Notebook

February 7, 2019

1 Files and How Computers Represent Data

In this lesson, we're going to learn how to open files and work with data from the disk. We'll start with the mechanical process of opening text files, and then move on to learn a little bit more about different kinds of data you'll see.

Here's the basic method of opening and reading text files. Suppose I have a file called hello.txt in my working directory. (Your working directory is the directory you run Python from on your hard drive. For those of you using Azure Notebooks, this should be your library, but talk to me if you see a file there and can't read it from Python.)

```
In [1]: with open("hello.txt", 'r') as my_awesome_file:
            hello = my_awesome_file.read()
        print(hello)
```

```
hello, I am a text file
```

Let's break down that code. The first line starting with `with` is known as a *context manager*. Any time you see `with` in Python, what you should think is "this is going to change what's going on in my computer for the duration of this block."

In this case, you can pretty much read exactly what's going to happen like it's English rather than Python. For the duration of the indented block below that first line, all the code in there is going to be executed `with` the file `hello.text` being `open` and hence available for reading. Inside that block, the name `my_awesome_file` is going to be assigned to the open file (that's what the `as` statement does). The second parameter to the `open` function, the `'r'`, just indicates that you're going to open it for reading—instead of, for example, opening it for writing, in which case the `r` would change to a `w`. You can also open a file to append with `a`—to add more data to the bottom of the file.

The thing that's assigned to `my_open_file` is called a *file handle*, it's just a normal Python object that gives you access to the data inside. In this case, that object has a method, `read()`, that gives you the contents of the file as a string. Which we then printed.

Let's look at writing and appending.

```
In [2]: with open("hello.txt", "w") as still_my_file:
            still_my_file.write("I contain something different now!!")

        with open("hello.txt", 'r') as my_awesome_file:
            hello = my_awesome_file.read()
        print(hello)
```

```
I contain something different now!!
```

You see that when we open a file for writing, we overwrite what was already there. What if you don't want to do that?

```
In [3]: with open("hello.txt", "a") as my_file:
            my_file.write("Hi again!")
            my_file.write("Hi human user!")

        with open("hello.txt", 'r') as my_awesome_file:
            hello = my_awesome_file.read()
        print(hello)
```

```
I contain something different now!!Hi again!Hi human user!
```

Now suppose we want to put more lines in the file, and then read the file line by line into a list. We could do that, in the first case, by adding the special character `'\n'`, and in the second case, by using the `readlines()` method of the file object.

```
In [4]: with open("hello.txt", "a") as my_file:
            my_file.write("\nHere's a newline.  ")
            my_file.write("But:\nNewlines don't have to be at the start of a string.")

        with open("hello.txt", 'r') as my_awesome_file:
            hello = my_awesome_file.read()
        print(hello)
```

```
I contain something different now!!Hi again!Hi human user!
Here's a newline.  But:
Newlines don't have to be at the start of a string.
```

```
In [5]: with open("hello.txt", 'r') as my_awesome_file:
            hellolist = my_awesome_file.readlines()
        print(hellolist)
```

```
['I contain something different now!!Hi again!Hi human user!\n', "Here's a newline.  But:\n", "N
```

As far as Python is concerned, there are two kinds of files that you might want to open: text files and binary files. If you want to open a binary file, you pass `'rb'` or `'wb'` to `open()` as the second parameter, depending on whether you want to read or write the binary file.

Pretty much all the files you'll be working with in this course will be text files: they'll be txt files, csv files (spreadsheet-like data stored in a plain text format), json files (key-value and list-like data stored in a plain text format), or the like. Here are some common binary file formats:

- zip files (compressed file archives)

- Microsoft Word docx files
- PDF files
- images of all kinds (with the exception of SVG files, which are fancy vector images that are stored as text

In almost every case, it'll make more sense to use a function from a library to open a binary file, rather than manipulate it directly. There are libraries to handle PDFs, Word files, and the like. So I won't belabor it here, but the difference between reading as text and reading as binary is one you need in your head for a moment (I'm about to break it).

For reading CSV files, the best choice is to use the *Pandas* library, which should be installed for you already, rather than the built-in Python CSV library. The latter is a bit obscurely organized.

```
In [6]: import pandas as pd
        mydata = pd.read_csv("rol-scores.csv")

In [7]: mydata.head()

Out[7]:          State  Pop. In Millions for 2012  RoLScore  elec_pros   pol_plur  \
        0     Albania                         3.2     42.60          8         10
        1   Argentina                        41.1     51.94         11         15
        2   Australia                        22.7     73.28         12         15
        3     Austria                         8.4     73.15         12         15
        4  Bangladesh                       154.7     31.57          9         11

           free_expr  assoc_org  per_auto       2012GDP  hprop  hfisc  hbiz  hlab  \
        0         13          8         9  1.264810e+10     30   92.6  81.0  49.0
        1         14         11        13  4.755020e+11     15   64.3  60.1  47.4
        2         16         12        15  1.532410e+12     90   66.4  95.5  83.5
        3         16         12        15  3.947080e+11     90   51.1  73.6  80.4
        4          9          8         9  1.163550e+11     20   72.7  68.0  51.9

           htra  hinv
        0  79.8    65
        1  67.6    40
        2  86.2    80
        3  86.8    85
        4  54.0    55
```

We'll spend some time in Pandas later on, but I wanted you to get a basic look at it now. We just opened a CSV file containing data from my rule of law book (shameless plug), The Rule of Law in the Real World.

