Using text and strings!

Using Text and Strings!

Doing useful things

In this module, we will cover useful things we can do with text, including

- How to read text files into python.
- How to edit, clean up, and modify text for additional analysis
- How to use our good old friends regular expressions!

Reading in text

So far, we've only discussed how to assign values to variables by typing out the value within an interactive python session or a python script. For example, to assign the book title *Mansfield Park* to a variable, we would need to type it out like so:

```
book_var = "Mansfield Park"
```

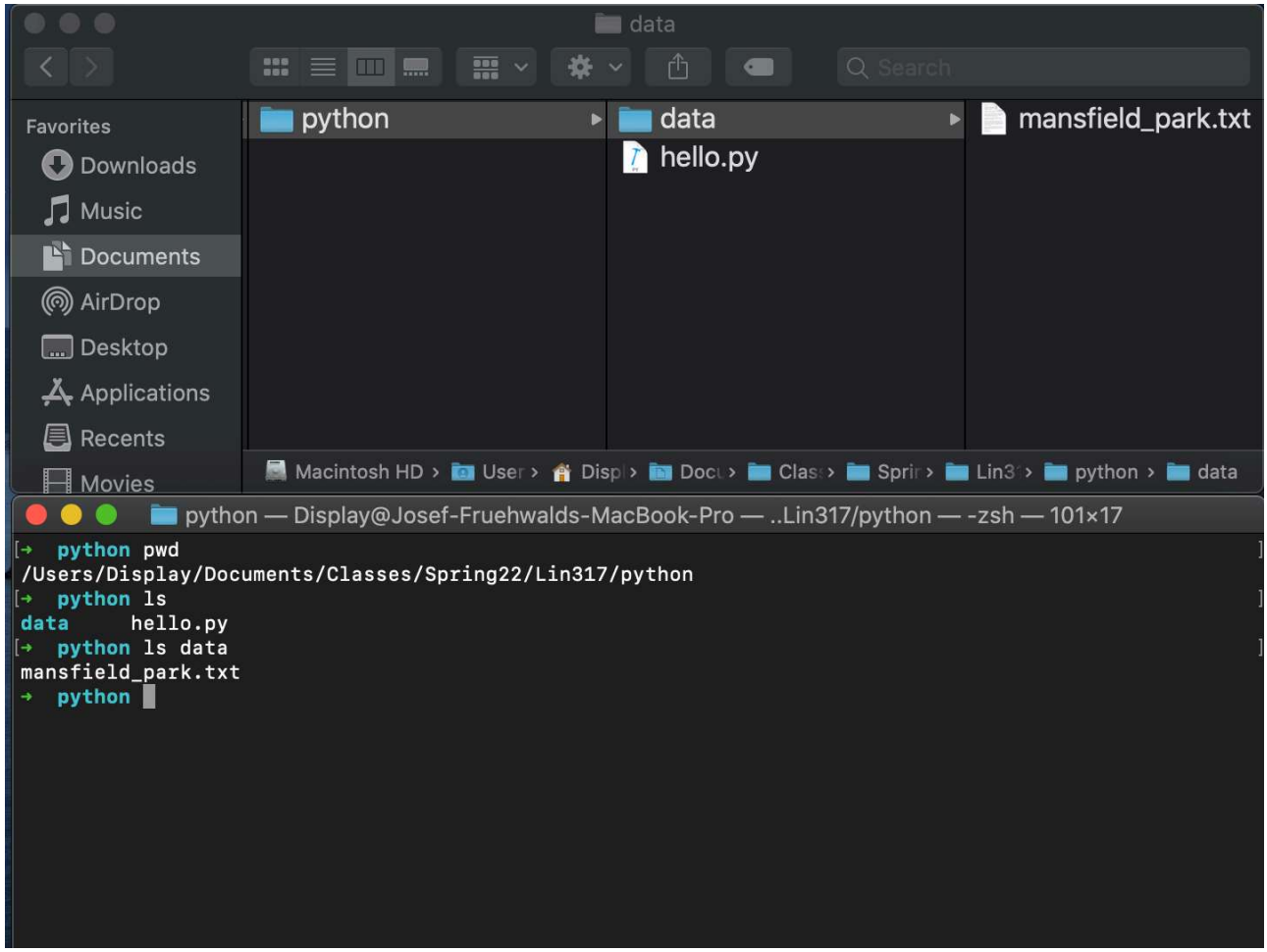
But let's say we wanted to analyze all of the text of ***Mansfield Park* after downloading it from Project Gutenberg (https://www.gutenberg.org/files/141/141-0.txt)**? It's not feasible to copy and paste the whole text into our interactive python session! Instead, we'll need to do the following steps:

1. We need to tell python *where* the text file is by giving it a path.
2. We need to tell python to *open* the text file to read it.
3. We need to tell python to *read* the lines of text as strings.

