Two Hour Tour

So you're curious about Python but new to programming and unsure how it might help you in your research. A great place to start! This lesson is designed to be a 2 hour hands on tour of how Python can be used as a utility for processing text, particularly for humanities/literature projects. Two hours isn't sufficient to give you every essential piece of knowledge, but it is enough time to better understand if it might be something useful to keep learning.

This means that we're going to dive right in and explore the basics as we're going. Expect to not understand every piece of code you see here, and get comfortable with that. Remember that we're going for a tour here rather than full tutorial. You'd rather not invest days into learning Python only to realize that it isn't something useful for your projects, right? But two hours is enough to explore some of the patterns and get some tools.

Install stuff

This lesson is designed to be platform independent, so long as you can execute a python script. An IDLE, IPython, or other command line interpreter will not be as straight forward, but could be made to work. Jupyter notebooks or any other python IDE or environment should be just fine.

I've also set up an online tool meant for use with this lesson. So no istallation required!

Installation links are provided for those who desire it.

Use the repl

https://repl.it/JQD9/3

- click the link
- make a free account
- log into your account (you may need to hit the link again)

I believe you should be able to use it without an account, but making an account will let you save your work.

You can install things on your computer

You can follow the Anaconda and PyCharm instructions here: https://github.com/elliewix/Ways-Of-Installing-Python/blob/master/ways-of-installing.md (NB: these need updating butare about 90% still OK, 2018-06 -EW)

1. Install anaconda: link
2. Install PyCharm: link
3. Test your anaconda installation: link
4. Connect your PyCharm to anaconda: link

You can't install things on your computer

There are a list of alternatives here: link

I've also made a repl to go along with this course that has a short base script in it to start with: https://repl.it/JQD9/3 This does presume that you have steady internet access for the workshop, but easier than installing things.

Below is a link with the text file but not the base script, which I'll be using as part of a specific workshop.

DHOxSS18: Click this Repl link: CLICK ME

The core starting place: reading in data

We're going to jump right into a large block of code used to read in the contents of a text file. Don't worry! We're going to discuss it in chunks, but there are still going to be chunks you won't (and don't need to) understand.

Here's the block of code to start with. You should have this in your repl to run, or you can type it into your Python interpreter.

This will be our first instance of writing a script. This means that we write a series of directions (via code), and tell Python to go through them and execute those directions.

```
with open('pettigrew_letters_ORIGINAL.txt', 'r') as file_in:
    text = file_in.read()

print(len(text))
```

```
110828
```

Key concept: what is a script and how to run them

This block of code is actually a complete Python script. This means you can save this block of text and have python execute it from start to finish. You can do this inside of the repl you have access to. That is a python

script and when you click "run", it executes the script and prints out the results to the screen to the right.

Start first by looking at the output.

Question: What does the output `110828` represent?

The are a few key things that we can recognize in the code there: * the file name * and then there's a collection of clues like variable names that say `file_in` and methods & functions that say `read` and `len`.
* looking at the output number and scanning the contents of the file, we can see that it is too long to be the number of pages or lines in the text, and likely too big to be a representation of the size (unless I've done something strange with the units).

Perhaps this is the number of characters in the document?

But at this point, we don't even know what this `text` thing is. Let's just print out the text to start.l

Key concept: objects in memory (such as a string) hold content, but can also be asked questions about that content.

Key concept: `.read()` will read in the entire contents of a file and save it to a variable

```
print(text[:1000]) # for the purposes of this notebook, I've truncated the print to be
```

```
Pettigrew papers from the Beinecke Rare Book and Manuscript Library
Series OSB MSS 113

Transcribed by Gabriel Moshenska (gmoshenska@yahoo.co.uk) and released under a CC-By li
Moshenska, G. 2012. Selected Correspondence from the papers of Thomas Pettigrew (1791-1

Letters are listed by location (box, folder) and are arranged alphabetically by writer.

For further details see collection catalogue: http://drs.library.yale.edu:8083/fedora/g



List of letters:

Box 1, folder 10, William Francis Ainsworth to TJP, 17 December 1845
Box 1, folder 12, John Yonge Akerman to TJP, 13 November 1850
Box 1, folder 12, John Yonge Akerman to TJP, 7 March 1857
Box 1, folder 16, Thomas Amyot to TJP, 23 March 18??
Box 1, folder 16, Th
```

Key concept: you need to print stuff to your screen so your human eyes can view the contents

We say that we "print" something to the screen for two reasons: * that's what the function name is called, so that kind of makes sense * computers didn't originally have electronic screens like we know now, so they literally printed stuff onto paper, and the name has stuck

You'll need to use the `print` function to print things to the screen to make them appear to you, the human. Python "sees" (knows about) the contents just fine, but you have to tell the computer what you want to appear on the screen. NB: things get a bit different with this when you're in the interactive/interpreter environment.

key concept: you "pass" functions values for them to take as input, and they do something with that or to that value. What it does depends on the function

So let's print out the contents of our `text` variable. We do this by placing our variable name inside of the `()` for print. That is actually called "passing". We pass functions values inside their (). You may also hear this referred to as parameters. For our purposes, you can consider these synonyms.

