Reading and writing data files

Learning goals

- read data files from disk
- practice importing python libraries
- practice exploring data
- understand calls to functions and variables assignments

In this tutorial, we will practice with reading data files from disk.

Reading files from disk is more generally known as file `i/o` where `i` and `o` stand for `i`nput and `o`utput, respectively. Why do we need to do file `i/o`?

File input

The input case is more obvious. To analyze data, you need data to analyze. Unless you know magic, you access data from *data files*, which are files just like a PDF documents, JPG images, or whatever, but they are specialized to some degree for containing data. Whether from a colleague, a boss, a webpage, or a government data repository, data will come in a data file that you will need to read as input in order visualize and analyze the data.

File output

The output case is perhaps less obvious. You read data into your Jupyter Notebook. The data are stored into Python variables. There are many different types of variables. We will learn about the fundamental ones. The variables can be used inside the Jupyter Notebook to make pretty graphs, and to do some cool analysis. But what if you want to share the numerical values, the results, of the analysis with someone else? In that case, you can write those values to a data file, save the file and send that to your colleagues. They can then read in the values on their end without having to wade through your notebook and cutting and pasting or whatever.

Let's get started!

Import libraries

As experienced in Tutorial 0001, python require importing the *Libraries* needed for the work.

Python *Libraries* are organized collections of Python code (files) written to perform related tasks. You will hear of "Libraries", "Packages", and "Modules". Technically, Libraries contain Packages which contain modules. Modules are single files that contain useful functions. Packages are directories containing files to a set of modules (or other packages), and Libraries are directories to a set of packages. That having been said, we will tend to refer to them interchagebly as "*useful chunks of pre-written code we can use!*"

Importing migth seem a bit tricky at the beginning as beginners do not know what libraries contains or do. So, all this at the beginning can appear a bit confusing, but later on as your experience will grow during the semester the role of each library in your work will become more clear, simpler.

This tutorial will cover the three fundamental libraries for data science in Python:

- pandas (https://pandas.pydata.org/) for reading, plotting and storing advanced data orbjects. Heareafter, we will using to read data saved in a file on your computer's hard drive. The operarions to read and write files from disk are generally referred to as `i/o` (i.e., `i` nput/ `o` utput file operations)
- matplotlib (https://matplotlib.org/) used to make simple plots
- seaborn (https://seaborn.pydata.org/) to plot more complex datasets

How to run code in Jupyter

*Remember, how we run code in Jupyter. There are two types of cells in this Jupyter Notebooks, text and code cellls. Text cells are in Markdown, code cells are python code.

Each code cell like the ones below, can be select and then hit we can hit **shift-return** (or shift-enter)*.

Import `pandas`

The first libray we import is called `pandas`. We import it using the shortname `pd`. Python programmers import libraries using nicknames. This helps making the code sorter when a library is used. For example, to use a command `read_csv()` available in `pandas` we would need to use the following line of code `pandas.read_csv()`, using the nickname `pd` the code shortens to `pd.read_csv()`. Nicknames are standardized in python, each libray is generally called with a specific nickname. Ok, let's import `pandas`:

In [1]:
```
import pandas as pd
```

Import `seaborn`

After importing `pandas` we will import another library called `seaborn`. `seaborn` is one of the most used libraries for data visualization. We will import it as `sns`, the commonly used shortcut:

In [2]:
```
import seaborn as sns
```

Import `matplotlib`'s `pyplot`

After `seaborn`, we import anohter major library, commonly used in data science applications: `pyplot`. Note that `pyplot` is part of the larger library called `matplotlib`. So, here we are importing a sub-module, a smaller library part of a larger library. The syntax goes as follows:

In [3]:
```
import matplotlib.pyplot as plt
```

Now you should have these three libraries available to you via their standard nicknames: `pd`, `plt`, and `sns`.

If 'sns' seems like a weird nickname for 'seaborn', I'll give you hint about it: It's an homage to one of the main characters in one of the best television series ever made. The show ran from 1999 until 2006 on NBC.

Data preparation

For this tutorial, we are going to read a data file called `006DataFile.csv`. The data was given to you and you are asked to save it inside a folder called `datasets`. The folder should be contained inside the same folder containing this Jupyter Notebook. Ideally, both `datasets` folder and Jupyter Notebook should be saved inside a GitHub repo.

