Lecture 03.3 Pandas

September 1, 2022

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.

[ ]:
```
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:

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

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

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

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.

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

1.3 Now let's make a dataframe

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

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

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.

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

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:

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

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

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.

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

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:

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

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.

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

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

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

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

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:

```
[ ]: !ls
```

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:

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

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`

```
[ ]: 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.

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

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:

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

We can also rename column names to make things prettier

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

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

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

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:

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

We can also look at the tail!

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

What's the shape of the frame?

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

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

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

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

We can also summarize all the numeric data:

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

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

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

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.

1.3.4 Indexing and slicing in Pandas

To motivate our interest going forward, let's ask a basic question about the data set, for example:

What hour had the lowest average wind generation in the last year?

Try thinking for a moment about how you'd do this, then we'll try to get there.

First let's figure out how to slice these data frames.

`.iloc` allows us to index and slice on **i**nteger row and column positions:

```
[ ]: caiso_data.iloc[1,1]
```

But what's nice about `.iloc` is that you can also slice. It works just like numpy.

1.3.5 Q: Take a slice of the `caiso_data` dataframe that grabs the first four columns of data and the first 10 rows

```
[ ]: caiso_data.iloc[0:10, :4]
```

1.3.6 Q: What would you do if you wanted to get the *last* 10 rows?

```
[ ]: caiso_data.iloc[-10:, :4]
```

1.3.7 Q: Can you print out the last ten rows in reverse order?

```
[ ]: caiso_data.iloc[:-10:-1,:4]
```

`.loc` is similar to `.iloc`, but it allows you to call the index and column names:

```
[ ]: caiso_data.loc[0:5,'Date and time']
```

You can even slice with column names:

```
[ ]: caiso_data.loc[0:5,'Date and time':'BIOGAS']
```

1.3.8 Q: Is .loc end-inclusive or exclusive when you slice?

Ans: *inclusive*. This is because it requires less knowledge about other rows in the DataFrame.

Note that this is true for both the index and the 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
- But you can also use the .loc and .iloc methods to access information in a way similar to numpy, including clean slicing.