in some situations to make operator precedence clear.

Let's look at a quick example:

```
>>> a_tuple = (1, 2, 3)
>>> another_tuple = 1, 2, 3
>>> a_tuple == b_tuple
True
```

This is also important if you need to make a single element tuple:

```
>>> x = ('one')
>>> y = ('one',)
>>> type(x)
str
>>> type(y)
tuple
```

This is a common gotcha, when a function or setting requires a `list` or `tuple` it is easy to forget the trailing comma if you're passing a single element.

OK, so there are two list-like things and one is just like a `list` but can't be modified. Why not use a `list` all the time? So why use them if lists do more?

- Tuples are lighter-weight and are more memory efficient and often faster if used in appropriate places.
- When using a `tuple` you protect against accidental modification when passing it between functions.
- Tuples, being immutable, can be used as a key in a dictionary, which we're about to learn about.

Dicts

Now with an understanding of mutability we're ready to discuss Python's `dict` type.

Dictionaries are a common feature of modern languages (often known as maps, associative arrays, or hasl v: latest
which let you associate pairs of values together. The term dictionary is a nod to this, as you can think of them as being terms/descriptions in a sense.

In Python `dict` is perhaps the most important type there is, when we learn about **modules** and **classes** we'll see why.

A dictionary declaration might look like this:

```
birth_year = {'Hamilton': 1755, 'Jefferson': 1743, 'Franklin': 1706, 'Adams': 1735}
```

Like `list` there is a second way to declare a `dict`:

```
sound = dict(dog='bark', cat='meow', snake='hiss')
```

And an empty `dict` can be declared as either `{}` or `dict()`

In Python we refer to 'dog', 'cat', and 'snake' as the **keys** and 'bark', 'meow', and 'hiss' as the **values**.

A few things we already saw on `list` work the same for `dict`:

- Similarly to how we can index into lists we use `d[key]` to access specific elements in the dict. There are also a number of methods available for manipulating & using data from `dict`.
- `len(d)` gets the number of item in the dictionary.
- `k in d` checks if `k` is a key in the dictionary.
- `d.pop(k)` pops an item out of the dictionary and returns it, similarly to how list's `pop` method worked.

Note: attempting to read from a key that hasn't been set yet will result in an exception, we'll cover Exceptions and reovery from them a bit later

Other commonly used `dict` methods:

- `keys()` - returns an **iterable** of all keys in the dictionary.
- `values()` - returns an **iterable** of all values in the dictionary.
- `items()` - returns an **iterable** list of `(key, value)` tuples.

We'll cover these methods in more detail when we get to `for` loops and iteration.

Check the Python docs for dict for more details.

Sets

You may have noticed in the definition of lists above we called them **ordered** in addition to **mutable**.

Items in a list have a definite order, which is what allows us to index into them or sort them. Sometimes you don't care what the order of items is, simply whether the collection contains it or not. In this case Python has a special type that is probably not familiar to you from other languages: `set`.

A `set` is an unordered, mutable collection. In math they're typically denoted with `{}`, you can use the same in Python. The difference from `dict` syntax is that there aren't key-value pairs separated by `:`.

Many of the same operations you have on lists and tuples are available on `set`, but anything dealing with order isn't. This means you can't index into a set, or pop a specific element. (You can call `s.pop()` which will pop an arbitrary element out of `s`.)

So now we've seen an ordered mutable collection (`list`), an ordered immutable collection (`tuple`), and an unordered mutable collection (`set`), so you may be wondering if there is an unordered immutable. The answer is that there is, and it is called `frozenset`.

v: latest

mutable?	Ordered	Unordered

mutable?	Ordered	Unordered
Mutable	`list`	`set`
Immutable	`tuple`	`frozenset`

Constructing Sets:

```
a = {1,2,3}                  # set literal
b = {}                       # not a set!, ``dict`` literal
c = set()                    # empty set
d = frozenset({1,2,3})       # frozenset
e = frozenset()              # empty frozenset
f = frozenset([1,1,1,1,1])   # == frozenset({1}), remember- sets cannot contain duplicates
```

The sequence operations available on list aren't available, though some basics like `len` and `in` still work.

Instead of `append`, `set` uses `add`:

```
>>> a = {1, 2, 3}
>>> a.add(4)
>>> a.add(2)
>>> a
{1, 2, 3, 4}
```

Common set arithmetic operators are available as methods and operators:

method	operator
`a.issubset(b)`	`a <= b`
`a.issuperset(b)`	`a >= b`
`a.union(b)`	`a | b`
`a.intersection(b)`	`a & b`
`a.difference(b)`	`a - b`
`a.symmetric_difference(b)`	`a ^ b`
`a.isdisjoint(b)`	

You can check out the Python set documentation for more details and other methods.

Next Steps

Next we'll be discussing *Control Flow*.

v: latest