These three steps will work for any plain text document.

Tell python *where* the text file is.

In order to read a text file into python, we need to tell python where it is. We do this by giving it **a path to the file (https://uk.instructure.com/courses/2051722/pages/navigating-directories)**. For these examples, I have navigated to a directory called `python`. Inside of the `python` directory is another directory called `data`. Inside of the data directory is a file called `mansfield_park.txt` which contains the text of the Jane Austen novel as available on **Project Gutenberg (https://www.gutenberg.org/files/141/141-0.txt)**. Here is an image showing all of this information in a file browser as well as in the command line interface.



If launch an interactive python session from this directory, or run a python script from this directory, I can either give python the relative path or the global path to the text file.

- relative path from `python` directory: `data/mansfield_park.txt`
- global path: `/Users/Display/Documents/Classes/Spring22/Lin317/python/data/mansfield_park.txt`

Remember: these paths are descriptions of where the file is on *my* computer. They won't be the same as the description of where the file is on *your* computer!

For now, I'll assume we're running an interactive `python3` or `ipython` session. I'll assign the relative path to the Mansfield Park file to a variable called `book_location` as a string.

```
book_location = "data/mansfield_park.txt"
```

The variable `book_location` is just a string right now. We can print it out just like any string variable.

```
print(book_location)
```

Opening the file

To open a file in python, whether we want to read data in from it or write data out to it, we use the function `open()`. The first argument to `open()` is the path to the file as a string, and the second argument is the "mode". In this case, we want to read data in, so we'll say `mode='r'`.

```
book_file = open(book_location, mode = 'r')
```

The variable `book_file` is now an object that lets us interact with the text file.

Reading in lines

To read in one line from the file, we use the `.readline()`. Just as a reminder, methods are like special functions that come bundled up in variables, and we use them by typing them in immediately after the variable name with a dot `.` in between.

```
book_file.readline()
```

When I did this, python printed the following line out:

```
# '\ufeffThe Project Gutenberg eBook of Mansfield Park, by Jane Austen\n'
```

Let's run `book_file.readline()` one more time. This time, it printed out the following line.

```
# '\n'
```

And if we run it again:

```
# 'This eBook is for the use of anyone anywhere in the United States and\n'
```

Every time we run `book_file.readline()`, it will read in the next line of the file as a string.

We could keep rerunning `book_file.readline()` until it was done reading in all of the lines, or we could read in all of the remaining lines with the method `.readlines()`. This method reads in all lines in the file, and returns a list with each line as a string.

```
book_lines = book_file.readlines()
```

The variable `book_lines` is now a list with every line of the book in it. You can print out the whole list, or double check the length of it with `len()`.

```
len(book_lines)
# 16051
```

To get the first line in the `book_lines` list, we can index it with `0`

```
book_lines[0]
# 'most other parts of the world at no cost and with almost no restrictions\n'
```

Doing things to text

Now that we've read in the entire book *Mansfield Park*, let's choose a single line to work with modifying text.

```
one_line = book_lines[200]
one_line
# 'introduced into the society of this country under such very favourable\n'
```

Cleaning up lines

The first thing to notice about the value in `one_line` is that at the very end it has `\n`. This is a symbol representing the new line character. It's very important for the text file to display properly, but it's less important for doing our pythonic analysis. In fact, it's better to almost always get rid of the newlines at the end of strings. One reason is that if we started doing textual analysis without getting rid of `\n` at the end, python would treat `favourable` and `favourable\n` as two different words!

Fortunately, it's very easy to get rid of newlines with the `.rstrip()` method, which strips off all whitespace character from the right side of a string.

```
one_line.rstrip()
# 'introduced into the society of this country under such very favourable'
```

Newline gone!

It's important to note that the `.rstrip()` method has not made changes "in place," meaning it hasn't changed the value assigned to `one_line`. Instead, it creates a new value that we can assign to a new variable. There are a few more steps of cleanup that I would recommend though.

First, it's usually a good idea to also do `.lstrip()`, which eliminates all white space characters to the left of the string. It's not relevant to the specific line we're using as an example, but it's possible some other line could begin with a space, or a tab, or something.

```
one_line.rstrip().lstrip()
```

It's also a good idea to convert all of the letters either to all upper case or all lower case. When doing textual analysis, we wouldn't want python to treat `The` and `the`. We can use either the `.lower()` or the `.upper()` methods.

```
clean_line = one_line.rstrip().lstrip().lower()
clean_line
# 'introduced into the society of this country under such very favourable'
```

Editing the string

Now that we've got the cleaned up string, we can *do* things to it. For example, let's say we wanted to change the British spelling of `favourable` to the American spelling of `favorable`. We can do this with the method `.replace()`. Replace takes two arguments:

- `.replace()` method arguments
    1. The substring you want to replace
    2. The subtring you want to replace it with

