Lecture 03.4 Pandas

August 31, 2023

1 Intro to Pandas

Duncan Callaway

My objective in this notebook is to review enough of the basics to give people a clear sense of * what a pandas **data frame** is * what it can contain * how to access the information in it

1.1 Pandas references

Introductory:

- Getting started with Python for research, a gentle introduction to Python in data-intensive research.
- A Whirlwind Tour of Python, by Jake VanderPlas, another quick Python intro (with notebooks).

Core Pandas/Data Science books:

- The Python Data Science Handbook, by Jake VanderPlas.
- Python for Data Analysis, 2nd Edition, by Wes McKinney, creator of Pandas. Companion Notebooks
- Effective Pandas, a book by Tom Augspurger, core Pandas developer.

Complementary resources:

- An introduction to "Data Science", a collection of Notebooks by BIDS' Stéfan Van der Walt.
- Effective Computation in Physics, by Kathryn D. Huff; Anthony Scopatz. Notebooks to accompany the book. Don't be fooled by the title, it's a great book on modern computational practices with very little that's physics-specific.

1.2 Introduction

In Data8 you used the `tables` library to organize and manipulate data. I've not used tables but it has been described to me as a 'light' version of pandas. So this will be somewhat familiar to you.

Pandas has several features that make it immediately better than numpy for organizing data. 1. You can have different data types (string, int, float) in each column 2. You can label columns (headers) 3. You can label rows (index)

We call the data structure that holds all these things together a **data frame**

Ambiguity alert: up to now we've talked about indexing to access individual elements of a data structure. The index in pandas has a slightly more specific meaning, in that it references the rows of the data frame. Pandas documentation talks about "location" or "position" in place using "index" as we did with the other data structures. I'll try my best to disambiguate, but also rely on context to clarify the meaning of the terms.

At its core, the data frame structure is what draws us to use pandas. But it also has a bunch of great built-in functions you can use to manipulate data once it is loaded in.

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

I'm going to define a simple data frame in a way that you can see the connections to existing Python data types and structures.

First let's define a dict of fruit information:

```
[2]: fruit_info_dict = {'fruit':['apple','banana','orange','raspberry'],
                        'color':['red','yellow','orange','pink'],
                        'weight':[120,150,250,15]}
     fruit_info_dict
```

```
[2]: {'fruit': ['apple', 'banana', 'orange', 'raspberry'],
      'color': ['red', 'yellow', 'orange', 'pink'],
      'weight': [120, 150, 250, 15]}
```

1.2.1 Q: What do you get if you call the dict of lists with a key?

```
[3]: fruit_info_dict['color']
```

```
[3]: ['red', 'yellow', 'orange', 'pink']
```

Ans: A the list associated with the key

1.2.2 Q: Figure out how to get the weight of a raspberry out of the dict.

```
[4]: fruit_info_dict['weight'][3]
```

```
[4]: 15
```

1.3 Now let's make a dataframe

It's just a fancy version of a dict of lists:

```
[5]: fruit_info_df = pd.DataFrame(
         data={'fruit':['apple','banana','orange','raspberry'],
                        'color':['red','yellow','orange','pink'],
                        'weight':[120,150,250,15]
              })
     fruit_info_df
```

[5]:
```
       fruit   color  weight
0      apple     red     120
1     banana  yellow     150
2     orange  orange     250
3  raspberry    pink      15
```

Some notes: 1. You can see that we put the data inside curly brackets much like we do in a dict. 1. We defined it as if fruit, color and weight are keys. 1. In fact, we'll be calling these column names. 2. The pandas data structure is called the data frame.

[6]:
```
type(fruit_info_df)
```

[6]:
```
pandas.core.frame.DataFrame
```

We can call the values associated with each column name (like the dict key) in much the same way that we did with the dict:

[7]:
```
fruit_info_dict['color']
```

[7]:
```
['red', 'yellow', 'orange', 'pink']
```

[8]:
```
fruit_info_df['color']
```

[8]:
```
0       red
1    yellow
2    orange
3      pink
Name: color, dtype: object
```

But notice above in the dataframe we don't just have a list of colors. Instead we have something called a **series**. This is a pandas object that is analogous to a numpy series.

[9]:
```
type(fruit_info_df['color'])
```

[9]:
```
pandas.core.series.Series
```

The series differs from the list in at least one important way: It has numbers directly associated with it that we call the index. (The left column of the Series.)

Note that in the above, the index is just numeric. But as we'll see, we can make it whatever we want. This makes the data frame much more flexible than a list – we can call elements from a sensical index, rather than just a number.

We can also call columns from the data frame as follows:

[10]:
```
fruit_info_df.color
```

[10]:
```
0       red
1    yellow
2    orange
3      pink
```

```
Name: color, dtype: object
```

But as we'll see soon, there are alternative ways to get access to the elements of the data frame (`.loc` and `.iloc`) that enable us to work with the frame more as we would a numpy array.

First, let me show you how to get into the dataframe to index individual elements if we *don't* use `.loc` or `.iloc`.

In this case you index into the df in a way that looks a bit like indexing into a *list*. That is, you use two sequences of square brackets, the first carrying information about the column and the second information about the row.

```
[11]: fruit_info_df['fruit'][1]
```

```
[11]: 'banana'
```

```
[12]: fruit_info_df.fruit[1]
```

```
[12]: 'banana'
```

Slicing is limited in this case: we can only slice down rows, not columns of the df:

```
[13]: fruit_info_df['fruit'][0:2]
```

```
[13]: 0     apple
      1    banana
      Name: fruit, dtype: object
```

1.3.1 Anatomy of the data frame.

Let's talk a little about the anatomy of the data frame.

We have the following important attributes: 1. Rows 2. Columns 2. Index 3. Column names

The "index" can be numeric, but as we'll see we can also make the indices strings.

1.3.2 Pandas and the CAISO data.

Let's see what's in the current directory. In macos, I type:

```
[14]: !ls
```

```
CAISO_2017to2018.csv                 Lecture 03.3 Numpy.pdf
Lecture 03 InClassNotebook.ipynb     Lecture 03.4 Pandas.ipynb
Lecture 03.1 DataTypes Pandas.pptx   Lecture 03.4 Pandas.pdf
Lecture 03.2 DataTypes.ipynb         dataframe_anatomy.png
Lecture 03.2 DataTypes.pdf           from drive
Lecture 03.3 Numpy.ipynb             ~$Lecture 03.1 DataTypes Pandas.pptx
```

But if someone picked up my ipynb and ran it on windows, they'd get an error, since windows doesn't recognize `ls` but rather `dir`.

Instead we can use the `os` library:

[15]:
```
import os
os.listdir()
```

[15]:
```
['Lecture 03.3 Numpy.pdf',
 '.DS_Store',
 '~$Lecture 03.1 DataTypes Pandas.pptx',
 'dataframe_anatomy.png',
 'Lecture 03.2 DataTypes.pdf',
 'Lecture 03.4 Pandas.ipynb',
 'Lecture 03 InClassNotebook.ipynb',
 'Lecture 03.1 DataTypes Pandas.pptx',
 'Lecture 03.3 Numpy.ipynb',
 '.ipynb_checkpoints',
 'from drive',
 'Lecture 03.4 Pandas.pdf',
 'Lecture 03.2 DataTypes.ipynb',
 'CAISO_2017to2018.csv']
```

I've already downloaded a file, "CAISO_2017to2018.csv", that has one year of renewables production data from CAISO.

Let's load that in as a dataframe and take a look at it.

The simplest command is `pd.read_csv`

[16]:
```
caiso_data = pd.read_csv('CAISO_2017to2018.csv')
```

Now we can look at the top of the dataframe using the .head method.

Note that you need to put parentheses on the end of the call, otherwise python returns the head "object" in a rather ugly form.

[17]:
```
caiso_data.head()
```

[17]:
```
            Unnamed: 0  GEOTHERMAL  BIOMASS  BIOGAS  SMALL HYDRO  WIND TOTAL  \
0  2017-08-29 00:00:00        1181      340     156          324        1551
1  2017-08-29 01:00:00        1182      338     156          326        1556
2  2017-08-29 02:00:00        1183      337     156          337        1325
3  2017-08-29 03:00:00        1185      339     156          313        1158
4  2017-08-29 04:00:00        1190      344     156          320        1209

   SOLAR PV  SOLAR THERMAL
0         0              0
1         0              0
2         0              0
3         0              0
4         0              0
```

Pandas loaded the date and time in as a column and put its own row numbers on the data frame.

As an alternative we can actually make the row labels *equal* to whatever column of data we'd like.

We'll come back to the notion of the index a little later, but for now let's just reload with the date-time as the index:

```
[18]: caiso_data = pd.read_csv('CAISO_2017to2018.csv', index_col=0)
      caiso_data.head()
```

```
[18]:                       GEOTHERMAL  BIOMASS  BIOGAS  SMALL HYDRO  WIND TOTAL  \
      2017-08-29 00:00:00        1181      340     156          324        1551
      2017-08-29 01:00:00        1182      338     156          326        1556
      2017-08-29 02:00:00        1183      337     156          337        1325
      2017-08-29 03:00:00        1185      339     156          313        1158
      2017-08-29 04:00:00        1190      344     156          320        1209

                           SOLAR PV  SOLAR THERMAL
      2017-08-29 00:00:00         0              0
      2017-08-29 01:00:00         0              0
      2017-08-29 02:00:00         0              0
      2017-08-29 03:00:00         0              0
      2017-08-29 04:00:00         0              0
```

We can also rename column names to make things prettier

```
[19]: caiso_data = pd.read_csv('CAISO_2017to2018.csv')
      caiso_data.columns
```

```
[19]: Index(['Unnamed: 0', 'GEOTHERMAL', 'BIOMASS', 'BIOGAS', 'SMALL HYDRO',
             'WIND TOTAL', 'SOLAR PV', 'SOLAR THERMAL'],
            dtype='object')
```

Note that we can't reassign easily because column and index names lists are immutable. Here is the workaround:

```
[20]: cols = caiso_data.columns.tolist()
      cols[0] = 'Date and time'
      caiso_data.columns = cols
      caiso_data
```

```
[20]:             Date and time  GEOTHERMAL  BIOMASS  BIOGAS  SMALL HYDRO  \
      0     2017-08-29 00:00:00        1181      340     156          324
      1     2017-08-29 01:00:00        1182      338     156          326
      2     2017-08-29 02:00:00        1183      337     156          337
      3     2017-08-29 03:00:00        1185      339     156          313
      4     2017-08-29 04:00:00        1190      344     156          320
      …                       …         …        …       …            …
      8755  2018-08-28 19:00:00         962      332     236          581
      8756  2018-08-28 20:00:00         967      336     234          547
      8757  2018-08-28 21:00:00         972      336     233          502
      8758  2018-08-28 22:00:00         975      333     234          361
      8759  2018-08-28 23:00:00         977      333     235          262
      WIND TOTAL  SOLAR PV  SOLAR THERMAL
0           1551         0              0
1           1556         0              0
2           1325         0              0
3           1158         0              0
4           1209         0              0
…              …         …              …
8755        3300        70             24
8756        3468         0             17
8757        3310         0             17
8758        3068         0              0
8759        2921         0              0

[8760 rows x 8 columns]
```

Ok, that looks a little better for now.

As you can see, all the data are the same type of numeric value – MWh.

In these cases, sometimes it's natural to "stack" the data.

We could do the stacking with a pandas command, `.stack`

Let's work with a few things to explore the data frame.

We've already learned about `.head` but we can use it slightly differently:

```
[21]: caiso_data.head(2) # the number in the parens tells pandas how many rows
```

```
[21]:         Date and time  GEOTHERMAL  BIOMASS  BIOGAS  SMALL HYDRO  WIND TOTAL  \
      0 2017-08-29 00:00:00        1181      340     156          324        1551
      1 2017-08-29 01:00:00        1182      338     156          326        1556

         SOLAR PV  SOLAR THERMAL
      0         0              0
      1         0              0
```

We can also look at the tail!

```
[22]: caiso_data.tail()
```

```
[22]:            Date and time  GEOTHERMAL  BIOMASS  BIOGAS  SMALL HYDRO  \
      8755 2018-08-28 19:00:00         962      332     236          581
      8756 2018-08-28 20:00:00         967      336     234          547
      8757 2018-08-28 21:00:00         972      336     233          502
      8758 2018-08-28 22:00:00         975      333     234          361
      8759 2018-08-28 23:00:00         977      333     235          262

            WIND TOTAL  SOLAR PV  SOLAR THERMAL
      8755        3300        70             24
```

```
8756         3468           0              17
8757         3310           0              17
8758         3068           0               0
8759         2921           0               0
```

What's the shape of the frame?

[23]:
```
caiso_data.shape
```

[23]:
```
(8760, 8)
```

The `.shape` method returns a tuple – number of rows and number of columns.

What about the total number of entries? Use the `.size` method:

[24]:
```
caiso_data.size
```

[24]:
```
70080
```

We can also summarize all the numeric data:

[25]:
```
caiso_data.describe()
```

[25]:
```
        GEOTHERMAL      BIOMASS       BIOGAS  SMALL HYDRO   WIND TOTAL  \
count  8760.000000  8760.000000  8760.000000  8760.000000  8760.000000
mean    949.228881   329.203311   224.342808   394.902626  1806.004338
std     108.304664    43.243815    24.228372   112.120046  1284.668963
min     468.000000   164.000000   133.000000   148.000000     0.000000
25%     923.000000   301.000000   219.000000   308.000000   598.000000
50%     956.000000   331.000000   234.000000   379.000000  1615.000000
75%     986.000000   363.000000   240.000000   476.250000  2900.250000
max    1230.000000   482.000000   253.000000   681.000000  5006.000000

          SOLAR PV  SOLAR THERMAL
count  8760.000000    8760.000000
mean   2988.774658     133.840753
std    3628.189420     200.374857
min       0.000000       0.000000
25%       0.000000       0.000000
50%     175.500000       0.000000
75%    6700.250000     261.000000
max   10102.000000     679.000000
```

1.3.3 Q: In class. Extract the first 5 entries of BIOGAS and SOLAR PV

[26]:
```
caiso_data[['BIOGAS', 'SOLAR PV']][0:4]
```

[26]:
```
   BIOGAS  SOLAR PV
0     156         0
1     156         0
```

```
2     156          0
3     156          0
```

Note that you need to put the column names in two sets of square brackets * the first set tells pandas you're passing a key / column name into the data frame, and * the second set of square brackets allows you to pass a *set* of column names.

1.4 Recap

- Pandas dataframes are sophisticated dicts of lists.
  - They have attributes like columns and index that have special meaning in the pandas context.
  - You can store any combination of data types in the dataframe
- You can access information in them as though they are dicts of lists