Let's read some data!

There are many ways that data can be stored, from excel files to tables on webpages.

We read a file with exstension `.csv` (more on this file type in a bit) using the `pandas.read_csv()` function. But, remember, we have imported pandas as `pd`, so we read the `.csv` file, with slightly less typing, like this:

In [4]:

```
myDataFromFile = pd.read_csv("datasets/006DataFile.csv")
```

This command will work "out of the box" if your copy of the data file is in your "*datasets*" directory, which should be a subdirectory of the one this notebook is in.

Otherwise, you would have to change the command above to specify the path to the data file – where on the file tree the data file exists (either in '*absolute*' terms from root, or in '*relative*' terms from you current directory).

Answer the following questions:

In the line of code above, what is the:

- name of the library used to load the file? [pandas]
- name of the `pandas` function we use to read the data file? [pd.read_csv]
- data file name? [006DataFile.csv]
- name of the variable used to store the file? [myDataFromFile]
- name of the folder containing the data file? ["datasets"]

Let's look at what we just read.

Okay, now let's look at the file. We can take a quick peek by using the `display()` function:

In [5]:
```
display(myDataFromFile)
```

	VarA	VarB
0	0.979109	-0.128890
1	0.196564	0.403177
2	0.260841	0.682448
3	2.432641	-0.295968
4	-0.689790	-0.088941
...	...	...
95	2.416932	-1.065406
96	4.166266	-1.053911
97	-0.203719	0.610032
98	1.232813	-0.744738
99	0.833993	-0.372451

100 rows × 2 columns

Here, we can see that this file (like almost all data files) consists of rows and columns. The rows represent *observations* and the columns represent *variables*. This type of data file contains "tidy" data (if you have used R, you may have encountered the tidyverse). Sometimes, we will encounter data files that violate this "rows = observations, columns = variables" rule – untidy data – we will deal with this issue later in the class.

A very common genertic data file type is the comma separated values file, or .csv file. This is the type of data file we just loaded (006DataFile***.csv***). As the name implies, a file in this format consists values separated by commas to form rows, and "carriage returns" (CR) or "line feeds" (LF) marking the end of each row.

Answer the following question:

- What are the dimensions (the size) of the data? [100 rows x 2 columns]

Useless Trivia Alert!:

These terms come from typewritters and old-old-old-school printers, respectively. Typewritters had a "carriage" that held the paper and moved to the left while you typed. When you got to the right edge of the paper, you hit the "*carriage return*" key and the whole carriage flew back (*returned*) to right with a loud clunk and advanced the paper down a line. To this day, the big fat important key on the right side of most keyboards still says "return".

Old-school printers used long continuous "fan fold" sheets of paper (they could be literally hundreds of feet long) and had to be told to advance the paper one line with a "*line feed*" command. Once you were done printing, you ripped/cut your paper off the printer sort of like you do with aluminum foil or plastic wrap!

Useful aside!: `pandas` has very convenient functionality. For example, we can even copy data to the clipboard and read that in.

Go to Wikipedia and copy the table of the [population of Burkina Faso by year (https://en.wikipedia.org/wiki/Demographics_of_Burkina_Faso)](https://en.wikipedia.org/wiki/Demographics_of_Burkina_Faso).

After that, you can read the data from Wikipedia Table into a data table (technically a `pandas` data frame) like this:

In [8]:
```
cb = pd.read_clipboard()
```

In [9]:
```
cb
```

Out[9]:

	Total population	Population aged 0–14 (%)	Population aged 15–64 (%)	Population aged 65+ (%)
1950	4 284 000	40.7	57.3	2.0
1955	4 517 000	41.0	56.9	2.2
1960	4 829 000	41.3	56.3	2.3
1965	5 175 000	42.2	55.2	2.5
1970	5 625 000	43.3	53.9	2.8
1975	6 155 000	44.2	52.8	3.0
1980	6 823 000	45.6	51.2	3.2
1985	7 728 000	46.7	50.0	3.3
1990	8 811 000	47.3	49.5	3.3
1995	10 090 000	47.1	49.8	3.1
2000	11 608 000	46.8	50.5	2.8
2005	13 422 000	46.5	50.9	2.6
2010	15 605 000	46.2	51.3	2.5
2015	18 111 000	45.6	52.0	2.4
2020	20 903 000	44.4	53.2	2.4

How cool is that?!?!

