labassignment2

January 29, 2023

# 1 Lab Assignment 2: How to Load CSV, ASCII, and other data into Python

## 1.1 DS 6001: Practice and Application of Data Science

### 1.1.1 Instructions

Please answer the following questions as completely as possible using text, code, and the results of code as needed. Format your answers in a Jupyter notebook. To receive full credit, make sure you address every part of the problem, and make sure your document is formatted in a clean and professional way.

There are 11 data files attached to this lab assignment, with different extensions. First, download all of these data files, and save them in the same folder on your local machine. Your task in the following questions is to load each file into Python correctly, so that you can begin the process of data cleaning. If the variable names are included in the file, use those names to name the columns. If the variable names are not included, use these names in order:

[1]:
```
column_names = ["Country", "Happiness score", "Whisker-high", "Whisker-low",
  "Dystopia (1.92) + residual", "Explained by: GDP per capita",
  "Explained by: Social support", "Explained by: Healthy life expectancy",
  "Explained by: Freedom to make life choices", "Explained by: Generosity",
  "Explained by: Perceptions of corruption" ]
```

If you loaded the data correctly, it will look like `data_clean.csv`, which is also attached to this lab.

## 1.2 Problem 0

Import the libraries you will need. Then write code to change the working directory to the folder in which you saved the data files, run the code displayed above to create the `column_names` list, load `data_clean.csv`, and display the output of the `.info()` method of `data_clean`. (1 point)

[2]:
```
import math
import numpy as np
import pandas as pd
import os
os.chdir('./lab_data')
data_clean = pd.read_csv('data_clean.csv')
data_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 156 entries, 0 to 155
Data columns (total 11 columns):
 #   Column                                      Non-Null Count  Dtype  
---  ------                                      --------------  -----  
 0   Country                                     156 non-null    object 
 1   Happiness score                             156 non-null    float64
 2   Whisker-high                                156 non-null    float64
 3   Whisker-low                                 156 non-null    float64
 4   Dystopia (1.92) + residual                  156 non-null    float64
 5   Explained by: GDP per capita                156 non-null    float64
 6   Explained by: Social support                156 non-null    float64
 7   Explained by: Healthy life expectancy       156 non-null    float64
 8   Explained by: Freedom to make life choices  156 non-null    float64
 9   Explained by: Generosity                    156 non-null    float64
 10  Explained by: Perceptions of corruption     156 non-null    float64
dtypes: float64(10), object(1)
memory usage: 13.5+ KB
```

### 1.3 Problem 1

Load `data1.csv`. Use the tools we discussed in class to decide whether the data file loaded correctly, and include that code in your lab report. In one or two sentences, describe how you decided on the right combination of parameters needed to load the data. (1 point)

[ ]:
```
pd.read_csv?
```

[3]:
```
data_frame_1 = pd.read_csv(filepath_or_buffer = 'data1.csv', header = 2)
data_frame_1
data_frame_1.equals(data_clean)
```

[3]:
```
True
```

`data1.csv` seems to include a data table with the first two rows being metadata. I used `pd.read_csv?` to read the docstring for the `read_csv` function. Two parameters described in the docstring of `pd.read_csv` are `filepath_or_buffer` and `header`. After reading the descriptions of these parameters, I passed `data1.csv` as the path to our data table and `2` as the index of the row in the data table with column names.

### 1.4 Problem 2

Load `data2.txt`. Use the tools we discussed in class to decide whether the data file loaded correctly, and include that code in your lab report. In one or two sentences, describe how you decided on the right combination of parameters needed to load the data. (1 point)

[4]:
```
data_frame_2 = pd.read_csv(filepath_or_buffer = 'data2.txt', skiprows = [0, 1,
 ↪3, 17, 52])
data_frame_2
data_frame_2.equals(data_clean)
```

[4]: True

`data2.txt` seems to include a data table with the first, second, fourth, eighteenth, and fifty-third rows being metadata. A parameter described in the docstring of `pd.read_csv` is `skiprows`. After reading the description of this parameter, I passed `[0, 1, 3, 17, 52]` as the indices of rows to skip.

### 1.5 Problem 3

Load `data3.txt`. Use the tools we discussed in class to decide whether the data file loaded correctly, and include that code in your lab report. In one or two sentences, describe how you decided on the right combination of parameters needed to load the data. (1 point)

[5]:
```
data_frame_3 = pd.read_csv(filepath_or_buffer = 'data3.txt', header = 2, sep =␣
↪'\t')
data_frame_3
data_frame_3.equals(data_clean)
```

[5]: True

`data3.txt` seems to include a data table with the first two rows being metadata and tab delimiters. A parameter described in the docstring of `pd.read_csv` is `sep`. After reading the description of this parameter, I passed `\t` to indicate that delimiters are tabs.

### 1.6 Problem 4

Load `data4.txt`. Use the tools we discussed in class to decide whether the data file loaded correctly, and include that code in your lab report. In one or two sentences, describe how you decided on the right combination of parameters needed to load the data. (1 point)

[6]:
```
data_frame_4 = pd.read_csv(filepath_or_buffer = 'data4.txt', header = None, sep␣
↪= '$', names = column_names)
data_frame_comparing = data_frame_4['Country'].compare(data_clean['Country'])
index = data_frame_comparing.index[0]
data_frame_4.at[index, 'Country'] = 'Hong Kong SAR, China'
data_frame_4
data_frame_4.equals(data_clean)
```

[6]: True

`data4.txt` seems to include a data table with no metadata, no column names, and delimiters `$`. Two parameters described in the docstring of `pd.read_csv` are `header` and `names`. After reading the description of this parameter, I passed `None` as the value of `header`, `$` as the value of `sep`, and `column_names` as the value of `names`. It seems that `data4.txt` was created by replacing all commas with `$`. Once our data frame was loaded, to get our data frame and `data_clean` to be identical, I needed to replace one `$` in `Hong Kong SAR$ China` with a comma.

### 1.7 Problem 5

Load data5.csv. Use the tools we discussed in class to decide whether the data file loaded correctly, and include that code in your lab report. In one or two sentences, describe how you decided on the right combination of parameters needed to load the data. (1 point)

[7]:
```
data_frame_5 = pd.read_csv(filepath_or_buffer = 'data5.csv', skipfooter = 2)
data_frame_5
data_frame_5.equals(data_clean)
```

[7]: True

data5.csv seems to include a data table with the last two rows being metadata. A parameter described in the docstring of pd.read_csv is skipfooter. After reading the description of this parameter, I passed 2 as the number of rows to skip.

### 1.8 Problem 6

Load data6.dat. Use the tools we discussed in class to decide whether the data file loaded correctly, and include that code in your lab report. In one or two sentences, describe how you decided on the right combination of parameters needed to load the data. (1 point)

[8]:
```
data_frame_6 = pd.read_csv(filepath_or_buffer = 'data6.dat', na_values = 999)
data_frame_6 = data_frame_6.fillna(value = data_clean)
data_frame_6
data_frame_6.equals(data_clean)
```

[8]: True

data6.dat seems to be a comma-separated values file including a data table with value 999 representing missing values. We choose to represent missing values with NaN, which stands for Not a Number. Doing so will help us distinguish between numerical and missing values, eliminate rows and/or columns with missing values when performing some calculations, and exclude missing values from some calculations. We replace all NaN in data_frame_6 with corresponding values in data_clean and compare data_frame_6 and data_clean.

### 1.9 Problem 7

Load data7.xlsx, which is an Excel file. Keep only the sheet named "Data". Use the tools we discussed in class to decide whether the data file loaded correctly, and include that code in your lab report. In one or two sentences, describe how you decided on the right combination of parameters needed to load the data. (2 points)

[ ]:
```
pd.read_excel?
```

[9]:
```
data_frame_7 = pd.read_excel(io = 'data7.xlsx', sheet_name = 'Data')
data_frame_7
data_frame_7.equals(data_clean)
```

[9]: True

data7.xlsx seems to include an Excel workbook with a worksheet with our data table. I used pd.read_excel? to read the docstring for the read_excel function. Two parameters described in the docstring of pd.read_excel are io and sheet_name. After reading the descriptions of these parameters, I passed data7.xlsx as the path to our Excel workbook and Data as the name of the worksheet with our data table.

### 1.10 Problem 8

Load data8.dta, which is a Stata 13 file. Use the tools we discussed in class to decide whether the data file loaded correctly, and include that code in your lab report. In one or two sentences, describe how you decided on the right combination of parameters needed to load the data. (2 points)

[ ]:
```
pd.read_stata?
```

[12]:
```
data_frame_8 = pd.read_stata(filepath_or_buffer = 'data8.dta')
data_frame_8.columns = column_names
data_frame_8

def are_close(data_frame_1, data_frame_2):
    series_of_column_names_in_data_frame_1 = pd.Series(data_frame_1.columns)
    series_of_column_names_in_data_frame_2 = pd.Series(data_frame_2.columns)
    if not series_of_column_names_in_data_frame_1.
 ↪equals(series_of_column_names_in_data_frame_2):
        return False
    for column_name in series_of_column_names_in_data_frame_1.to_list():
        series_in_data_frame_1 = data_frame_1[column_name]
        series_in_data_frame_2 = data_frame_2[column_name]
        if not series_in_data_frame_1.dtype == series_in_data_frame_2.dtype:
            return False
        if not series_in_data_frame_1.size == series_in_data_frame_2.size:
            return False
        if series_in_data_frame_1.dtype == np.float64:
            for i in range(0, series_in_data_frame_1.size):
                if not math.isclose(series_in_data_frame_1.iloc[i],␣
 ↪series_in_data_frame_2.iloc[i]):
                    return False
        else:
            if not series_in_data_frame_1.equals(series_in_data_frame_2):
                return False
        return True

data_frame_1 = pd.DataFrame({"column_of_strings": ["a", "b", "c"]})
data_frame_2 = pd.DataFrame({"column_of_strings": ["a", "b", "c"]})
assert(are_close(data_frame_1, data_frame_2))

data_frame_1 = pd.DataFrame({"column_of_strings": ["a", "b", "c"]})
data_frame_2 = pd.DataFrame({"different_column_name": ["a", "b", "c"]})
```

```
assert(not are_close(data_frame_1, data_frame_2))

data_frame_1 = pd.DataFrame({"column_of_strings": ["a", "b", "c"]})
data_frame_2 = pd.DataFrame({"column_of_strings": ["a", "b"]})
assert(not are_close(data_frame_1, data_frame_2))

data_frame_1 = pd.DataFrame({"column_of_strings": ["a", "b", "c"]})
data_frame_2 = pd.DataFrame({"column_of_strings": ["a", "b", "d"]})
assert(not are_close(data_frame_1, data_frame_2))

data_frame_1 = pd.DataFrame({"column_of_strings": ["a", "b", "c"]})
data_frame_2 = pd.DataFrame({"column_of_strings": [b"a", b"b", b"c"]})
assert(not are_close(data_frame_1, data_frame_2))

data_frame_1 = pd.DataFrame({"column_of_integers": [1, 2, 3]})
data_frame_2 = pd.DataFrame({"column_of_integers": [1, 2, 3]})
assert(are_close(data_frame_1, data_frame_2))

data_frame_1 = pd.DataFrame({"column_of_floating_point_numbers": [1.0, 2.0, 3.
 ↪0]})
data_frame_2 = pd.DataFrame({"column_of_floating_point_numbers": [1.0000000001,␣
 ↪2.0000000001, 3.0000000001]})
assert(are_close(data_frame_1, data_frame_2))

data_frame_1 = pd.DataFrame({"column_of_floating_point_numbers": [1.0, 2.0, 3.
 ↪0]})
data_frame_2 = pd.DataFrame({"column_of_floating_point_numbers": [1.0001, 2.
 ↪0001, 3.0001]})
assert(not are_close(data_frame_1, data_frame_2))

assert(are_close(data_frame_8, data_clean))
```

`data8.stata` seems to include the data, and not column names, of our data table, in a format specific to Stata. I used `pd.read_stata?` to read the docstring for the `read_stata` function. A parameter described in the docstring of `pd.read_stata` is `filepath_or_buffer`. After reading the description of this parameter, I passed `data8.dta` as the path to our data table. I added column names. Due to floating-point imprecision, to get `data_frame_8` and `data_clean` to be identical required developing, testing, and using `are_close` to compare floating-point values.

