datasframe

ARCHIVE

Modern Pandas (Part 1)

Posted on: Mon 21 March 2016
Tags: #pandas

This is part 1 in my series on writing modern idiomatic pandas.

Effective Pandas

Introduction

This series is about how to make effective use of pandas, a data analysis library for the Python programming language. It's targeted at an intermediate level: people who have some experience with pandas, but are looking to improve.

Prior Art

There are many great resources for learning pandas; this is not one of them. For beginners, I typically recommend Greg Reda's 3-part introduction, especially if they're familiar with SQL. Of course, there's the pandas documentation itself. I gave a talk at PyData Seattle targeted as an introduction if you prefer video form. Wes McKinney's Python for Data Analysis is still the goto book (and is also a really good introduction to NumPy as well). Jake VanderPlas's Python Data Science Handbook, in early release, is great too. Kevin Markham has a video series for beginners learning pandas.

With all those resources (and many more that I've slighted through omission), why write another? Surely the law of diminishing returns is kicking in by now. Still, I thought there was room for a guide that is up to date (as of March 2016) and emphasizes idiomatic pandas code (code that is *pandorable*). This series probably won't be appropriate for people completely new to python or NumPy and pandas. By luck, this first post happened to cover topics that are relatively introductory, so read some of the linked material and come back, or let me know if you have questions.

Get the Data

We'll be working with flight delay data from the BTS (R users can install Hadley's NYCFlights13 dataset for similar data.

```
import os
import zipfile

import requests
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

if int(os.environ.get("MODERN_PANDAS_EPUB", 0)):
    import prep
```

```
import requests

headers = {
    'Referer': 'https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=
    'Origin': 'https://www.transtats.bts.gov',
    'Content-Type': 'application/x-www-form-urlencoded',
}

params = (
    ('Table_ID', '236'),
    ('Has_Group', '3'),
    ('Is_Zipped', '0'),
)

with open('modern-1-url.txt', encoding='utf-8') as f:
    data = f.read().strip()

os.makedirs('data', exist_ok=True)
dest = "data/flights.csv.zip"

if not os.path.exists(dest):
    r = requests.post('https://www.transtats.bts.gov/DownLoad_Table.asp',
                      headers=headers, params=params, data=data, stream=True)

    with open("data/flights.csv.zip", 'wb') as f:
        for chunk in r.iter_content(chunk_size=102400):
            if chunk:
                f.write(chunk)
```

That download returned a ZIP file. There's an open Pull Request for automatically decompressing ZIP archives with a single CSV, but for now we have to extract it ourselves and then read it in.

```
zf = zipfile.ZipFile("data/flights.csv.zip")
fp = zf.extract(zf.filelist[0].filename, path='data/')
```

```
df = pd.read_csv(fp, parse_dates=["FL_DATE"]).rename(columns=str.lower)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 450017 entries, 0 to 450016
Data columns (total 33 columns):
fl_date                  450017 non-null datetime64[ns]
unique_carrier           450017 non-null object
airline_id               450017 non-null int64
tail_num                 449378 non-null object
fl_num                   450017 non-null int64
origin_airport_id        450017 non-null int64
origin_airport_seq_id    450017 non-null int64
origin_city_market_id    450017 non-null int64
origin                   450017 non-null object
origin_city_name         450017 non-null object
dest_airport_id          450017 non-null int64
dest_airport_seq_id      450017 non-null int64
dest_city_market_id      450017 non-null int64
dest                     450017 non-null object
dest_city_name           450017 non-null object
crs_dep_time             450017 non-null int64
dep_time                 441476 non-null float64
dep_delay                441476 non-null float64
taxi_out                 441244 non-null float64
wheels_off               441244 non-null float64
wheels_on                440746 non-null float64
taxi_in                  440746 non-null float64
crs_arr_time             450017 non-null int64
arr_time                 440746 non-null float64
arr_delay                439645 non-null float64
cancelled                450017 non-null float64
cancellation_code        8886 non-null object
carrier_delay            97699 non-null float64
weather_delay            97699 non-null float64
nas_delay                97699 non-null float64
security_delay           97699 non-null float64
late_aircraft_delay      97699 non-null float64
unnamed: 32              0 non-null float64
dtypes: datetime64[ns](1), float64(15), int64(10), object(7)
memory usage: 113.3+ MB
```

