Lecture 9: Reading in data II

Custom reading from a file

Ok… Using argparse, it's easy to get the name of a file. But how do we open it up and read from it? We do this using the built-in command called open. Open takes in the name of the file (you can use a relative or absolute path) and then a mode (typically 'r' for reading or 'w' for writing). It returns a file IO object:

To read in the contents of a file, it's common to use the built-in iterator method. This breaks on new lines (i.e., each element is one line of a file) and returns each line as a string.

```
>>> f = open('WorldDemographics.csv')
>>> type(f)
<class '_io.TextIOWrapper'>
>>> dir(f)
['_CHUNK_SIZE', '__class__', '__del__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__lt__', '__ne__', '__new__', '__next__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '_checkClosed', '_checkReadable', '_checkSeekable',
'_checkWritable', '_finalizing', 'buffer', 'close', 'closed', 'detach',
'encoding', 'errors', 'fileno', 'flush', 'isatty', 'line_buffering', 'mode',
'name', 'newlines', 'read', 'readable', 'readline', 'readlines', 'reconfigure',
'seek', 'seekable', 'tell', 'truncate', 'writable', 'write', 'write_through',
'writelines']
>>> for line in f:
...     print(line)
...
,PopulationID,country_code,Level,continent_code,Age,#Alive

0,Afghanistan,4,Country,5501,0,1134501

1,Afghanistan,4,Country,5501,1,1134501

2,Afghanistan,4,Country,5501,2,1134501

3,Afghanistan,4,Country,5501,3,1124250

4,Afghanistan,4,Country,5501,4,1113998
```

One thing that is strange is that each line of the file is separated by a line. This is because Python includes the newline character when it reads in the line. You can see this as so:

```
>>> line
'20300,Zimbabwe,716,Country,910,100,19\n'
```

You can process each line using string methods:

```
>>> line.rstrip().split(',')
['20300', 'Zimbabwe', '716', 'Country', '910', '100', '19']
```

While this is a basic way to analyze data from a file and you might use it for common text files, in general this is NOT the approach you want to take

1. What is the format of the data I want to import into the Python program?
2. How do I want to store the data in my Python program? Data can then be stored in many different variables.

There are many defined file formats used for storing data. Some examples: fasta format to store DNA sequence data, html to store websites, pdb files to store protein sequence data…. I could go on and on. Some advice on programming: in general, you will always be served by adapting previously defined file formats. If you want to store information on a phylogenetic tree, use one of the existing file formats. If you want to store information on DNA polymorphisms, use the vcf format. While you will need to invest some time in understanding the file format, the payoff will be the ability to use existing, tested, efficient code to read and write in the data.

The second piece of advice is to ALWAYS USE PREXISTING MODULES for opening and storing data from these files. These modules will almost always be more accurate and more efficient than anything you might write. The challenge is to find and learn how to use the appropriate modules.

In this lecture we will go over a few modules in Python that can read in data from defined formats. The goal is to go over the process. Understand the file format, identify a module that can read in the data, and then understand how the data is stored in Python. Once we get to here, we can then worry how we can run whatever custom analysis we think is important for our data.

FASTA

Let's get into biological data files. Fasta file is the standard way to store DNA sequence. **FASTA format** is a text-based format for representing either nucleotide sequences or peptide sequences, in which nucleotides or amino acids are represented using single-letter codes. The format also allows for sequence names and comments to precede the sequences. The description line begins with a '>'.

```
>Protein_seq
MTEITAAMVKELRESTGAGMMDCKNALSETNGDFDKAVQLLREKGLGKAAKKADRLAAEG
LVSVKVSDDFTIAAMRPSYLSYEDLDMTFVENEYKALVAELEKENEERRRLKDPNKPEHK
IPQFASRKQLSDAILKEAEEKIKEELKAQGKPEKIWDNIIPGKMNSFIADNSQLDSKLTL
MGQFYVMDDKKTVEQVIAEKEKEFGGKIKIVEFICFEVGEGLEKKTEDFAAEVAAQL
>DNA_seq
ATGGTAGCGCTAGGGCTAGCTGAGCTAGGCTAGGAGGAGTAGAAAAAACGTCGCTAGCTA
atgccgagaccgaATGAGCCGATGAG
```

As an example Fasta file, we will cover the human genome. This file is found in example data and was originally downloaded from here:
https://www.ncbi.nlm.nih.gov/genome/guide/human/

This file contains the DNA sequences of 22 automosomes, the X chromosome, the Y chromosome, along with alternative haplotypes of hypervariable DNA and unplaced DNA sequences. A description of all of the data can be found here:

https://genome-asia.ucsc.edu/cgi-bin/hgTracks?db=hg38&chromInfoPage=

Additional FAQs about this fasta file can also be found here:

https://genome.ucsc.edu/FAQ/FAQdownloads.html

Now you might wonder why we should go to the trouble of using a custom module for reading in this data – a fasta file is pretty straightforward to parse. However, there are a few features that make it problematic.