So, to replace `favourable` with `favorable`, we would do the following:

```
clean_line.replace("favourable", "favorable")
# 'introduced into the society of this country under such very favorable'
```

We didn't need to type out the whole word, though. Python doesn't "know" about words, it just knows about characters that appear in a certain order. So this would have worked too:

```
clean_line.replace("favour", "favor")
# 'introduced into the society of this country under such very favorable'
```

We don't want to go too overboard with simplifying our replacement command, though. If we say we wanted to just replace `ou` with `o`, it would also change `country`!

```
clean_line.replace("ou", "o")
# 'introduced into the society of this contry under such very favorable'
```

Splitting up the string

Finally, we can take our `clean_line` and split it up into a list of words. Doing this is an important step in, for example, counting up the frequency of words, or searching for words.

To split a string into a list of substrings, we can use the `.split()` method. There is only one argument to `.split()`, the character(s) we want to split the string on. For words in a sentence, we'll want to split the string along the spaces, or `" "`.

```
clean_line.split(" ")
# ['introduced',
#  'into',
#  'the',
#  'society',
#  'of',
#  'this',
#  'country',
#  'under',
#  'such',
#  'very',
#  'favourable']
```

Now, each individual word is its own values in the list!

Sometimes you might want to split a list up according to some other value. The `.split()` method doesn't care if you're splitting the strings up into words, or any other meaningless strings. For example, we could have split `clean_line` according to the letter `e` instead.

```
clean_line.split("e")
# ['introduc',
#  'd into th',
#  ' soci',
#  'ty of this country und',
#  'r such v',
#  'ry favourabl',
#  '']
```

Regular Expressions!

Our old friends, Regular Expressions, can be used in python! They're not available by default, though. To use them, we're going to have to import a python package called `re`, for "regular expressions". Importing a package is very easy with they keyword `import`.

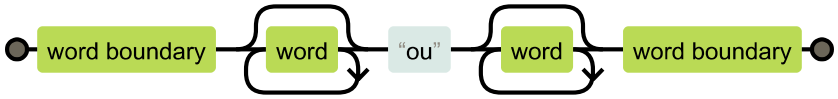
```
import re
```

Searching with Regex

To access the functions that `re` makes available, we enter the name of the package, a dot `.` then the name of the function. For example, in this line we've been working with, let's say I wanted to find every word that contained an `ou` sequence. We need a two step process for this:

1. Write the right regex. To indicate that it's a regular expression, we need use special quotes: `r''`
2. Use `re.findall()` with the regex and the string we want to search.

The right regex for "a whole word containing `ou` is `\b\w*ou\w*\b`



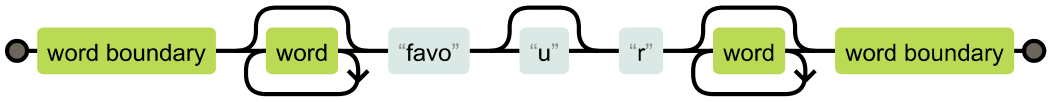
So, we'll assign this regex to a variable called `ou_regex`

```
ou_regex = r'\b\w*ou\w*\b'
```

Now, we can search for all examples in `clean_line`

```
re.findall(ou_regex, clean_line)
# ['country', 'favourable']
```

Or, maybe we want to find every word that contains "favor", but we're not sure if the author has used British or American spelling. The regex for that would be `\b\w*favou?r\w*\b`



Again, we can do a two step process here.

```
favor_regex = r'\b\w*favou?r\w*\b'
re.findall(favor_regex, clean_line)
# ['favourable']
```

Cleaning text with regex

It wasn't an issue for the particular line we were looking at, but another thing we may want to do is remove all punctuation from a string. For example, let's write a short two sentences and assign it to a variable.

```
hello = "  Hello, what a great day! Don't you agree?  "
```

Even if we do all he cleanup to this line as we did before, when we split it into separate words with spaces, the punctuation is going to be part of the words.

```
hello.rstrip().lstrip().lower().split(" ")
# ['hello,',
#  'what',
#  'a',
#  'great',
#  'day!',
#  "don't",
#  'you',
#  'agree?']
```

One thing we could do is do an additional cleaning pass with `re.sub()`, and remove any character that is not a word character or a space.

```
hello_clean = hello.rstrip().lstrip().lower()
hello_no_punct = re.sub(r'[^\w ]', '', hello_clean)
hello_no_punct
# 'hello what a great day dont you agree'

hello_no_punct.split(" ")
# ['hello',
#  'what',
#  'a',
#  'great',
#  'day',
#  'dont',
#  'you',
#  'agree']
```

Or we could use `re.split()` to split up the string with a regular expression in the first place.

```
re.split(r'\W', hello_clean)
# ['hello', '', 'what', 'a', 'great', 'day', '', 'don', 't', 'you', 'agree', '']
```