Indexing

Or, *explicit is better than implicit*. By my count, 7 of the top-15 voted pandas questions on Stackoverflow are about indexing. This seems as good a place as any to start.

By indexing, we mean the selection of subsets of a DataFrame or Series. `DataFrames` (and to a lesser extent, `Series`) provide a difficult set of challenges:

- Like lists, you can index by location.
- Like dictionaries, you can index by label.
- Like NumPy arrays, you can index by boolean masks.

- Any of these indexers could be scalar indexes, or they could be arrays, or they could be `slice`s.
- Any of these should work on the index (row labels) or columns of a DataFrame.
- And any of these should work on hierarchical indexes.

The complexity of pandas' indexing is a microcosm for the complexity of the pandas API in general. There's a reason for the complexity (well, most of it), but that's not *much* consolation while you're learning. Still, all of these ways of indexing really are useful enough to justify their inclusion in the library.

Slicing

Or, *explicit is better than implicit*.

By my count, 7 of the top-15 voted pandas questions on Stackoverflow are about slicing. This seems as good a place as any to start.

Brief history digression: For years the preferred method for row and/or column selection was `.ix`.

```
df.ix[10:15, ['fl_date', 'tail_num']]
```

```
/Users/taugspurger/Envs/blog/lib/python3.6/site-packages/ipykernel_launcher.py:1: Deprecation
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate_ix
  """Entry point for launching an IPython kernel.
```

	fl_date	tail_num
10	2017-01-01	N756AA
11	2017-01-01	N807AA
12	2017-01-01	N755AA
13	2017-01-01	N951AA
14	2017-01-01	N523AA
15	2017-01-01	N155AA

As you can see, this method is now deprecated. Why's that? This simple little operation hides some complexity. What if, rather than our default `range(n)` index, we had an integer index like

```
# filter the warning for now on
import warnings
warnings.simplefilter("ignore", DeprecationWarning)
```

```
first = df.groupby('airline_id')[['fl_date', 'unique_carrier']].first()
first.head()
```

	fl_date	unique_carrier
airline_id		

	fl_date	unique_carrier
airline_id		
19393	2017-01-01	WN
19690	2017-01-01	HA
19790	2017-01-01	DL
19805	2017-01-01	AA
19930	2017-01-01	AS

Can you predict ahead of time what our slice from above will give when passed to `.ix`?

```
first.ix[10:15, ['fl_date', 'tail_num']]
```

	fl_date	tail_num
airline_id		

Surprise, an empty DataFrame! Which in data analysis is rarely a good thing. What happened?

We had an integer index, so the call to `.ix` used its label-based mode. It was looking for integer *labels* between 10:15 (inclusive). It didn't find any. Since we sliced a range it returned an empty DataFrame, rather than raising a KeyError.

By way of contrast, suppose we had a string index, rather than integers.

```
first = df.groupby('unique_carrier').first()
first.ix[10:15, ['fl_date', 'tail_num']]
```

	fl_date	tail_num
unique_carrier		
VX	2017-01-01	N846VA
WN	2017-01-01	N955WN

And it works again! Now that we had a string index, `.ix` used its positional-mode. It looked for *rows* 10-15 (exclusive on the right).

But you can't reliably predict what the outcome of the slice will be ahead of time. It's on the *reader* of the code (probably your future self) to know the dtypes so you can reckon whether `.ix` will use label indexing (returning the empty DataFrame) or positional indexing (like the last example). In general, methods whose behavior depends on the data, like `.ix` dispatching to label-based indexing on integer Indexes but location-based indexing on non-integer, are hard to use correctly. We've been trying to stamp them out in pandas.