1. The size of the data. In general, some genomes are so big that they are larger than your RAM size.
2. Compression. It is often beneficial to store the data in a compressed format that doesn't take up so much size on your disc.

For example:

```
(base) ➜  ExampleData ls -la
total 8483456
-rw-r--r--@  1 pythonclass  staff  3313061631 Sep 14 10:51 GRCh38_latest_genomic.fna
-rw-r--r--@  1 pythonclass  staff  1017005731 Sep 14 11:07 GRCh38_latest_genomic.fna.gz
```

Notice that the zipped version is 1/3 of the size as the normal version.

In order to read in the fasta file, we will use pysam. Instructions for how to use this can be found here:

https://pysam.readthedocs.io/en/latest/

To install, run the following commands:

```
conda config --add channels defaults
conda config --add channels conda-forge
conda config --add channels bioconda
conda install pysam
```

```
(base) ➜  ~ conda config --add channels defaults
conda config --add channels conda-forge
conda config --add channels bioconda
conda install pysam
Warning: 'defaults' already in 'channels' list, moving to the top
```

Now to use pysam to read in the fasta file:

```
>>> import pysam
>>> data = pysam.FastaFile('ExampleData/GRCh38_latest_genomic.fna.gz')
```

Note that this takes a while and two new files are created:

```
(base) ➜  ~ ls -la GRCh38_latest_genomic.fna.gz*
-rw-r--r--@ 1 pythonclass  staff  1017005731 Sep 14 14:33 GRCh38_latest_genomic.fna.gz
-rw-r--r--  1 pythonclass  staff       23583 Sep 14 14:35 GRCh38_latest_genomic.fna.gz.fai
-rw-r--r--  1 pythonclass  staff      812024 Sep 14 14:35 GRCh38_latest_genomic.fna.gz.gzi
```

These smaller files are known as index files. They help python search through the file to find the sequence that you are interested in WITHOUT needing to import the entire file. Note that this file is 1GB in size. That is a lot of RAM to take up. Data is a custom object that contains all of the sequences and names.

```
>>> data
<pysam.libcfaidx.FastaFile object at 0x7fc91816f400>
>>> dir(data)
['__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__enter__',
'__eq__', '__exit__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__len__',
'__lt__', '__ne__', '__new__', '__pyx_vtable__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__',
'__subclasshook__', '_open', 'close', 'closed', 'fetch', 'filename',
'get_reference_length', 'is_open', 'lengths', 'nreferences', 'references']
>>> data.references[0:10]
['NC_000001.11', 'NT_187361.1', 'NT_187362.1', 'NT_187363.1', 'NT_187364.1',
'NT_187365.1', 'NT_187366.1', 'NT_187367.1', 'NT_187368.1', 'NT_187369.1']
>>> data.lengths[0:10]
[248956422, 175055, 32032, 127682, 66860, 40176, 42210, 176043, 40745, 41717]
```

Notice the object has a built in __len__ magic method so you can get the number of contigs that are in the file with the len function.

```
>>> len(data)
639
```

If you want to access the individual sequence, the object acts very similarly to a dictionary of strings, where the key is the sequence name and the DNA sequence is the value:

```
>>> data['NT_187676.1'][5:10]
'aaaaa'
```

SPREADSHEETS

If you have ever worked with Microsoft Excel, you are probably familiar with spreadsheets. A spreadsheet file stores data in columns and rows. These can be stored in an excel specific format (.xls or .xlsx) or as CSV files. CSV is stored in plain text is a sequence of characters that is human-readable with a standard text editor. Each line is a data record. Each record consists of one or more fields, separated by a delimiter. Each record should have the same number of fields. In its strictest format, the delimiter should be a comma (i.e. each record is separated by a comma), however, CSV is flexible for what is used as the delimiter and often spaces, tabs, or semicolons are used. String fields can also be quoted (typically using a quote character) to overcome issues with the data field including the delimiter as part of its data. The first line may or may not be a header. An example of valid CSV format is found in the WorldDemographics.csv file that you have already seen.

In order to open csv/excel files, we will use pandas:

https://pandas.pydata.org/

```
(base) ➜  ~ conda install -c conda-forge pandas
```

In general, pandas is a very powerful, but complex module to use. We could spend an entire semester covering all of the things that pandas can do, and it is one of the main modules you would use if you became a Data Scientist. But even if you are not an expert, it can still be very powerful for basic data analysis.