At the bottom of your script, leaving everything else alone, add this line: `print(text)`

Key concept: variables hold values for things and become placeholders for us to act on those values or data objects

- Question: This is a variable that holds a value. What is the current value of this variable?
- Answer: The text contents of the 'pettigrew.txt' file.

We're going to expand on the idea of variables as we go.

Let's back up for a moment and talk about the rest of the block of code you've just run

```
with open('pettigrew_letters_ORIGINAL.txt', 'r') as file_in:
    text = file_in.read()
```

This block can be read as "read in the file `pettigrew_letters_ORIGINAL.txt` as text, and save the entire contents to the variable called `text`. Pinning down exactly which parts of that sentence match up to the code is not exactly a 1:1 activity, but we can look at some of the core elements of this pattern.

Python is a very high level language, meaning that it is further removed from 'computer speak' than many languages. However, the act of reading in files is something with a lot of nuance and moving pieces that it usually exposes some of the lower level components of any language.

Key concepts: know the parts of speech in Python syntax, including keywords, variables, punctuation, indent, functions, and methods

It is composed of:

- Keywords: `with` and `as`
    - these help construct the `with` block of code (keep reading for more about blocks) by providing a template for us to plug in our desired stuff.
    - imagine these as a madlib, "the ___ ate the ___" you fill in the subject and the object
    - so `with` and `as` provide a little mad lib for `with` (*open function for the thing you want*) as (*file handling variable*). You provide the contents of what you want to open, and provide the name of the file handling variable. This line is then finished with a `:` to indicate you are done opening the `with` block.
- the literal string values for the file name and the open mode
- the indent: see that white space on line two?
    - that extra white space is how you say that some lines of code belong to the `with` statement, and other subsequent lines (that are not indented) do not belong to that block. Imagine this a bit like how a copier collates documents when you print out many copies of a long document. The pages are physically shifted over to 'belong' with that stack.
    - some other languages use braces of various kinds to define how code relates to blocks, but in this case Python uses white space.
- variables: `file_in` and `text`
    - these items store values in them and represent a data type that can be acted on in certain ways. But they alo know things about themselves and can answer questions about their content or be asked to perform certain actions that are relevant to their content.
- functions and methods: `open()` and `.readlines()`
    - these are the things that end with `()` and take specific actions. You can call functions as part of a normal code, such as how you've seen `print`, `len`, and `open`.

key concept: piecing together the meaning of a block of code based on your understanding of the moving parts

Code is also not always ready by humans from left to right. We have to read through it in the order of operations (or what will be executed first). This takes time to develop an eye for, but the action mostly starts in the middle of lines when there's a lot of nesting (as in, there are a ton of `()` in there) and on the right if there's a lot of chaining (so there's a bunch of methods being called like

`stuff().stuff().morestuff()`.

So I read this block of code as:

- I'm opening the file (the `open()` function) in read mode (the `r` parameter tells me this)
- that open file object is being saved as `file_in`. This is the object that can be acted on the read the contents
    - there's a distinction between a file being "open for reading" versus "being read". The file itself doesn't know a thing about any of what we're doing in python, and the variable that we make aren't the file that we know and see on our system. They represent Python's knowledge of the file on your computer and Python's ability to interact with that kind of file and content.
    - being "open for reading" means that Python knows what the file is, knows it is there, and has an internal connection to it
    - "being read" means I'm actively accessing and processing the contents of that file
    - this is normally a single step in programs, so a bit funny when out of GUI context
- I'm calling `.read()` on the file open object (the thing that knows the file exists), so I know that the full contents of that file will be read and stored into the variable `text`
- I can see that the `with` syntax is being used here, so I don't need to call `file_in.close()` to close the connection to the file.
    - I know this because I know the `with` method because I also learned about it in a workshop

Learning how to read code is a skill that you develop over time as you learn to recognize components and patterns. Remember that you had to learn how to read a paragraph, and you did so by reading words into sentences, and sentences into narrative thought. Code is the same way, and you don't rarely read it in a linear process.

key concept: finding the data type of an object with `type()`

Next, change `print(text)` to `print(type(text))`. See the new `()` in there and how things are nested? We've added another function in the `()` of our previous function.

- Question: What does it tell you, and what do you think it means?
- Answer: it says `'str'` which stands for string. So this just means that the `text` object holds some text in it.