Ok, so remember how I said that there's a difference between text and binary files? That's... not entirely honest. Computers ultimately store everything in binary, just as a bunch of ones and zeroes.

You won't have to work with the ones and zeroes directly. But you might have to worry about the layer of abstraction one level up. The way that Python mostly sees all kinds of files is as a series of *bytes*—eight-bit binary numbers. This includes text files! So, if we want, we can open up a text file as bytes. (Let's reset our text file to its original state first, so it's nice and short.)

```
In [8]: with open("hello.txt", 'w') as mf:
            mf.write("hello, I am a text file")
```

Now to open it as binary:

```
In [9]: with open("hello.txt", "rb") as binary_text:
            data = binary_text.read()
            print(data)
```

```
b'hello, I am a text file'
```

Not all that exciting, right? The b in front of the string just means that it's held in memory as bytes—but Python still knows it's a string and will happily print it out for you just like a text file. If you really want to see an approximation of what things look like under the hood . . .

```
In [10]: for x in data:
             print(x)
```

```
104
101
108
108
111
44
32
73
32
97
109
32
97
32
116
101
120
116
32
102
105
108
101
```

What's going on there? Well, when you asked for it to be binary, Python read the string as bytes, and the internal representation is a series of numbers. Those numbers correspond to letters. To get a string back out of it, you have to *decode* it:

```
In [11]: decoded = data.decode("utf-8")
         for x in decoded:
             print(x)
```

```
h
e
l
l
o
,

I

a
m

a

t
e
x
t

f
i
l
e
```

Why does this stuff all matter? Well, basically, it matters because languages other than English exist.

See, if strings are just sequences of numbers, there needs to be a rule on how to translate numbers to things humans can read and back. Humans can't read "104 101 108 108 111" as "hello" very easily!

Back in the early days of computing, this was easy but lazy. The only people that the people who made computers cared about, at least in the U.S., were English speakers. And so the main rule for how to translate numbers to letters—known as an "encoding"—was called "ASCII." It contained the entire alphabet that we're familiar with in the U.S., plus some punctuation, and represented it all as a series of numbers from 0 to 127. Here's a table of ASCII numbers to letters. You can still see that today. For example:

```
In [12]: chr(65)

Out[12]: 'A'
```

Read that as "give me the character at number 65."

But today, we don't use ASCII, and Python's internal representation of characters isn't ASCII. To prove that, let's look at some big numbers for characters!

```
In [13]: chr(128516)

Out[13]: ''
```

That's right. It's an emoji. We have emojis now. But, more importantly, we have letters from other languages. Like Japanese:

```
In [14]: chr(12365)

Out[14]: ''
```

The new text encoding that most things are in these days is called *UTF-8*. It's much, much, much bigger than ASCII, which is what allows us to have Japanese and emojis and more besides.

The catch, and why this is sometimes a problem, is that in addition to ASCII and UTF-8, there are lots of other character encodings floating around out there. For example, there's an encoding called Latin-1 that a bunch of Microsoft products have been known to use. Sometimes, you'll see data (like maybe saved as a CSV from some version of Excel) in that character encoding, and then everything will look funny. If you've ever opened a file with a bunch of question marks and weird black diamonds in place of characters you've expected to see (known as mojibake), that's because the program you used to open it thought it was a different character encoding than the program used to save it did.

For the most part, this shouldn't be a problem for you in this class, as long as you follow the following basic rules:

1. Use a version of Python numbered 3 or greater. Python 2.x had a lot of screwed up stuff with string encodings that Python 3 fixed.
2. Always open and save anything in UTF-8. Don't touch any other encoding, ever.
3. If you get unlucky enough to find something in another encoding, try Latin-1.

Also, ideally your system default coding is UTF-8 (UTF-16 will also do). You can check that with the following code:

```
In [15]: import sys
         sys.getdefaultencoding()

Out[15]: 'utf-8'
```

Finally, some terminology. Decoding means converting from binary to a string. Encoding means converting from a string to a binary with a particular encoding.

If you run into problems with this, usually involving Python complaining that a codec can't decode something, come talk to me and we'll dig into it!

```
In [ ]:
```