Text

Lists and Tuples

Section 219

Completed

< back to module

A list is exactly what it sounds like: a list of things stored in a particular order. It could be anything: the numbers from 1 - 10, a list of animals, all the nail polish colours you could name... Here is a list of Agent M's Favourite Linux distributions:

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(fav_linux_distros)
```

If we print a list like we have above, we'll get the whole list returned to us like so:

```
['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
```

We probably won't find too much use for this list in this form: it will be more useful for us to be able to access individual items within this list using its index position within the list.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(fav_linux_distros[0])
print(fav_linux_distros[1])
print(fav_linux_distros[2])
```

The above example will output:

```
Mint
Debian
Ubuntu
```

Note that when we want to access the very first item in the list, we use 0 instead of 1. This is true of most programming languages, wherever you're counting 'things' in computer code, we always start with 0. So if we're looking for the fifth item in Agent M's Linux distributions list, we'd use `fav_linux_distros[4]`.

What happens if we try to use an index outside the number of items in the list?

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(fav_linux_distros[7])
```

Here we've asked for the item in the list at index `7`, but we only have 6 items in the list. If nothing exists at the index we've requested, we'll get a `IndexError` error back instead.

```
$ python program.py
Traceback (most recent call last):
  File "program.py", line 2, in <module>
    print(fav_linux_distros[7])
IndexError: list index out of range
```

Python also gives us a shortcut way of getting the very last item in a list:

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(fav_linux_distros[-1])
```

By asking for an item at the index of -1 Python will always return the last item in the list. This is helpful because we don't always know how long our list will be. This syntax extends to other negative index values as well, letting you count backward from the end of the list rather than forward from the front. Let's try it out.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(fav_linux_distros[-1])
print(fav_linux_distros[-2])
```

This code will output:

```
Fedora
Manjaro
```

When we pluck an item from a list like this, we also get its individual item type. In the Linux distributions example we get a string, which we can then manipulate as a string using our various string manipulation tools.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
top_fav_distro = fav_linux_distros[0]

print("Agent M's favourite Linux distro is " + top_fav_distro.upper() + "!")
```

This gives us a string where we've modified an item from our list to appear in all uppercase, to best communicate how much Agent M loves Mint.

```
Agent M's favourite Linux distro is MINT!
```

We can also get a quick count of the things in our list by using the `len()` function.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']

print(len(fav_linux_distros))
```

This will print out `6`, which is indeed the number of items in Agent M's list.

Mixed type lists

Python lets us create mixed-type lists, incorporating all different types of data into a single list.

```
misc = ['purple', 99, 3.14, False]

print(misc[0])
print(misc[1])
print(misc[2])
print(misc[3])
```

And the result:

```
purple
99
3.14
False
```

We can even put lists inside of other lists, like this:

```
misc = ['purple', 99, 3.14, False, ['apple', 'orange', 'pear']]

print(misc[0])
print(misc[1])
print(misc[2])
print(misc[3])
print(misc[4])
print(misc[4][0])
print(misc[4][1])
print(misc[4][2])
```

Which outputs as:

```
purple
99
3.14
False
['apple', 'orange', 'pear']
apple
orange
pear
```

Notice how we access the individual items 'apple', 'orange' and 'pear' inside the inner list? The inner list is at index 4 of the primary list, and 'apple' is at index 0 of the inner list. So to get the first item inside the inner list, we use `misc[4][0]`.

Modifying a list

What happens when Agent M wants to update the list of Linux distributions? How do we modify the list we have with new information?

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
print(fav_linux_distros)

fav_linux_distros[4] = 'Elementary'
print(fav_linux_distros)
```

We've created a list and saved it to the variable `fav_linux_distros` with `Mint` as the first item in the list. Then we've changed the value of the first item `Debian`.

We've printed the list out before and after we've made the change so you can see what it looks like at both stages. Only the first item in the list has changed: everything else remains the same.

```
['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
['Elementary', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
```

We can also modify a list by applying a sort to it. Agent M likes things to be nice and orderly, so let's sort this list of Linux distros alphabetically.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
fav_linux_distros.sort()

print(fav_linux_distros)
```

As we hoped, this gives us:

```
['Arch', 'Debian', 'Fedora', 'Manjaro', 'Mint', 'Ubuntu']
```

Alot of periwinkle, if we want to frustrate Agent M we can always reverse this list.

```
fav_linux_distros = ['Mint', 'Debian', 'Ubuntu', 'Manjaro', 'Fedora', 'Arch']
fav_linux_distros.sort()

print(fav_linux_distros)

fav_linux_distros.reverse()
print(fav_linux_distros)
```

Our output from the above code would be:

```
['Arch', 'Debian', 'Fedora', 'Manjaro', 'Mint', 'Ubuntu']
['Ubuntu', 'Mint', 'Manjaro', 'Fedora', 'Debian', 'Arch']
```

Adding items to lists

There are a few different ways to add things to lists. Let's start by simply adding things onto the end of our list.

```
travel_bucket_list = ['Tokyo', 'Hawaii', 'London']
travel_bucket_list.append('New York')
travel_bucket_list.append('Berlin')

print(travel_bucket_list)
```

Here we use the `append()` method, which allows us to stick items onto the end of a list. When we print out our list after appending our new items, we can indeed see they've been added:

```
['Tokyo', 'Hawaii', 'London', 'New York', 'Berlin']
```

We can even start with an empty list:

```
travel_bucket_list = []
travel_bucket_list.append('Tanzania')
travel_bucket_list.append('Barcelona')
travel_bucket_list.append('Dubai')

print(travel_bucket_list)
```

As expected, we've taken an empty list in line 1 and turned it into a list with 3 items by appending them one by one.

```
['Tanzania', 'Barcelona', 'Dubai']
```

We can also add items into an existing list at any position we want. Let's try adding a new item to the front of our travel bucket list:

```
travel_bucket_list = ['Tokyo', 'Hawaii', 'London']
travel_bucket_list.insert(0, 'New York')

print(travel_bucket_list)
```

Now we've put New York to the front of our list, and bumped everything else down one. Our output shows us our new list:

```
['New York', 'Tokyo', 'Hawaii', 'London']
```

Removing items from lists

Sometimes we want to remove an item from our list, for instance if we travelled to one of the places on our travel bucket list. Let's remove 'London' from our list.

```
travel_bucket_list = ['Tokyo', 'Hawaii', 'London']
visited = travel_bucket_list.pop()

print(travel_bucket_list)
print(visited)
```

We can see how we've changed our list, and also that we've been able to 'pop' London out of the list into the variable `visited`, so that we can continue to use it later in our program if we want.

```
['Tokyo', 'Hawaii']
London
```

The `pop()` method has taken the last item from the list and 'popped' it off. If we wanted to remove a different item from the list, we can use `remove()` with an index to specify which item we want popped out.

```
travel_bucket_list = ['Tokyo', 'Hawaii', 'London']
visited = travel_bucket_list.pop(1)

print('I recently went to ' + visited)
print(travel_bucket_list)
```

This outputs:

```
I recently went to Hawaii
['Tokyo', 'London']
```

What if we don't know what the position is of the thing we want to remove? Perhaps we write our list a long time ago, and can't remember what index position 'Hawaii' is at. How do we remove Hawaii from our list then? Voila:

```
travel_bucket_list = ['Tokyo', 'Hawaii', 'London']
travel_bucket_list.remove('Hawaii')

print(travel_bucket_list)
```

The `remove()` method will allow us to remove an item from our list without specifying an index. A very handy method to remember. As expected, the above code gets us the following output:

```
['Tokyo', 'London']
```

Tuples

Tuples are very similar to lists, except for 1 important quality - they are immutable 'immutable' means the content items in the list can't be changed once they are set. This can sometimes be useful is a programmer if we want to ensure a piece of data can't be changed. The use of a tuple forces us to copy the information and modify that copy, leaving the original intact and unchanged.

Here's an example of how a tuple is declared and used:

```
stonehenge = ('51.1789256376', '-1.8223767284')
print('Stonehenge latitude: ' + stonehenge[0])
print('Stonehenge longitude: ' + stonehenge[1])
```

We've used a tuple for the coordinates of Stonehenge, since we don't want anyone to be able to change them (they're immutable, because Stonehenge isn't going anywhere (at least not anytime soon)). Notice how we've used round brackets `()` instead of square brackets `[]`. When we created our tuple? That's how Python knows it's a tuple we want instead of a list.

We can access the individual items in our tuple exactly the same way we access items in a list. The output of our code above is shown below.

```
Stonehenge latitude: 51.1789256376
Stonehenge longitude: -1.8223767284
```

What happens if we try to modify one of our values?

```
stonehenge = ('51.1789256376', '-1.8223767284')
stonehenge[0] = 'something else'
```

Here's the output we get when we try to run this code:

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

Are lists just arrays?

If you're familiar with other high-level programming languages, you've probably played with a data structure called an 'array' before. A list looks an awful lot like an array, and for most use cases it will more or less act like one. In this course, you can think of a 'list' and an 'array' as being the same thing.

However, there are some subtle differences in Python between a list and an actual array, which you can import and use. We'll leave you to Google for the difference if you're interested to dig into the details.

Completed

< back to module