Let’s see how we can open the WorldDemographics.csv file using pandas:

```
(base) ➜  ~ python3
Python 3.9.5 (default, May 18 2021, 12:31:01)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import pandas as pd
>>> dt = pd.read_csv('WorldDemographics.csv', sep = ',', index_col = 0)
>>> type(dt)
<class 'pandas.core.frame.DataFrame'>
```

A couple things to notice right away. 1st it is customary to import pandas as pd for shorthand. 2nd when you read it a table, it is customary to call it dt or df. You often need to type the name of the table multiple times in a line of code so it’s important that you keep the name of the table as short as possible.

Now, read_csv is a method that does most of the work, you need to specify the delimiter for each column (e.g. tab, etc) and the index column (more on that in later lectures.

```
>>> dt
      PopulationID  country_code    Level  continent_code  Age   #Alive
0      Afghanistan             4  Country            5501    0  1134501
1      Afghanistan             4  Country            5501    1  1134501
2      Afghanistan             4  Country            5501    2  1134501
3      Afghanistan             4  Country            5501    3  1124250
4      Afghanistan             4  Country            5501    4  1113998
...            ...           ...      ...             ...  ...      ...
20296     Zimbabwe           716  Country             910   96      163
20297     Zimbabwe           716  Country             910   97       53
20298     Zimbabwe           716  Country             910   98       41
20299     Zimbabwe           716  Country             910   99       30
20300     Zimbabwe           716  Country             910  100       19
```

You can also read in excel files pretty easily. There is an example excel file for you to play around with.

```
>>> dt = pd.read_excel('Cao_Supp.xlsx', sheet_name = 'Table S4', skiprows = [0])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/pythonclass/opt/miniconda3/lib/python3.9/site-
packages/pandas/util/_decorators.py", line 311, in wrapper
    return func(*args, **kwargs)
  File "/Users/pythonclass/opt/miniconda3/lib/python3.9/site-
packages/pandas/io/excel/_base.py", line 364, in read_excel
    io = ExcelFile(io, storage_options=storage_options, engine=engine)
  File "/Users/pythonclass/opt/miniconda3/lib/python3.9/site-
packages/pandas/io/excel/_base.py", line 1233, in __init__
    self._reader = self._engines[engine](self._io,
storage_options=storage_options)
  File "/Users/pythonclass/opt/miniconda3/lib/python3.9/site-
packages/pandas/io/excel/_openpyxl.py", line 521, in __init__
    import_optional_dependency("openpyxl")
  File "/Users/pythonclass/opt/miniconda3/lib/python3.9/site-
packages/pandas/compat/_optional.py", line 118, in import_optional_dependency
    raise ImportError(msg) from None
ImportError: Missing optional dependency 'openpyxl'.  Use pip or conda to install
openpyxl.
```

Note that pandas requires openpyxl as a dependency for reading in an excel file. Install this as well.

```
(base) ➜  ~ conda install -c conda-forge openpyxl
```

```
>>> import pandas as pd
>>> dt = pd.read_excel('Cao_Supp.xlsx', sheet_name = 'Table S4', skiprows = [0])
>>> dt
              gene_id    symbol   Germline  Intestinal/rectal_muscle  ...
Excretory_cells  Socket_cells     Rectum  Intestine
0      WBGene00000001     aap-1  62.497001                 10.205136  ...
11.550903     17.451509   0.000000  25.809255
1      WBGene00000002     aat-1   0.000000                  0.000000  ...
0.000000      0.000000  40.750949   1.025635
```

```
2      WBGene00000003     aat-2   0.000000                        80.016157  ...
0.000000      8.374096   4.062960   0.000000
3      WBGene00000004     aat-3   0.248650                        52.852916  ...
43.392868     76.614359  44.354308   5.056305
4      WBGene00000005     aat-4   0.118407                         2.012866  ...
0.000000      0.000000   0.000000  16.171277
...                ...       ...        ...                              ...  ...
...           ...       ...        ...
20266  WBGene00269389  C08F1.12   0.000000                         0.000000  ...
0.000000      0.000000   0.000000   0.000000
20267  WBGene00269391  F54F2.14   0.695717                         1.230763  ...
0.000000      0.000000   0.000000   8.098190
20268  WBGene00269392  F54F7.14   0.000000                         0.000000  ...
0.000000      0.000000   0.000000   0.902975
20269  WBGene00269394  F59H6.15   0.194782                         0.000000  ...
0.000000      0.000000   0.000000   0.000000
20270  WBGene00269421  R06F6.14   0.000000                         2.399670  ...
0.000000      0.000000   0.000000   0.000000
```

Success! Note we specified what worksheet to import and we also skipped the first row of metadata. There are a couple ways to now iterate through this table. One is to use the method iterrows() like so. You'll see the data is returned as a tuple.