Answer the following questions:

- What are the names of the columns of the data copied from Wikipedia? [Total population, Population aged 0–14 (%), Population aged 15–64 (%), and Population aged 65+ (%)]
- What is the name of the variable I used to save the data in Python? [cb]

Okay, now back to the show. In addition to `display()`, we use can use data frame "methods".

What is a "method"? Methods are things that an object, like our loaded datasets (technically a `pandas` data frame), can do. They are actions that an object can perform for you without any additional coding on your part!

Methods are invoked using the following syntax `ObjectName.MethodName`.

One thing a data frame knows how to do is show you its first few rows with the `head()` method. This method returns the top (leading, or head of a data table):

In [10]:
```
myDataFromFile.head()
```

Out[10]:

	VarA	VarB
0	0.979109	-0.128890
1	0.196564	0.403177
2	0.260841	0.682448
3	2.432641	-0.295968
4	-0.689790	-0.088941

Another method is `tail()`. This second method shows the last rows of the table:

In [11]:
```
myDataFromFile.tail()
```

Out[11]:

	VarA	VarB
95	2.416932	-1.065406
96	4.166266	-1.053911
97	-0.203719	0.610032
98	1.232813	-0.744738
99	0.833993	-0.372451

But how do you know what methods a given object has? Python's `dir()` function will give you a directory of any objects methods:

In [12]:
```
dir(myDataFromFile)
```

Out[12]:
```
['T',
 'VarA',
 'VarB',
 '_AXIS_LEN',
 '_AXIS_ORDERS',
 '_AXIS_TO_AXIS_NUMBER',
 '_HANDLED_TYPES',
 '__abs__',
 '__add__',
 '__and__',
 '__annotations__',
 '__array__',
 '__array_priority__',
 '__array_ufunc__',
 '__array_wrap__',
 '__bool__',
 '__class__',
 '__contains__',
 '__copy__',
```

HFS!!! Data frames know how to do a LOT! It's a bit overwhelming actually.

We can ignore all the things that look like __this__ at the top. Scrolling the the others, the method called `describe()` looks promising. Let's see what it does!

In [13]:
```
myDataFromFile.describe()
```

Out[13]:

	VarA	VarB
count	100.000000	100.000000
mean	1.195657	0.017781
std	1.620649	0.924191
min	-3.640916	-2.275922
25%	0.247035	-0.653144
50%	1.084324	0.049298
75%	2.338135	0.615286
max	5.092458	2.669149

Answer the following question:

- Describe in your own words what the method describe returns of the data. Describe the measures the method returns.

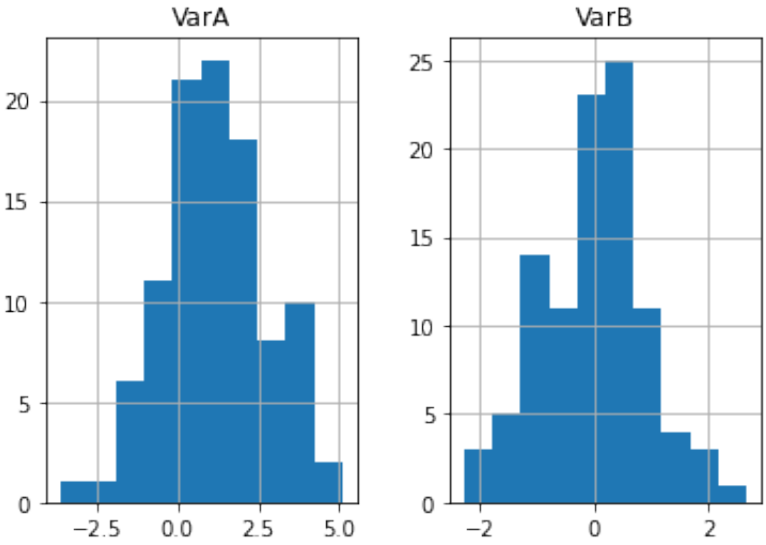
  [The "describe" mehtod gives us the summary of the data: total datapoints, mean, std, minimun valye, intervals, and maximum values.]

