Lists and Dictionaries

List and Dictionaries

Data Structures

In the previous module, we learned about **assignment and the kinds of values (https://uk.instructure.com/courses/2051722/pages/values-and-variables)** we can have in python. Just a quick recap:

- We *assign* a *value* to a *variable* with `=`
  - e.g. `this_class = "Lin317"`
- *Numeric* values are numbers. We can do any kind of math with them.
- *String* values are text. They are enclosed either in double quotes (`""`) or single quotes (`''`)
- *Logical* values are `True` and `False`. We'll usually be creating these values with some kind of logical comparison.
  - e.g. `this_class == "Lin221"` would give us `False`.

In this module, we'll be learning about data structures in python, specifically *lists* and *dictionaries*.

Lists

We've already seen how we can assign single values to a variable. For example, Jane Austen's first published novel was *Sense and Sensibility*. We can assign this title to a variable like so:

```
first_novel = "Sense and Sensibility"
```

However, Jane Austen published seven novels:

- *Sense and Sensibility*
- *Pride and Prejudice*

- *Mansfield Park*
- *Emma*
- *Northanger Abbey*
- *Persuasion*
- *Lady Susan*

If we were doing some kind of data analysis of her books, it wouldn't be very efficient to assign each title to its own variable. Instead, we'd want to create a *list* of these book titles, and assign that list to a variable. We can create such a list with `[]`

```
novels = ["Sense and Sensibility", "Pride and Prejudice",
          "Mansfield Park", "Emma",
          "Northanger Abbey", "Persuasion",
          "Lady Susan"]
```

Inside of the brackets `[]`, we've put each novel title in quotes as a string value. Each value is separated from the next with a comma `,`. Whitespace, like spaces and new lines, are there just to make it look visually nicer.

If you run `print(novels)` after running the code above, you'll see that it prints out the list, with the values in the same order as we entered them above.

Adding two lists together

Just like we could concatenate two strings together with `+`, we can also concatenate two lists together. For example, Jane Austen also had two unfinished novels, *The Watsons* and *Sandition*. We can create a new list called `unfinished` like so:

```
unfinished = ["The Watsons", "Sandition"]
```

Now, to create a list of all of Jane Austen's novels, finished and unfinished, we just add the two together.

```
novels + unfinished
```

This will print out a new list with all of the novel titles from `novels`, followed by all of the titles from `unfinished`.

"Indexing" a list

We'll often want to get a specific value from a specific location in a list. For example, the titles in `novels` are entered in publication order. If we wanted to get the title of the first book Jane

Austen published, we'd use a numeric *index* that we put in brackets `[]` after `novels`. So, the title of the first book Jane Austen published was:

```
novels[0]
```

Here, we see the most commonly annoying thing in learning programming languages:

Indexing starts with 0. The first item in a list has the index 0.

To get the index of the second item in the list, we'd do.

```
novels[1]
```

Getting the index of a value

Lists in python come with a number of *methods*. Methods in python are special kinds of functions that come bundled up in the values. You can get a full list of available methods with `dir()`.

One special method for lists is `.index()`. If we know the value we want, but don't know its index, we can use `.index()` to get it. For example, if we wanted the index for *Emma* in the list of novels, we would get it like this:

```
novels.index("Emma")
```

Emma's index is `3`, meaning it was Jane Austen's 4th novel.

For a more practical example of how we might use the `.index()`, let's create another list with the years each of Jane Austen's novels were published.

```
pub_year = [1811, 1813, 1814, 1815, 1818, 1818, 1871]
```

Since both `novels` and `pub_year` are both in order of publication, we can use the indices from one on the other. So, if we wanted to get the year in which *Emma* was published, we could do so like this:

```
emma_index = novels.index("Emma")
pub_year[emma_index]
```

If we wanted to know which book she published in the year 1814, we could do this:

```
year_index = pub_year.index(1814)
novels[year_index]
```

Turns out, it was *Mansfield Park* that was published in 1814.