Since pandas 0.12, these tasks have been cleanly separated into two methods:

1. `.loc` for label-based indexing
2. `.iloc` for positional indexing

```
first.loc[['AA', 'AS', 'DL'], ['fl_date', 'tail_num']]
```

	fl_date	tail_num
unique_carrier		

	fl_date	tail_num
unique_carrier		
AA	2017-01-01	N153AA
AS	2017-01-01	N557AS
DL	2017-01-01	N942DL

```
first.iloc[[0, 1, 3], [0, 1]]
```

	fl_date	airline_id
unique_carrier		
AA	2017-01-01	19805
AS	2017-01-01	19930
DL	2017-01-01	19790

`.ix` is deprecated, but will hang around for a little while. But if you've been using `.ix` out of habit, or if you didn't know any better, maybe give `.loc` and `.iloc` a shot. I'd recommend carefully updating your code to decide if you've been using positional or label indexing, and choose the appropriate indexer. For the intrepid reader, Joris Van den Bossche (a core pandas dev) compiled a great overview of the pandas `__getitem__` API. A later post in this series will go into more detail on using Indexes effectively; they are useful objects in their own right, but for now we'll move on to a closely related topic.

SettingWithCopy

Pandas used to get *a lot* of questions about assignments seemingly not working. We'll take this StackOverflow question as a representative question.

```
f = pd.DataFrame({'a':[1,2,3,4,5], 'b':[10,20,30,40,50]})
f
```

	a	b
0	1	10
1	2	20
2	3	30
3	4	40
4	5	50

The user wanted to take the rows of `b` where `a` was 3 or less, and set them equal to `b / 10` We'll use boolean indexing to select those rows `f['a'] <= 3`,

```
# ignore the context manager for now
with pd.option_context('mode.chained_assignment', None):
    f[f['a'] <= 3]['b'] = f[f['a'] <= 3 ]['b'] / 10
f
```

	a	b
0	1	10

	a	b
1	2	20
2	3	30
3	4	40
4	5	50

And nothing happened. Well, something did happen, but nobody witnessed it. If an object without any references is modified, does it make a sound?

The warning I silenced above with the context manager links to an explanation that's quite helpful. I'll summarize the high points here.

The "failure" to update `f` comes down to what's called *chained indexing*, a practice to be avoided. The "chained" comes from indexing multiple times, one after another, rather than one single indexing operation. Above we had two operations on the left-hand side, one `__getitem__` and one `__setitem__` (in python, the square brackets are syntactic sugar for `__getitem__` or `__setitem__` if it's for assignment). So `f[f['a'] <= 3]['b']` becomes

1. `getitem`: `f[f['a'] <= 3]`
2. `setitem`: `_['b'] = ...` # using `_` to represent the result of 1.

In general, pandas can't guarantee whether that first `__getitem__` returns a view or a copy of the underlying data. The changes *will* be made to the thing I called `_` above, the result of the `__getitem__` in `1`. But we don't know that `_` shares the same memory as our original `f`. And so we can't be sure that whatever changes are being made to `_` will be reflected in `f`.

Done properly, you would write

```
f.loc[f['a'] <= 3, 'b'] = f.loc[f['a'] <= 3, 'b'] / 10
f
```

	a	b
0	1	1.0
1	2	2.0
2	3	3.0
3	4	40.0
4	5	50.0

Now this is all in a single call to `__setitem__` and pandas can ensure that the assignment happens properly.

The rough rule is any time you see back-to-back square brackets, `][`, you're in asking for trouble. Replace that with a `.loc[..., ...]` and you'll be set.

The other bit of advice is that a SettingWithCopy warning is raised when the *assignment* is made. The potential copy could be made earlier in your code.