OMG, that was a good find!

I also noticed a `hist` method. Could it even be possible that data frames know how to draw histograms of themselves?

In [14]:
```
myDataFromFile.hist()
```

Out[14]:
```
array([[<AxesSubplot:title={'center':'VarA'}>,
        <AxesSubplot:title={'center':'VarB'}>]], dtype=object)
```



NO WAY!!!!!

As you can see, our journey of learning to play with data is going to be part learning to code and part figuring how to use what's already out there!

Answer the following questions:

- What are the titles of the two histograms in the figures created by the method `.hist` ? [VarA and VarB]
- How do the titles of the histograms relate to the data? [It is the names of the columns of the data]
- What are the values in the x-axis of the histograms? [The datapoints of each columns]

More advanced plotting

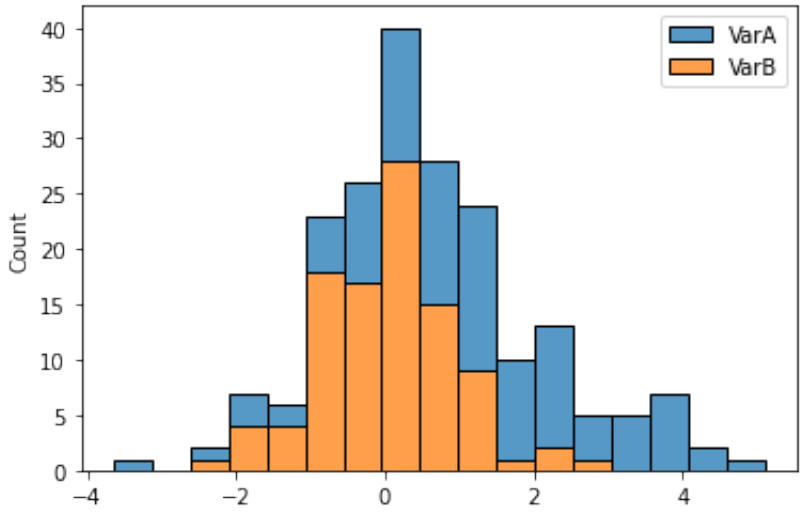
Okay, next we will practice making a plot using the fancy library called `seaborn`. Let's make a histogram with `seaborn`. `Seaborn` is a library that really just calls `matplotlib` functions, but it provides a way to use those functions that is easier than using them directly.

It also by default generally makes prettier plots.

Here's a histogram:

In [15]:
```
sns.histplot(myDataFromFile, multiple="stack")
```

Out[15]:
```
<AxesSubplot:ylabel='Count'>
```



The method `.hisplot` takes as inputs a data object (like `myDataFromFile`), but also it takes as input arguments. Arguments are also variables, that are passed to methods to provide additional requests. For example, above we asked for the two variables `A` and `B` to be stacked, to have the histograms plotted on top of eachother.

`seaborn.histplot()` has many additional arguments. You can Google `seaborn.histplot()` to see what other possible arguments are.

Alternatively, you can use python's `?` to open a new window that shows all the arguments that `.histplot()` has:

In [24]:
```
sns.histplot?
```

Answer the following questions:

- report three arguments of `.hist`
  [x=None,
  y=None,
  hue=None,]

The Kernel Density Estimate (KDE)