We asked a question to the object and it gave us an answer.

key concept: you can nest functions and values

Let's also take a moment to unpack the syntax we've seen of `print(len(text))` and

`print(type(text))`.

These parentheses `()` have significant meaning, in that they hold a value to pass to the function. This means that the function, in this case `print` takes an input of whatever you pass it. In the two cases with `len` and `type` they are also functions, but they are being nested within another function.

```
with open('pettigrew_letters_ORIGINAL.txt', 'r') as file_in:
  text = file_in.readlines()

print(len(text))
print(type(text))
```

```
1911
<class 'list'>
```

Our two values have now changed. What do you think the `1911` stands for?

How can you check?

Answers:

It represents the number of lines found within the text file.

key concept: use `.readlines()` to read in a text file and store each line (as defined by a newline character (`\n`) as a string inside of a list

Add `print(text)` to the bottom of your script and rerun it.

```
print(text[:50]) # for the purposes of keeping this notebook small, I've limited this t
```

```
['Pettigrew papers from the Beinecke Rare Book and Manuscript Library \n', 'Series OSB
```

What can we observe about this output?

- the text file content is there
- but it is broken up by line
- the lines all appear in the correct order that matches the original file
- none of the line breaks are printing
- there's the `\n` character showing up (there's our newline, it's a text character!)
- each line is surrounded by `''`

- all the lines are contained within `[]` (this is our list, as you can see in the `print(type(text))` output

key concept: lists are containers (bound by square brackets `[]`) that can store values and objects and have a specific order

key concept: there are several ways to read in files, and you can use them by calling the different read methods on the file reading object. Those read and parse actions are performed on the file every time the script is run.

So let's hone in on the piece of code that we changed.

`file_in.read()` changed to `file_in.readlines()`

key concept: the `.` (read as: 'dot') notation allows us to call methods/functions on specific objects. That object "knows" that it can do certain tasks, and you access those tasks via a `.` followed by the name of the method.

Our `file_in` variable remained the same, but we changed the action that we took on it. More specifically, we called `readlines()` on it rather than `read()`. The `.` there matters a lot (R people, it doesn't work this way!! Prepare your bodies.).

key concept: some functions will work on many kinds of objects, but they usually act differently when they do. You will be asking the same questions, but the object will be answering it its own way.

We can also note that our functions of `len` and `type` still worked on `text` even though the content and data type of `text` has changed. We can further observe that `len` will out the number of characters in a string because strings count their length by the number of characters stored within.

Let's change this one more time: change `file_in.readlines()` to `file_in.readline()` (so you're taking the s off and making it singular). Now run your script again.

```
with open('pettigrew_letters_ORIGINAL.txt', 'r') as file_in:
  text = file_in.readline()

print(len(text))
print(type(text))
```

```
69
<class 'str'>
```

Hmm, that's much shorter. What do you think happened? Let's print out `text` to investigate.

```
print(text)
```

```
Pettigrew papers from the Beinecke Rare Book and Manuscript Library
```

What can we observe about this output?

- Just one line of text
- the data type is back to a string
- look closely at the output, there's an extra line in there and the `\n` is gone. This indicates that the `\n` was rendered by the print statement instead of being printed out in the raw format.

key concept: the `\n` character is the newline character. When rendered on the screen it will look like an empty line, but will appear as `\n` elsewhere (when the text is being printed in 'raw' form. It can also be directly typed in using `\n` .

```
with open('pettigrew_letters_ORIGINAL.txt', 'r') as file_in:
    text = file_in.readline()
    text = file_in.readline()

print(len(text))
print(type(text))
```

```
19
<class 'str'>
```

```
print(text)
```

```
Series OSB MSS 113
```

Take a look at the actual text file. What do you think has happened here?

Answer: The first line was read in, as before, and assigned to `text` . Then the second line was read in, and also assigned to the variable `text` . This overwrote the original value.

So add the numbers 1 and 2 to the `text` variable name so you have `text1` and `text2` . Now just have `print(text1)` and `print(text2)` .

key concept: `.readline()` will read one line at a time, which you can choose to only temporarily store. Now add in another `file_in.readline()`. Yup, copy/paste in another line of it and run the script again.

```
with open('pettigrew_letters_ORIGINAL.txt', 'r') as file_in:
    text1 = file_in.readline()
    text2 = file_in.readline()

print(text1)
print(text2)
```

```
Pettigrew papers from the Beinecke Rare Book and Manuscript Library

Series OSB MSS 113
```

Why is knowing about `.readline()` valuable?

This function really shows off the power and flexibility of working with code. You are used to opening and reading in an entire document. So nothing really shocking with the other two methods of reading in a file.