Multidimensional Indexing

MultiIndexes might just be my favorite feature of pandas. They let you represent higher-dimensional datasets in a familiar two-dimensional table, which my brain can sometimes handle. Each additional level of the MultiIndex represents another dimension. The cost of this is somewhat harder label indexing.

My very first bug report to pandas, back in November 2012, was about indexing into a MultiIndex. I bring it up now because I genuinely couldn't tell whether the result I got was a bug or not. Also, from that bug report

> *Sorry if this isn't actually a bug. Still very new to python. Thanks!*

Adorable.

That operation was made much easier by this addition in 2014, which lets you slice arbitrary levels of a MultiIndex.. Let's make a MultiIndexed DataFrame to work with.

```
hdf = df.set_index(['unique_carrier', 'origin', 'dest', 'tail_num',
                    'fl_date']).sort_index()
hdf[hdf.columns[:4]].head()
```

					airline_id	**fl_num**	**origin_airport_id**	**origin_airport_seq_id**
unique_carrier	**origin**	**dest**	**tail_num**	**fl_date**				
AA	**ABQ**	**DFW**	**N3ABAA**	**2017-01-15**	19805	2611	10140	1014003
				2017-01-29	19805	1282	10140	1014003
			N3AEAA	**2017-01-11**	19805	2511	10140	1014003
			N3AJAA	**2017-01-24**	19805	2511	10140	1014003
			N3AVAA	**2017-01-11**	19805	1282	10140	1014003

And just to clear up some terminology, the *levels* of a MultiIndex are the former column names (`unique_carrier`, `origin`...). The labels are the actual values in a level, (`'AA'`, `'ABQ'`, ...). Levels can be referred to by name or position, with 0 being the outermost level.

Slicing the outermost index level is pretty easy, we just use our regular `.loc[row_indexer, column_indexer]`. We'll select the columns `dep_time` and `dep_delay` where the carrier was American Airlines, Delta, or US Airways.

```
hdf.loc[['AA', 'DL', 'US'], ['dep_time', 'dep_delay']]
```

					dep_time	**dep_delay**
unique_carrier	**origin**	**dest**	**tail_num**	**fl_date**		
AA	**ABQ**	**DFW**	**N3ABAA**	**2017-01-15**	500.0	0.0
				2017-01-29	757.0	-3.0
			N3AEAA	**2017-01-11**	1451.0	-9.0

					dep_time	dep_delay
unique_carrier	**origin**	**dest**	**tail_num**	**fl_date**		
			N3AJAA	**2017-01-24**	1502.0	2.0
			N3AVAA	**2017-01-11**	752.0	-8.0
			N3AWAA	**2017-01-27**	1550.0	50.0
			N3AXAA	**2017-01-16**	1524.0	24.0
				2017-01-17	757.0	-3.0
			N3BJAA	**2017-01-25**	823.0	23.0
			N3BPAA	**2017-01-11**	1638.0	-7.0
			N3BTAA	**2017-01-26**	753.0	-7.0
			N3BYAA	**2017-01-18**	1452.0	-8.0
			N3CAAA	**2017-01-23**	453.0	-7.0
			N3CBAA	**2017-01-13**	1456.0	-4.0
			N3CDAA	**2017-01-12**	1455.0	-5.0
				2017-01-28	758.0	-2.0
			N3CEAA	**2017-01-21**	455.0	-5.0
			N3CGAA	**2017-01-18**	759.0	-1.0
			N3CWAA	**2017-01-27**	1638.0	-7.0
			N3CXAA	**2017-01-31**	752.0	-8.0
			N3DBAA	**2017-01-19**	1637.0	-8.0
			N3DMAA	**2017-01-13**	1638.0	-7.0
			N3DRAA	**2017-01-27**	753.0	-7.0
			N3DVAA	**2017-01-09**	1636.0	-9.0
			N3DYAA	**2017-01-10**	1633.0	-12.0
			N3ECAA	**2017-01-15**	753.0	-7.0
			N3EDAA	**2017-01-09**	1450.0	-10.0
				2017-01-10	753.0	-7.0
			N3ENAA	**2017-01-24**	756.0	-4.0
				2017-01-26	1533.0	33.0
...	**...**	**...**	**...**	**...**	...	...
DL	**XNA**	**ATL**	**N921AT**	**2017-01-20**	1156.0	-3.0
			N924DL	**2017-01-30**	555.0	-5.0
			N925DL	**2017-01-12**	551.0	-9.0
			N929AT	**2017-01-08**	1155.0	-4.0
				2017-01-31	1139.0	-20.0
			N932AT	**2017-01-12**	1158.0	-1.0
			N938AT	**2017-01-26**	1204.0	5.0
			N940AT	**2017-01-18**	1157.0	-2.0
				2017-01-19	1200.0	1.0
			N943DL	**2017-01-22**	555.0	-5.0

					dep_time	dep_delay
unique_carrier	origin	dest	tail_num	fl_date		
			N950DL	2017-01-19	558.0	-2.0
			N952DL	2017-01-18	556.0	-4.0
			N953DL	2017-01-31	558.0	-2.0
			N956DL	2017-01-17	554.0	-6.0
			N961AT	2017-01-14	1233.0	-6.0
			N964AT	2017-01-27	1155.0	-4.0
			N966DL	2017-01-23	559.0	-1.0
			N968DL	2017-01-29	555.0	-5.0
			N969DL	2017-01-11	556.0	-4.0
			N976DL	2017-01-09	622.0	22.0
			N977AT	2017-01-24	1202.0	3.0
				2017-01-25	1149.0	-10.0
			N977DL	2017-01-21	603.0	-2.0
			N979AT	2017-01-15	1238.0	-1.0
				2017-01-22	1155.0	-4.0
			N983AT	2017-01-11	1148.0	-11.0
			N988DL	2017-01-26	556.0	-4.0
			N989DL	2017-01-25	555.0	-5.0
			N990DL	2017-01-15	604.0	-1.0
			N995AT	2017-01-16	1152.0	-7.0

142945 rows × 2 columns

So far, so good. What if you wanted to select the rows whose origin was Chicago O'Hare (`ORD`) or Des Moines International Airport (DSM). Well, `.loc` wants `[row_indexer, column_indexer]` so let's wrap the two elements of our row indexer (the list of carriers and the list of origins) in a tuple to make it a single unit:

```
hdf.loc[(['AA', 'DL', 'US'], ['ORD', 'DSM']), ['dep_time', 'dep_delay']]
```

					dep_time	dep_delay
unique_carrier	origin	dest	tail_num	fl_date		
AA	DSM	DFW	N424AA	2017-01-23	1324.0	-3.0
			N426AA	2017-01-25	541.0	-9.0
			N437AA	2017-01-13	542.0	-8.0
				2017-01-23	544.0	-6.0
			N438AA	2017-01-11	542.0	-8.0
			N439AA	2017-01-24	544.0	-6.0
				2017-01-31	544.0	-6.0
			N4UBAA	2017-01-18	1323.0	-4.0
			N4WNAA	2017-01-27	1322.0	-5.0

					dep_time	dep_delay
unique_carrier	**origin**	**dest**	**tail_num**	**fl_date**		
			N4XBAA	**2017-01-09**	536.0	-14.0
			N4XEAA	**2017-01-21**	544.0	-6.0
			N4XFAA	**2017-01-31**	1320.0	-7.0
			N4XGAA	**2017-01-28**	1337.0	10.0
				2017-01-30	542.0	-8.0
			N4XJAA	**2017-01-20**	552.0	2.0
				2017-01-21	1320.0	-7.0
			N4XKAA	**2017-01-26**	1323.0	-4.0
			N4XMAA	**2017-01-16**	1423.0	56.0
				2017-01-19	1321.0	-6.0
			N4XPAA	**2017-01-09**	1322.0	-5.0
				2017-01-14	545.0	-5.0
			N4XTAA	**2017-01-10**	1355.0	28.0
			N4XUAA	**2017-01-13**	1330.0	3.0
				2017-01-14	1319.0	-8.0
			N4XVAA	**2017-01-28**	NaN	NaN
			N4XXAA	**2017-01-15**	1322.0	-5.0
				2017-01-16	545.0	-5.0
			N4XYAA	**2017-01-18**	559.0	9.0
			N4YCAA	**2017-01-26**	545.0	-5.0
				2017-01-27	544.0	-6.0
...	**...**	**...**	**...**	**...**	...	...
DL	**ORD**	**SLC**	**N316NB**	**2017-01-23**	1332.0	-6.0
			N317NB	**2017-01-09**	1330.0	-8.0
				2017-01-11	1345.0	7.0
			N319NB	**2017-01-17**	1353.0	15.0
				2017-01-22	1331.0	-7.0
			N320NB	**2017-01-13**	1332.0	-6.0
			N321NB	**2017-01-19**	1419.0	41.0
			N323NB	**2017-01-01**	1732.0	57.0
				2017-01-02	1351.0	11.0
			N324NB	**2017-01-16**	1337.0	-1.0
			N326NB	**2017-01-24**	1332.0	-6.0
				2017-01-26	1349.0	11.0
			N329NB	**2017-01-06**	1422.0	32.0
			N330NB	**2017-01-04**	1344.0	-6.0
				2017-01-12	1343.0	5.0
			N335NB	**2017-01-31**	1336.0	-2.0

					dep_time	dep_delay
unique_carrier	origin	dest	tail_num	fl_date		
			N338NB	2017-01-29	1355.0	17.0
			N347NB	2017-01-08	1338.0	0.0
			N348NB	2017-01-10	1355.0	17.0
			N349NB	2017-01-30	1333.0	-5.0
			N352NW	2017-01-06	1857.0	10.0
			N354NW	2017-01-04	1844.0	-3.0
			N356NW	2017-01-02	1640.0	20.0
			N358NW	2017-01-05	1856.0	9.0
			N360NB	2017-01-25	1354.0	16.0
			N365NB	2017-01-18	1350.0	12.0
			N368NB	2017-01-27	1351.0	13.0
			N370NB	2017-01-20	1355.0	17.0
			N374NW	2017-01-03	1846.0	-1.0
			N987AT	2017-01-08	1914.0	29.0

5582 rows × 2 columns

Now try to do any flight from ORD or DSM, not just from those carriers. This used to be a pain. You might have to turn to the `.xs` method, or pass in `df.index.get_level_values(0)` and zip that up with the indexers your wanted, or maybe reset the index and do a boolean mask, and set the index again... ugh.

But now, you can use an `IndexSlice`.

```
hdf.loc[pd.IndexSlice[:, ['ORD', 'DSM']], ['dep_time', 'dep_delay']]
```

					dep_time	dep_delay
unique_carrier	origin	dest	tail_num	fl_date		
AA	DSM	DFW	N424AA	2017-01-23	1324.0	-3.0
			N426AA	2017-01-25	541.0	-9.0
			N437AA	2017-01-13	542.0	-8.0
				2017-01-23	544.0	-6.0
			N438AA	2017-01-11	542.0	-8.0
			N439AA	2017-01-24	544.0	-6.0
				2017-01-31	544.0	-6.0
			N4UBAA	2017-01-18	1323.0	-4.0
			N4WNAA	2017-01-27	1322.0	-5.0
			N4XBAA	2017-01-09	536.0	-14.0
			N4XEAA	2017-01-21	544.0	-6.0
			N4XFAA	2017-01-31	1320.0	-7.0
			N4XGAA	2017-01-28	1337.0	10.0
				2017-01-30	542.0	-8.0