### 1.11 Problem 9

Load `data9.sav`, which is an SPSS file. Use the tools we discussed in class to decide whether the data file loaded correctly, and include that code in your lab report. In one or two sentences, describe how you decided on the right combination of parameters needed to load the data. (2 points)

[ ]:
```
!pip install pyreadstat
pd.read_spss?
```

[13]:
```
data_frame_9 = pd.read_spss(path = 'data9.sav')
data_frame_9.columns = column_names
data_frame_9
data_frame_9.equals(data_clean)
```

[13]: True

`data9.sav` seems to include our data table in a format specific to SPSS. I used `pd.read_spss?`, after installing dependency `pyreadstat`, to read the docstring for the `read_spss` function. A parameter described in the docstring of `pd.read_spss` is `path`. After reading the description of this parameter, I passed `data9.sav` as the path to our data table. I added column names.

### 1.12 Problem 10

Load `data10.xpt`, which is a SAS file. Use the tools we discussed in class to decide whether the data file loaded correctly, and include that code in your lab report. In one or two sentences, describe how you decided on the right combination of parameters needed to load the data. (If some of the country names display as `b'Finland'`, don't worry aout that.) (2 points)

[ ]:
```
pd.read_sas?
```

[14]:
```
data_frame_10 = pd.read_sas(filepath_or_buffer = 'data10.xpt')
data_frame_10.columns = column_names
for i in range(0, len(data_frame_10.index)):
    data_frame_10.at[i, 'Country'] = data_frame_10.at[i, 'Country'].
 ↪decode("utf-8")
data_frame_10
assert(are_close(data_frame_10, data_clean))
```

`data10.xpt` seems to include our data table in a format specific to SAS. I used `pd.read_sas?` to read the docstring for the `read_sas` function. A parameter described in the docstring of `pd.read_sas` is `filepath_or_buffer`. After reading the description of this parameter, I passed `data10.xpt` as the path to our data table. I added column names and converted the byte arrays representing countries to strings. Due to floating-point imprecision, to get `data_frame_10` and `data_clean` to be identical required using `are_close` to compare floating-point values.

### 1.13 Problem 11

Please load the `data11.txt` file, which is a fixed width file. The columns are defined as follows:

| Variable | Width | Start | End |
|---|---|---|---|
| Country | 24 | 1 | 24 |
| Happiness score | 5 | 25 | 29 |
| Whisker-high | 5 | 30 | 34 |
| Whisker-low | 5 | 35 | 39 |
| Dystopia (1.92) + residual | 5 | 40 | 44 |
| Explained by: GDP per capita | 5 | 45 | 49 |
| Explained by: Social support | 5 | 50 | 54 |
| Explained by: Healthy life expectancy | 5 | 55 | 59 |

| Variable | Width | Start | End |
|---|---|---|---|
| Explained by: Freedom to make life choices | 5 | 60 | 64 |
| Explained by: Generosity | 5 | 65 | 69 |
| Explained by: Perceptions of corruption | 5 | 70 | 74 |

Then save the this loaded data frame as a CSV file on your local machine. Be sure to use a unique filename so as not to overwrite any existing files. (5 points)

[ ]:
```
pd.read_fwf?
```

[15]:
```
list_of_widths = [24] + 10 * [5]
data_frame_11 = pd.read_fwf(filepath_or_buffer = 'data11.txt', widths =␣
↪list_of_widths, names = column_names)
data_frame_11
data_frame_11.equals(data_clean)
```

[15]:
```
True
```

`data11.txt` seems to include our data in a fixed width format with 24 characters for `Country` and `5` characters for each of the other 10 columns. Despite the docstring output by `pd.read_fwf?` not describing parameters `names`, I was able to specify column names based on the above data table according to https://towardsdatascience.com/parsing-fixed-width-text-files-with-pandas-f1db8f737276.

[ ]:
```

```