You may find yourself is a situation where a data file is too big to open all at once into your computer's memory. Alternatively, you may have a file that will read in, but will take a really long time to do so, when you really just want to preview a littly bit of it.

Module summary

- how to run a python script
- even though you may not understand everything in this block of code, you can know enough about what it is doing to make it useful.
- you can change around certain 'action' areas of the template code to customize what you want it to do
- you've explored several key methods of reading in text files.

Part 2: Let work with the file content now

Selecting the reading in method for your file is a personal/stylistic choice. You can usually get away with using any of the three methods (`.read()` , `.readlines()` , or `.readline()`) but some will require more fussing than others depending on the task. Your best bet is to try and match the task with the tool, but this is something you develop over time.

The Pettigrew file contains the plain text transcription of correspondence from Pettigrew's files. These letters are identified with a box number and a folder number. We presume that together these should create a unique identifier for each letter, but we need to investigate this.

The file also has a manifest at the top that reports out which letters (again as specified by the corresponding box/folder numbers).

Here's the question: are all the letters reported in the manifest actually there?

So here's two tasks:

1. check if all the letters reported in the manifest appear in the actual file.
2. check if all the letters in the file are reported in the manifest

Discussion: what are some approaches you can think of? Take a moment outside of this notebook and look through the text file.

Key facts:

- Our most common unit of data is the line of text. This means that each line reporting the box/folder numbers are contained on a single line of text.
    - This tells us that reading in the file via `.readlines()` would be good
- Each line that is reporting a letter location starts with the literal word "Box" then followed by a number
- No line reporting a letter location has any other information
- A mix of "folder" and "folders" is used in the table of contents
- All location lines consistently use Box (e.g. Boxes does not appear)

Once we start with these clues, we can then seek out how to solve the problem in Python.

Our next task is to think about what our end points should be. Let's say we want to construct two variables: a list of every location line from the manifest, and a list of every location line within the file proper.

So we begin to move toward our goal!

Step 1: read in the file using `.readlines()`

We know this code well by now!

```
with open('pettigrew_letters_ORIGINAL.txt', 'r') as file_in:
    text = file_in.readlines()

print(len(text))
```

```
1911
```

We can look in our file and see that there are 1911 lines in the file, so looks good so far!

Working with lists

We've got our data stored in a list, and we need to learn how to perform inquiries into a list.

Recall that lists have a specific order about them, so we can look up the values by those places. This is called the index position. Index positions start at 0. No, I'm not going to defend this. But it is reality that we have to work with.

The basic syntax for looking up an item in a list by the index position is:

`list_name[position_number]`

Take a moment to experiment with this in your script.

Key concept: you can provide an index number to look up the value of something in a list based on that position

Here's some starting points:

- `text[0]` will give you the first line
- `text[4]` will give you the fifth line (I know)
- `text[-1]` will give you the last line

Try it yourself! Look in the data file and pick out a line you want to see. Change your program to print out just that line.

Key concept: integer numbers are a data type, and they represent whole numbers either positive or negative

Something to note about these numbers is that they are just the plain numbers. No quotes or anything else. These are what are called integer numbers (for python), or whole numbers (to humans).

```
print("Line 1 is:", text[0])
print("Line 5 is:", text[4])
print("The last line is:", text[-1])
```

```
Line 1 is: Pettigrew papers from the Beinecke Rare Book and Manuscript Library

Line 5 is: Moshenska, G. 2012. Selected Correspondence from the papers of Thomas Pettig

The last line is: 18th June 1833
```

Take a look at the contents of the `print` functions there. I've passed in a string as well to improve readability. This means that you can pass multiple things into the `print` function! You actually can pass as many things as you want in there, just separate them with a comma. You can even mix data types, such as strings and numbers. Bonus, it'll automatically put that space in there for you.

key concept: you can pass multiple things into a single `print()` function to have them all print in one line.

Step 2: Use list slicing to get the manifest list out

Sometimes we also want to get groups of elements out of a list. We call this "slicing". We can take a "slice" from a list using the following notation, which we'll unpack.

This is one of those syntax patterns **worth just writing down somewhere prominent and referencing a lot.**

```
my_list[inclusive_start_position:exclusive_stop_position:optional_step_amount]
```

A few things to note about the punctuation here:

- we add a set of square brackets `[]` after our list's variable name
- then we add our position numbers in separated by a colon character `:`

Now, let's unpack the names:

- inclusive**start**position: this position number will be included, so if I say 4, then the item at position 4 will be included
- exclusive**stop**position: this position number will NOT be included, so if I say 10, then it will go up to but not include item 10, so stopping at including 9.
- optional**step**amount: this is an optional thing to include, and it's just the number of items to move forward, e.g. if I said 2, it would give me every other item in the list

key concept: you can slice out elements within a list using the start:stop:step notation

Technically speaking, all these numbers are optional. This statement will make more sense with examples. Take some time to play with these in pairs. Fun fact: you can use the negative position numbers in here, what will that do?

```
print(text[0:3]) # will print a list with lines 0, 1, and 2 but NOT 3
```

```
['Pettigrew papers from the Beinecke Rare Book and Manuscript Library \n', 'Series OSB
```

```
print(text[5:6]) # will print a list with only line 5
```

```
['\n']
```

```
print(text[0:10:2]) # will print a list with lines 0, 2, 4, 6, and 8
```

```
['Pettigrew papers from the Beinecke Rare Book and Manuscript Library \n', '\n', 'Moshe
```

Now that we can slice out stuff, how about we go after the letter manifest?

There are programmatic ways to sort out where it begins and ends, but we can also just hard code it.

Challenge: look through the file with a partner and determine the list slice that will give you the manifest. Save that new list to a variable called `manifest_list` and print the length. You should end up with a list of lenth 145.

```
manifest_list = text[15:159]
print(len(manifest_list))
```

```
144
```

Step 3: loop over the list of manifest letters

Now that we have our data in a collection, we can loop through it. This means that we tell Python to traverse that list of data, giving us access to each data point one at a time.

Accessing the data one at a time allows us to write a short piece of code that is generalized across the entire loop. We will use an iterable variable to reference the changing data items (remember, one at a time) inside this loop.

Since this is a loop over a specific set of items, it is a definite loop. Meaning that know (or can know) exactly how many times it should run.

There are two parts to a for loop:

1. The setup (remember how we had the setup line for the `with` statement?) where you define what you're looping over and the name of the iterable variable.
2. The block of code you'd like executed for each item in that collection or sequence.

key concept: for loops allow you to act on all the items in a collection, one at a time

So I already have my list of letters in the manifest, called `manifest_list`. I need to pick out the name of my iterable variable. This will hold the location value of each letter in succession. So let's call it `location`. Thus, the opening line for my for loop would be:

```
for location in manifest_list:
```

See the madlib there? `for (iterable_variable) in (list or other sequence):`. I need to provide the thing to iterate over, and then name what the iterable variable should be called.

The iterable variable will be created during the first run of the loop. Don't worry, it's completely fine to not have ever mentioned this variable anywhere else. In fact, you should always try to come up with unique iterable variable names for each loop. You can accidentally erase an existing variable's value if you reuse that name. This variable will also persist after the for loop is done, holding the last value it saw before exiting from the for loop. Occasionally this is nice to have, but usually you leave them alone after.

key concept: iterable variables are created during the for loop, but will continue to exist after. Don't reuse variable names!

Interesting to note here is that we don't need to tell Python what to iterate over. The item we're looping over knows how it wants to be looped over and tells the for loop. So you can trust Python to sort itself out, but it means that items may be looped over in unexpected ways and you should always look it up or test it out when starting up your loop.

key concept: the for loop itself will determine how to loop over the sequence object that you give it, which may be unexpected. So always test your loop with a print statement before going further

Let's just set up a basic for loop, and in fact this is how I suggest you start every for loop. Get the basic set up and just print out your iterable variable to confirm that you

```
for location in manifest_list[:10]: # to keep the notebook short I'm limiting to the fi
    print(location)
```

```
Box 1, folder 12, John Yonge Akerman to TJP, 13 November 1850

Box 1, folder 12, John Yonge Akerman to TJP, 7 March 1857

Box 1, folder 16, Thomas Amyot to TJP, 23 March 18??

Box 1, folder 16, Thomas Amyot to TJP, 20 August 18??

Box 1, folder 16, Thomas Amyot to TJP, n.d.

Box 1, folder 16, Thomas Amyot to TJP, 24 August 18??

Box 1, folder 20, Francis Arundale to TJP, n.d.

Box 1, folder 27, Robert John Eden Auckland to TJP, 20 June 18??

Box 1, folder 28, William Ayrton to TJP, 1 February 1827

Box 1, folder 29, Benjamin Guy Babbington to TJP, 10 June 18??
```

See that extra white space between each line? That's the `\n` character being rendered. Let's get rid of that and explore messing with strings.

We can add the `.strip()` method to a string to remove whitespace. Example:

```
print("  fizzy pop sandwich .           ".strip())
```

```
fizzy pop sandwich .
```

So all the white space on the left and right was removed, but it stopped when it saw the `.` character in there. `.strip()` does not remove white space from inside a string. Let's now apply this to our location lines in our for loop.

key concept: strings have many methods, and `.strip()` is useful to clean whitespace off

```
for location in manifest_list[:10]: # to keep the notebook short I'm limiting to the fi
    print(location.strip())
```

```
Box 1, folder 12, John Yonge Akerman to TJP, 13 November 1850
Box 1, folder 12, John Yonge Akerman to TJP, 7 March 1857
Box 1, folder 16, Thomas Amyot to TJP, 23 March 18??
Box 1, folder 16, Thomas Amyot to TJP, 20 August 18??
Box 1, folder 16, Thomas Amyot to TJP, n.d.
Box 1, folder 16, Thomas Amyot to TJP, 24 August 18??
Box 1, folder 20, Francis Arundale to TJP, n.d.
Box 1, folder 27, Robert John Eden Auckland to TJP, 20 June 18??
Box 1, folder 28, William Ayrton to TJP, 1 February 1827
Box 1, folder 29, Benjamin Guy Babbington to TJP, 10 June 18??
```

Much cleaner! See how we're applying it to the iterable variable there? That variable will become the individual strings inside the list, so we can treat it like a string. Again, this is a pretty opaque process that's happening in the background, which is why I say you should test things out first to ensure that you're looping over what you expect.

key concept: the iterable variable will have the data type of whatever the element is within the sequence you are iterating over

Now we can start using these data points.

Step 4: Gather all the locations in the full document

Let's go ahead and slice out the rest of our document (and thus skip over the metadata and manifest area.

Challenge: If we grabbed the manifest up until line 159, what list slice can we do to text to get everything after that until the end?

Answer, you can omit the start or stop positions in a slice and it will presume that you want the start or finish (respectively). So you can say `[159:]` to go from 159 to the end of the list. Let's call this list `letters`.

```
letters = text[159:]
print(len(letters))
```

```
1752
```

We have a list of 1752 lines of text. How can we use the known pattern to detect which lines are referencing the box locations? We know that they are all in a format like "Box (number), folder (number)" with some variation in the comma and the word folder is sometimes plural.

Going back to my advice, let's start first with just setting up a basic for loop. Given that the lines of text represent many things in this loop, we're going to use a more generic iterable variable.

```
for text_line in letters[:50]:
    print(text_line.strip())
```

```
Box 1, folder 10
William Francis Ainsworth to TJP


My Dear Sir,
I only received your favour of the 16th late last night, on my return from the Syro-Egy
I shall be very happy to meet Mr Wright at the Archaeological meeting this evening, the
Believe me,
My dear sir,
Yours very sincerely,
W. Francis Ainsworth
Wednesday Morn
Dec 17th 1845


Box 1, folder 12
John Yonge Akerman to TJP


My dear sir,
I enclose you the copy of the letter of Nelson of which I spoke.  You may rest assured
Yours faithfully,
J.Y. Akerman
S.A.
13. Nov. 1850


My dear sir,
I send you the required memorandum of the Coins, which I think you will find correct.
Very truly yours,
J.Y. Akerman
S.A.
7. Mar. 1857.
```

```
Box 1, folder 16
Thomas Amyot (proposed TJP as FSA)


Athenaeum, Sunday
23rd March
My dear Pettigrew,
On coming here, I have found your kind Note with its printed inclosure.  I thank you fo
With the kindest regard,
I remain always,
My dear Pettigrew,
Yours most sincerely,
```

There are several methods of trying to detect this kind of thing, but one of the easiest to start with is another string method: `.startswith()`. We call this method on a string, pass it another string to check, and it will say True or False. Let's test this out with something smaller:

```
print("fizzy pop sandwich".startswith("fizzy"))
print("Box 1, folder 10".startswith("Box "))
print("Box 1, folder 10".startswith("box "))
```

```
True
True
False
```

Not several things:

- we can call this directly onto a string, but could have done it on a variable storing a string
- we can pass it a string with many letters
- you can include whitespace as part of the match string
- case matters

key concept: use `.startswith()` and `.endswith()` to check the beginning/end content of a string. this will return a True or False result.

Now that we have a tool to give us a boolean result for matches, we can bust out a logical check to filter out our results.

But let's take a moment to sketch out the flow of what we want to happen here.

```
* loop over the list of lines:
    * strip the line
    * check if that stripped line starts with "Box "
        * True:  print it out! We'll want to do more later, but start with printing
        * False: move along, nothing to see here
```

key concept: sketch out what you want to happen in a program when you start adding elements like a conditional check. Make sure you think about how each condition should be handled.

Now we can construct this. We're going to start with just ensuring that `.startswith()` is acting as we expect. We're going to call it on the stripped line variable. We expect to see more `False` than `True` here.

```
for text_line in letters[:25]:
    stripped_line = text_line.strip() # saving to a variable!
    print(stripped_line.startswith("Box "))
```

```
False
False
False
True
False
False
False
False
False
False
False
False
False
False
False
False
False
False
True
False
False
False
False
False
False
False
```

We can't quite tell what's happening here because we can't see the line anymore. So let's also print out the line and review the results.

key concept: design your programs step by step, printing out your results to check that things are working according to your hopes and dreams

```
for text_line in letters[:25]:
    stripped_line = text_line.strip() # saving to a variable!
    print(stripped_line.startswith("Box "), stripped_line)
```

```
False
False
False
True Box 1, folder 10
False William Francis Ainsworth to TJP
False
False
False My Dear Sir,
False I only received your favour of the 16th late last night, on my return from the Sy
False I shall be very happy to meet Mr Wright at the Archaeological meeting this evenir
False Believe me,
False My dear sir,
False Yours very sincerely,
False W. Francis Ainsworth
False Wednesday Morn
False Dec 17th 1845
False
False
True Box 1, folder 12
False John Yonge Akerman to TJP
False
False
False My dear sir,
False I enclose you the copy of the letter of Nelson of which I spoke.  You may rest as
False Yours faithfully,
```

Some of these lines are just newline characters, so it looks like nothing is printing and that's fine!

Do we see anything here that's concerning? I don't, but the results are pretty cluttered with `False` results.

So let's now add in our if/else statement.

```
for text_line in letters[:25]:
    stripped_line = text_line.strip() # saving to a variable!
    is_box = stripped_line.startswith("Box ")
    if is_box == True:
        print(stripped_line)
    else:
        print("---")
```

```
---
---
---
Box 1, folder 10
---
---
---
---
---
---
---
---
---
---
---
---
---
Box 1, folder 12
---
---
---
---
---
---
```

That's a bit easier to read! Let's explore the syntax here:

We're using this `==` symbol which conducts the logical check.

Again, we've got one of our madlib lines opening it up, followed by indented code under it creating a block of code we can call the "if block". An oddity to discuss is the "else" block that appears below it, which is considered part of the `if` block. It may only appear within the presence of an if block, but the if block may appear alone.

There's also this 'elif' block we can add in. So let's reimagine our logical check:

```
* if the line starts with "Box "
    * then print it
* else, if the line is only \n
    * then ignore it entirely
* otherwise, print out "---"
```

```
for text_line in letters[:25]:
    stripped_line = text_line.strip() # saving to a variable!
    is_box = stripped_line.startswith("Box ")
    if is_box == True:
        print(stripped_line)
    elif text_line == '\n': #remember that stripped lines will remove \n
        print("blank line")
    else:
        print("---", stripped_line)
```

```
blank line
blank line
blank line
Box 1, folder 10
--- William Francis Ainsworth to TJP
blank line
blank line
--- My Dear Sir,
--- I only received your favour of the 16th late last night, on my return from the Syr
--- I shall be very happy to meet Mr Wright at the Archaeological meeting this evening
--- Believe me,
--- My dear sir,
--- Yours very sincerely,
--- W. Francis Ainsworth
--- Wednesday Morn
--- Dec 17th 1845
blank line
blank line
Box 1, folder 12
--- John Yonge Akerman to TJP
blank line
blank line
--- My dear sir,
--- I enclose you the copy of the letter of Nelson of which I spoke.  You may rest assu
--- Yours faithfully,
```

We can reduce our results even further by just ignoring any blank lines. We can use the `continue` keyword to make the for loop restart.

```
for text_line in letters[:25]:
    stripped_line = text_line.strip() # saving to a variable!
    is_box = stripped_line.startswith("Box ")
    if is_box == True:
        print(stripped_line)
    elif text_line == '\n': #remember that stripped lines will remove \n
        continue
    else:
        print("---", stripped_line)
```

```
Box 1, folder 10
--- William Francis Ainsworth to TJP
--- My Dear Sir,
--- I only received your favour of the 16th late last night, on my return from the Syr
--- I shall be very happy to meet Mr Wright at the Archaeological meeting this evening
--- Believe me,
--- My dear sir,
--- Yours very sincerely,
--- W. Francis Ainsworth
--- Wednesday Morn
--- Dec 17th 1845
Box 1, folder 12
--- John Yonge Akerman to TJP
--- My dear sir,
--- I enclose you the copy of the letter of Nelson of which I spoke.  You may rest assu
--- Yours faithfully,
```

Now we have a better visual result to check our data. Once we review that our False results are all indeed false, or at least most of our results look fine, let's go ahead and only print out the True lines.

```
for text_line in letters:
    stripped_line = text_line.strip() # saving to a variable!
    is_box = stripped_line.startswith("Box ")
    if is_box == True:
        print(stripped_line)
    elif text_line == '\n': #remember that stripped lines will remove \n
        continue
    else:
        continue #print("---", stripped_line)
```

```
Box 1, folder 10
Box 1, folder 12
Box 1, folder 16
Box 1, folder 20
Box 1, folder 27
Box 1, folder 28
Box 1, folder 29
Box 1, folder 31
Box 1, folder 33
Box 1, folder 37
Box 1, folder 39
Box 1, folder 51
```

Box 2, folder 55
Box 2, folder 62
Box 2, folder 65
Box 2, folder 86
Box 2, folder 89
Box 2, folder 92
Box 3, folder 104
Box 3, folder 109
Box 3, folder 114
Box 3, folders 119
Box 3, folder 120
Box 3, folder 131
Box 4, folder 135
Box 4, folder 136
Box 4, folder 142
Box 4, folder 144
Box 4, folder 148
Box 4, folder 157
Box 4, folder 158
Box 4, folder 175
Box 5, folder 202
Box 5, folder 212
Box 5, folder 223
Box 5, folder 227
Box 5, folder 232
Box 5, folder 241
Box 6 folder 257
Box 6 folder 262
Box 6 folder 263
Box 6, folder 270
Box 6, folder 273
Box 6, folder 274
Box 6, folder 275
Box 7, folder 309
Box 7 folder 310
Box 7 folder 317
Box 7, folder 326
Box 7, folder 328
Box 7, folder 330
Box 7, folder 332
Box 7, folder 333
Box 7, folder 342
Box 8, folder 374
Box 8, folder 378
Box 8, folder 379

```
Box 9, folder 389
Box 9, folder 392
Box 9, folder 393
Box 9, folder 399
Box 9, folder 405
Box 9, folder 411
Box 9, folder 412
Box 9 folder 417
Box 9, folder 421
Box 9, folder 431
Box 9, folder 442
Box 10, folder 454
Box 10, folder 458
Box 10, folder 496
Box 10, folder 498
Box 11, folder 504
Box 11 folder 510
Box 11, folder 514
Box 11, folder 523
Box 11, folder 526
Box 11 folder 528
Box 11, folder 529
Box 11 folder 530
Box 11, folder 537
Box 11 folder 542
Box 12 folder 553
Box 12, folder 555
Box 12 folder 562
Box 12, folder 569
Box 13, folder 591
Box 13 folder 594
Box 13, folder 597
Box 13, folder 598
Box 13, folder 599
Box 13, folder 601
Box 13, folder 607
Box 13, folder 609
```

See how I just commented out that line? That can help us experiment with tweaks.

key concepts: use logical checks to filter out results.

Let's also unpack that syntax a bit. We've been used to seeing the madlib starts to blocks with just one, but there's in fact three happening now.

```
if something:
    do this
elif something:
    do this
else:
    do this
```

This is all one big if block. The elif and the else belong to the opening if line. Both are optional, but neither may appear without the if statement to start. elif always comes after if and else always comes at the end. Think about it like a multi-book series:

- `if` is always the first book, which can sometimes stand alone.
- `elif` is the middle book, and you can have unlimited sequels, but they may only come after the opening `if`
- `else` may only happen once and always at the end, is always your conclusion or series finale.

key concept: if blocks may contain many elif statements or one else statement

Step 5: let's collect the positive results

Now that we have a filter that is finding just the lines that we want, we can collect them in a new list. This is a basic accumulator pattern, which has two stages:

1. Start with something that is empty, in this case we can start with an empty list: `[]`
2. Add what you want to that thing! In this case, we can use the `.append()` method to add things to our list

```
letter_locations_to_collect = [] #here's our empty list

for text_line in letters:
    stripped_line = text_line.strip()
    is_box = stripped_line.startswith("Box ")
    if is_box == True:
#         print(stripped_line)
        letter_locations_to_collect.append(stripped_line) # here's step 2 of adding it
    elif text_line == '\n':
        continue
    else:
        continue #print("---", stripped_line)
```

```
print(letter_locations_to_collect)
```

```
['Box 1, folder 10', 'Box 1, folder 12', 'Box 1, folder 16', 'Box 1, folder 20', 'Box 1
```

Step 6: See what we've got and wrapup

We've accomplished two things:

1. Collect all the reported letters from the manifest
2. Collect all the reported letters from the transcription content

Theoretically, these should have the same number. How can we check that?

```
print(len(letter_locations_to_collect))
print(len(manifest_list))
```

```
94
144
```

Hmm, so they don't!

But this is where we need to stop.

Now you've seen the basics of processing text data in python and a few methods of handling both. Hopefully this has given you a taste for the kind of tasks that something like Python can help you with and a starting point for doing more with it in the future.

```
```