					dep_time	dep_delay
unique_carrier	**origin**	**dest**	**tail_num**	**fl_date**		
			N4XJAA	**2017-01-20**	552.0	2.0
				2017-01-21	1320.0	-7.0
			N4XKAA	**2017-01-26**	1323.0	-4.0
			N4XMAA	**2017-01-16**	1423.0	56.0
				2017-01-19	1321.0	-6.0
			N4XPAA	**2017-01-09**	1322.0	-5.0
				2017-01-14	545.0	-5.0
			N4XTAA	**2017-01-10**	1355.0	28.0
			N4XUAA	**2017-01-13**	1330.0	3.0
				2017-01-14	1319.0	-8.0
			N4XVAA	**2017-01-28**	NaN	NaN
			N4XXAA	**2017-01-15**	1322.0	-5.0
				2017-01-16	545.0	-5.0
			N4XYAA	**2017-01-18**	559.0	9.0
			N4YCAA	**2017-01-26**	545.0	-5.0
				2017-01-27	544.0	-6.0
...	**...**	**...**	**...**	**...**	...	...
WN	**DSM**	**STL**	**N635SW**	**2017-01-15**	1806.0	6.0
			N645SW	**2017-01-22**	1800.0	0.0
			N651SW	**2017-01-01**	1856.0	61.0
			N654SW	**2017-01-21**	1156.0	126.0
			N720WN	**2017-01-23**	605.0	-5.0
				2017-01-31	603.0	-7.0
			N724SW	**2017-01-30**	1738.0	-7.0
			N734SA	**2017-01-20**	1839.0	54.0
			N737JW	**2017-01-09**	605.0	-5.0
			N747SA	**2017-01-27**	610.0	0.0
			N7718B	**2017-01-18**	1736.0	-9.0
			N772SW	**2017-01-31**	1738.0	-7.0
			N7735A	**2017-01-11**	603.0	-7.0
			N773SA	**2017-01-17**	1743.0	-2.0
			N7749B	**2017-01-10**	1746.0	1.0
			N781WN	**2017-01-02**	1909.0	59.0
				2017-01-30	605.0	-5.0
			N7827A	**2017-01-14**	1644.0	414.0
			N7833A	**2017-01-06**	659.0	49.0
			N7882B	**2017-01-15**	901.0	1.0
			N791SW	**2017-01-26**	1744.0	-1.0

					dep_time	dep_delay
unique_carrier	**origin**	**dest**	**tail_num**	**fl_date**		
			N903WN	**2017-01-13**	1908.0	83.0
			N905WN	**2017-01-05**	605.0	-5.0
			N944WN	**2017-01-02**	630.0	5.0
			N949WN	**2017-01-01**	624.0	4.0
			N952WN	**2017-01-29**	854.0	-6.0
			N954WN	**2017-01-11**	1736.0	-9.0
			N956WN	**2017-01-06**	1736.0	-9.0
			NaN	**2017-01-16**	NaN	NaN
				2017-01-17	NaN	NaN

19466 rows × 2 columns

The `:` says include every label in this level. The `IndexSlice` object is just sugar for the actual python `slice` object needed to remove slice each level.

```
pd.IndexSlice[:, ['ORD', 'DSM']]
```

```
(slice(None, None, None), ['ORD', 'DSM'])
```

We'll talk more about working with Indexes (including MultiIndexes) in a later post. I have an unproven thesis that they're underused because `IndexSlice` is underused, causing people to think they're more unwieldy than they actually are. But let's close out part one.

WrapUp

This first post covered Indexing, a topic that's central to pandas. The power provided by the DataFrame comes with some unavoidable complexities. Best practices (using `.loc` and `.iloc`) will spare you many a headache. We then toured a couple of commonly misunderstood sub-topics, setting with copy and Hierarchical Indexing.

Tom Augspurger