Reading a table from a webpage

Sometimes data is stored on a webpage that you would like to analyze. Pandas can also read in this data with little work. First though, install the following modules:

```
(base) ➜  ~ conda install -c conda-forge lxml
(base) ➜  ~ conda install -c conda-forge html5lib
(base) ➜  ~ conda install -c conda-forge bs4
```

The simplest way is to read all the tables from a webpage. For example, previously we used data from the CDC to estimate the IFR for each age group. This information is updated periodically as more data is supplied. If you would like to grab this data, you can with the following command:

```
>>> tables = pd.read_html('https://www.cdc.gov/coronavirus/2019-
ncov/hcp/planning-scenarios.html')
```

This grabs all of the tables on a webpage and returns them as a list of data frames:

```
>>> tables
[                                          Parameter  ...                Scenario 5: Current Best Estimate
0                                              R0*  ...                                              2.5
1  Infection fatality ratio (Estimated number of ...  ...  0-17 years old: 20 18-49 years old: 500 50-64 ...
2       Percent of infections that are asymptomatic§  ...                                              30%
3  Infectiousness of asymptomatic individuals rel...  ...                                              75%
4  Percentage of transmission occurring prior to ...  ...                                              50%

[5 rows x 6 columns],                                                     0
1
0     Pre-existing immunity Assumption, ASPR and CDC  No pre-existing immunity before the pandemic b...
1                Time from exposure to symptom onset*                                      ~6 days (mean)
2   Time from symptom onset in an individual and s...                                      ~6 days (mean)
3   Mean ratio of estimated infections to reported...                                         11 (6, 24)
4        Parameter Values Related to Healthcare Usage       Parameter Values Related to Healthcare Usage
5   Median number of days from symptom onset to SA...                            Overall: 2 (0, 4) days
6   Median number of days from symptom onset to ho...  0-17 years old: 2 (0, 7) days 18-49 years old:...
7   Median number of days of hospitalization among...  0-17 years old: 2 (1, 4) days 18-49 years old:...
8   Median number of days of hospitalization among...  0-17 years old: 5 (2, 10.5) days 18-49 years o...
9   Percent admitted to the ICU among those hospit...  0-17 years old: 27.5% 18-49 years old: 18.9% 5...
10  Percent on mechanical ventilation among those ...  0-17 years old: 5.8% 18-49 years old: 9.0% 50-...
11  Percent that die among those hospitalized. Inc...  0-17 years old: 0.7% 18-49 years old: 2.1% 50-...
12  Median number of days of mechanical ventilatio...                              Overall: 5 (2, 11) days
13  Median number of days from symptom onset to de...  0–17 years old: 10 (4, 31) days 18–49 years ol...
14  Median number of days from death to reporting ...  0–17 years old: 8 (3, 33) days 18–49 years old...]
```

If you want to grab a specific table, you can by knowing the ID of the table you are looking for. This requires some basic knowledge of html.

```
tables = pd.read_html('https://www.cdc.gov/coronavirus/2019-ncov/hcp/planning-scenarios.html', attrs={'id': 'table01'})
```

Printing to a file

Printing to a file starts with opening it and designating it for writing:

```
f = open('outfile.txt','w')
```

Now you can use the print statement with the file keyword to print directly to the file. Note that a newline character is automatically added to the end of the line. If you want to suppress this, you can specify this with the end keyword:

```
print('Adding data to a file', file = f)
print('Adding data to a file', file = f)
print('Newline. ', file =f, end = '')

print('Now this prints on the same line', file = f)

print('This is on a new line', file = f)
```

```
$ python3 print_demonstration.py
$ more outfile.txt
Adding data to a file
Newline. Now this prints on the same line
This is on a new line
```

Don't forget to close the file.

You can also add equal spacing with the print statement

```
>>> for x in range(1,10):
...     print('{:2} {:3} {:4}'.format(x,x*x,x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
```

pandas modules

If you want to output a pandas datatable to an excel file or a csv file, it's easily accomplished like:

```
import pandas
```

```
writer = pandas.ExcelWriter('PythonExport.xlsx')
dt.to_excel(writer,'Sheet5')
writer.save()

# DF TO CSV
dt.to_csv('PythonExport.csv', sep=',')
```