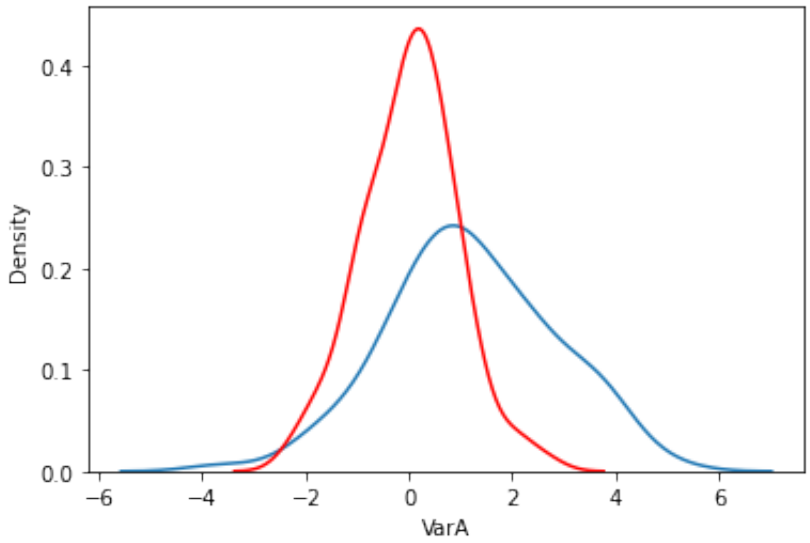
Now let's make Kernel Density Estimate (KDE) plots of the distributions. KDEs are essentially smoothed versions of histograms (we can unpack more about exactly what these are later). They can give us a better visual represtation of the "vibe" of a distribution.

`seaborn` has a method that automatically makes KDE plots: `.kdeplot()` .

For the KDE plot, we we will plot each variable separatedly. We will also use the `color` argument in the second plot to set the curves apart:

In [17]:
```
sns.kdeplot(myDataFromFile["VarA"])
sns.kdeplot(myDataFromFile["VarB"], color="r")
```

Out[17]:
```
<AxesSubplot:xlabel='VarA', ylabel='Density'>
```

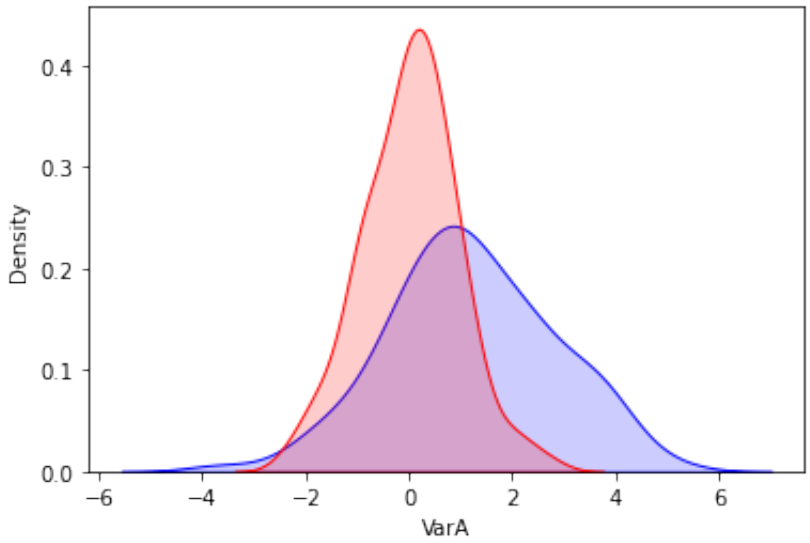


Here we can more clearly see the difference in means as compared to the histogram.

We can make the visualization more aethetically appealing by using some of the optional arguments to `seaborn.kdeplot()`

In [18]:
```
sns.kdeplot(myDataFromFile["VarA"], color="b", fill=True, alpha=0.2)
sns.kdeplot(myDataFromFile["VarB"], color="r", fill=True, alpha=0.2)
```

Out[18]:
```
<AxesSubplot:xlabel='VarA', ylabel='Density'>
```



The `color` argument is obvious. `fill` colors the area under the curve, and `alpha` make the fill transparent so you can see one curve through the other. Play around with these!

Answer the following questions:

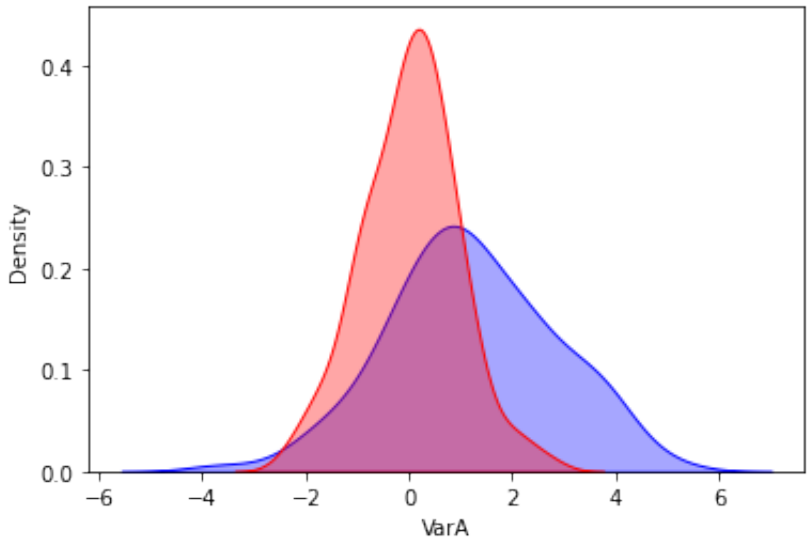
- Make the same plot as above but increaset the transparency of the histograms to 0.35. How does it look

  [sns.kdeplot(myDataFromFile["VarA"], color="b", fill=True, alpha=0.35)]

  [sns.kdeplot(myDataFromFile["VarB"], color="r", fill=True, alpha=0.35)]

In [19]:
```
sns.kdeplot(myDataFromFile["VarA"], color="b", fill=True, alpha=0.35)
sns.kdeplot(myDataFromFile["VarB"], color="r", fill=True, alpha=0.35)
```

Out[19]:
```
<AxesSubplot:xlabel='VarA', ylabel='Density'>
```



Let's see if we can write data to a file!

Now maybe we can write a summary of the original data to a file so we could potentially share it with other. What we'll do is use the `describe()` method again, but this time we'll assign it to new data frame.

In [20]:
```
mySummary = myDataFromFile.describe()
```

Let's just quickly that `mySummary` contains what we hope it does. The python command `print` will help us take a look at the summary:

In [21]:
```
print(mySummary)
```

```
              VarA        VarB
count  100.000000  100.000000
mean     1.195657    0.017781
std      1.620649    0.924191
min     -3.640916   -2.275922
25%      0.247035   -0.653144
50%      1.084324    0.049298
75%      2.338135    0.615286
max      5.092458    2.669149
```

See what we did above? Instead of returning the results of the method `.describe()` directly in the notebook output, we saved the output into a variable. We then used `print` to display the content of the variable.

Answer the following questions:

- What is the name of the variable we saved the output of the method `.describe()` in? [mySummary]

Next let's write the variable to a file! Given that we are dealing with a table, we will save the variable in a `.csv` file. The variable has a method `.to_csv`, the method can be found by lurking the output of `dir(<varName>)`.

In [22]:
```
mySummary.to_csv("mySummary.csv")
```

Okay, but how do we know that worked? Easy! We'll read that file back in using `pandas.read_csv()` and see what it looks like!

In [23]:
```
mySummary2 = pd.read_csv("mySummary.csv")
```

And then we can look at it using `display()`.

In [25]:
```
display(mySummary2)
```

	Unnamed: 0	VarA	VarB
0	count	100.000000	100.000000
1	mean	1.195657	0.017781
2	std	1.620649	0.924191
3	min	-3.640916	-2.275922
4	25%	0.247035	-0.653144
5	50%	1.084324	0.049298
6	75%	2.338135	0.615286
7	max	5.092458	2.669149

Sweet! We can now read and write data files. File I/0 handled!

Final Report for this Tutorial.

- create a new repository under your user account on github.com/yourUserName (github.com/yourUserName).
- clone the repository locally on your computer (pick an ideal folder to do so, say, `~/git` or `~/code`)
- move the file of jupyter notebookd for this tutorial in the folder of the repository cloned above. This will require using a combination of commands such as `mv` and `cp` and `cd` and `ls` (you can also do this by use your mouse, but make sure you are copying inside the proper).
- use `git add [file name]` Add the file you edited for this tutorial to the repository
- use `git commit –am "Your message here"` to commit the file to the repository
- push the local repository with the changes and the newly added file to the cloud using `git push [repository name]`
- Submit the URL to the github repository on Canvas

  Note. Now on, the above is going to be required for every tutorial. We will always ask for you to report the URL of the tutorial with all the operations performed by you and the answers to the questions visible.

In [ ]: