Effective Pandas

Tom Augspurger

Chapter 1

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 experince 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 theyre'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.

```python
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

headers = {
    'Pragma': 'no-cache',
    'Origin': 'http://www.transtats.bts.gov',
    'Accept-Encoding': 'gzip, deflate',
    'Accept-Language': 'en-US,en;q=0.8',
    'Upgrade-Insecure-Requests': '1',
    'User-Agent': ('Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) '
                   'AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.'
                   '0.2564.116 Safari/537.36'),
    'Content-Type': 'application/x-www-form-urlencoded',
    'Accept': ('text/html,application/xhtml+xml,application/xml;q=0.9,'
               'image/webp,*/*;q=0.8'),
    'Cache-Control': 'no-cache',
    'Referer': ('http://www.transtats.bts.gov/DL_SelectFields.asp?Table'
                '_ID=236&DB_Short_Name=On-Time'),
    'Connection': 'keep-alive',
    'DNT': '1',
}

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('http://www.transtats.bts.gov/DownLoad_Table.asp?Table_ID=236'
                      '&Has_Group=3&Is_Zipped=0',
                      headers=headers, 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: 471949 entries, 0 to 471948
Data columns (total 37 columns):
fl_date                  471949 non-null datetime64[ns]
unique_carrier           471949 non-null object
airline_id               471949 non-null int64
tail_num                 467903 non-null object
fl_num                   471949 non-null int64
origin_airport_id        471949 non-null int64
origin_airport_seq_id    471949 non-null int64
origin_city_market_id    471949 non-null int64
origin                   471949 non-null object
origin_city_name         471949 non-null object
origin_state_nm          471949 non-null object
dest_airport_id          471949 non-null int64
dest_airport_seq_id      471949 non-null int64
dest_city_market_id      471949 non-null int64
dest                     471949 non-null object
dest_city_name           471949 non-null object
dest_state_nm            471949 non-null object
crs_dep_time             471949 non-null int64
dep_time                 441622 non-null float64
dep_delay                441622 non-null float64
taxi_out                 441266 non-null float64
wheels_off               441266 non-null float64
wheels_on                440453 non-null float64
taxi_in                  440453 non-null float64
crs_arr_time             471949 non-null int64
arr_time                 440453 non-null float64
arr_delay                439620 non-null float64
cancelled                471949 non-null float64
cancellation_code        30852 non-null object
diverted                 471949 non-null float64
distance                 471949 non-null float64
carrier_delay            119994 non-null float64
```

```
weather_delay            119994 non-null float64
nas_delay                119994 non-null float64
security_delay           119994 non-null float64
late_aircraft_delay      119994 non-null float64
unnamed: 36              0 non-null float64
dtypes: datetime64[ns](1), float64(17), int64(10), object(9)
memory usage: 133.2+ 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']]
```

	fl_date	tail_num
10	2014-01-01	N3LGAA
11	2014-01-01	N368AA
12	2014-01-01	N3DDAA
13	2014-01-01	N332AA
14	2014-01-01	N327AA
15	2014-01-01	N3LBAA

However this simple little operation hides some complexity. What if, rather than our default `range(n)` index, we had an integer index like

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

fl_date unique_carrier airline_id 19393 2014-01-01 WN 19690 2014-01-01 HA 19790 2014-01-01 DL 19805 2014-01-01 AA 19930 2014-01-01 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 UA 2014-01-01 N14214 US 2014-01-01 N650AW VX 2014-01-01 N637VA WN 2014-01-01 N412WN

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 AA 2014-01-01 N338AA AS 2014-01-01 N524AS DL 2014-01-01 N911DL

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

fl_date airline_id unique_carrier AA 2014-01-01 19805 AS 2014-01-01 19930 DL 2014-01-01 19790

`.ix` is still around, and isn't being deprecated any time soon. Occasionally it's useful. 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. 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
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  \
unique_carrier origin dest tail_num fl_date
AA             ABQ    DFW  N200AA   2014-01-06          19805    1662
                                    2014-01-27          19805    1090
                           N202AA   2014-01-27          19805    1332
                           N426AA   2014-01-09          19805    1662
                                    2014-01-15          19805    1467
```

```
                                                 origin_airport_id  \
unique_carrier origin dest tail_num fl_date
AA             ABQ    DFW  N200AA   2014-01-06               10140
                                    2014-01-27               10140
                           N202AA   2014-01-27               10140
                           N426AA   2014-01-09               10140
                                    2014-01-15               10140

                                                origin_airport_seq_id
unique_carrier origin dest tail_num fl_date
AA             ABQ    DFW  N200AA   2014-01-06                1014002
                                    2014-01-27                1014002
                           N202AA   2014-01-27                1014002
                           N426AA   2014-01-09                1014002
                                    2014-01-15                1014002
```

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  N200AA   2014-01-06     1246.0       71.0
                                    2014-01-27      605.0        0.0
                           N202AA   2014-01-27      822.0      -13.0
                           N426AA   2014-01-09     1135.0        0.0
                                    2014-01-15     1022.0       -8.0
...                                                   ...        ...
US             TUS    PHX  N824AW   2014-01-16     1900.0      -10.0
                                    2014-01-20     1903.0       -7.0
                           N836AW   2014-01-08     1928.0       18.0
                                    2014-01-29     1908.0       -2.0
                           N837AW   2014-01-10     1902.0       -8.0

[139194 rows x 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 our 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  N200AA   2014-01-12     603.0       -7.0
                                    2014-01-17     751.0      101.0
                           N424AA   2014-01-10    1759.0       -1.0
                                    2014-01-15    1818.0       18.0
                           N426AA   2014-01-07    1835.0       35.0
...                                                  ...        ...
US             ORD    PHX  N806AW   2014-01-26    1406.0       -4.0
                           N830AW   2014-01-28    1401.0       -9.0
                           N833AW   2014-01-10    1500.0       50.0
                           N837AW   2014-01-19    1408.0       -2.0
                           N839AW   2014-01-14    1406.0       -4.0

[5205 rows x 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  N200AA   2014-01-12     603.0       -7.0
                                    2014-01-17     751.0      101.0
                           N424AA   2014-01-10    1759.0       -1.0
                                    2014-01-15    1818.0       18.0
                           N426AA   2014-01-07    1835.0       35.0
...                                                  ...        ...
WN             DSM    MDW  N941WN   2014-01-17    1759.0       14.0
                           N943WN   2014-01-10    2229.0      284.0
                           N963WN   2014-01-22     656.0       -4.0
                           N967WN   2014-01-30     654.0       -6.0
                           N969WN   2014-01-19    1747.0        2.0
```

```
[22380 rows x 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 use `IndexSlice` since `hdf.loc[(:, ['ORD', 'DSM'])]` isn't valid python syntax. Now we can slice to our heart's content; all flights from O'Hare to Des Moines in the first half of January? Sure, why not?

```
hdf.loc[pd.IndexSlice[:, 'ORD', 'DSM', :, '2014-01-01':'2014-01-15'],
        ['dep_time', 'dep_delay', 'arr_time', 'arr_delay']]
```

```
                                             dep_time  dep_delay  arr_time  \
unique_carrier origin dest tail_num fl_date
EV             ORD    DSM  NaN     2014-01-07      NaN        NaN       NaN
                           N11121  2014-01-05      NaN        NaN       NaN
                           N11181  2014-01-12   1514.0        6.0    1625.0
                           N11536  2014-01-10   1723.0        4.0    1853.0
                           N11539  2014-01-01   1127.0      127.0    1304.0
...                                                ...        ...       ...
UA             ORD    DSM  N24212  2014-01-09   2023.0        8.0    2158.0
                           N73256  2014-01-15   2019.0        4.0    2127.0
                           N78285  2014-01-07   2020.0        5.0    2136.0
                                   2014-01-13   2014.0       -1.0    2114.0
                           N841UA  2014-01-11   1825.0       20.0    1939.0

                                                arr_delay
unique_carrier origin dest tail_num fl_date
EV             ORD    DSM  NaN      2014-01-07        NaN
                           N11121   2014-01-05        NaN
                           N11181   2014-01-12       -2.0
                           N11536   2014-01-10       19.0
                           N11539   2014-01-01      149.0
...                                                   ...
UA             ORD    DSM  N24212   2014-01-09       34.0
                           N73256   2014-01-15        3.0
                           N78285   2014-01-07       12.0
                                    2014-01-13      -10.0
                           N841UA   2014-01-11       19.0
```

```
[153 rows x 4 columns]
```

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.