Negative indexing

We can also use negative numbers to index lists, which count from the end of the list. To get the last novel Jane Austen published, we'd use the index `-1`.

```
novels[-1]
```

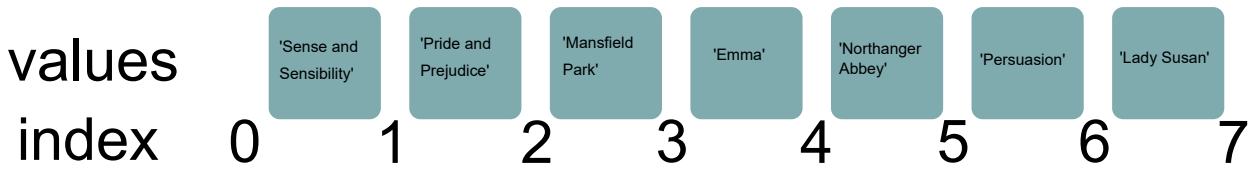
Turns out this was *Lady Susan* (which was published posthumously).

Slicing

Sometimes, you'll want to get a *range* of values from a list. We can do this by "slicing" the list. To get the first three novel titles from `novels`, we'll put `[0:3]` immediately after the variable name

```
novels[0:3]
```

It's a little weird that to get the third novel title, we use `novels[2]`, but to get the first three novels we do `novels[0:3]`. When slicing, it's more helpful to think of the indices as coming in between the values in the list, so that `[0:3]` is giving us back everything *between* the 0 index and the 3 index.



Check if a value is in a list

One really useful thing we can do is check to see if a particular value is in a list. For example, if we wanted to double check that *Emma* made it into the list, we could run

```
"Emma" in novels
```

On the other hand, if I'd gotten really confused and thought that Jane Austen had also written Frankenstein, I could double check that as well.

```
"Frankenstein" in novels
```

Remember that we can use variables exactly like the values they are assigned. So that if we assign a string to a variable called `novel_to_check`, we can do the same thing.

```
novel_to_check = "Lady Susan"
novle_to_check in novels
```

Lists can be very complex!

So far, we've only looked at lists of strings and lists of numbers, but you can create a list of *anything*, including other lists. For example, instead of creating two separate lists of novel title and novel publication date, we could create one complex list of lists like so:

```
complex_list = [ ["Sense and Sensibility", 1811],
                 ["Pride and Prejudice", 1813],
                 ["Mansfield Park", 1814],
                 ["Emma", 1815],
                 ["Northanger Abbey", 1818],
                 ["Persuasion", 1818],
                 ["Lady Susan", 1871] ]
```

Now, if we index `complex_list`, we get back *another* list!

```
complex_list[0]
```

To get the year of publication of Jane Austen's third book from `complex_list`, we need to do this:

```
complex_list[2][1]
```

Dictionaries

Dictionaries are very useful data structures that relate certain values, called *keys* to other values, called *values.* Much like we created lists with `[]`, we'll create dictionaries with `{}`. When creating a dictionary, we need to define the key and value pair like `key : value`. For example, here's a dictionary representing the relationship between Jane Austen's fist three novel titles, and their year of publication.

```
novel_dict = {"Sense and Sensibility" : 1811,
              "Pride and Prejudice" : 1813,
              "Mansfield Park": 1814}
```

The value on the left of the `:` is the key. You can only use a key value once in a dictionary. The value on the right of `:` is the value, and there are no restrictions on how many times you can repeat a value.

Indexing a dictionary

To index a dictionary, we enter the key value inside of `[]`. So, to get the publication year of *Pride and Prejudice*, we do the following:

```
novel_dict["Pride and Prejudice"]
```

Adding to a dictionary

To add a new value to a dictionary, we enter whatever value we want the key to be in `[]`, then assign the value to it. For example, to add *Emma* and its publication year to this dictionary, we would do the following:

```
novel_dict["Emma"] = 1815
```

Checking if a key exists

If we use `in` on a dictionary, instead of checking to see if the value we're asking about exists in the values, it check to see if it exists in the *keys.*

```
# this will return True
"Pride and Prejudice" in novel_dict

# this will return False
1811 in novel_dict
```