**19 Essential Snippets in Pandas**

Jeff Delaney

After playing around with [Pandas Python Data Analysis Library](http://pandas.pydata.org/) for about a month, I’ve compiled a pretty large list of useful snippets that I find myself reusing over and over again. These tips can save you some time sifting through the comprehensive Pandas docs.

For this article, we are starting with a DataFrame filled with Pizza orders. If you’re brand new to Pandas, here’s a few translations and key terms.

* DataFrame - Indexed rows and columns of data, like a spreadsheet or database table.
* Series = Single column of data.
* Axis - 0 == Rows, 1 == Columns
* Shape - (number\_of\_rows, number\_of\_columns) in a DataFrame

**1. Importing a CSV File**

There are a ton of options for the read\_csv function that can simplify preprocessing of data. Nobody want to waste time cleaning data, so see if you can knock it out when import the initial file.

df = pd.read\_csv('pizza.csv')

**Need to parse dates?** Just pass in the corresponding column name(s).

df = pd.read\_csv('pizza.csv', parse\_dates=['dates'])

**Only need a few specific columns?**

df = pd.read\_csv('pizza.csv', usecols=['foo', 'bar'])

**2. Exploring Data in a DataFrame**

The first thing you probably want to do is see what the data looks like. Here a few ways to check out Pandas data.

df.head() # first five rows

df.tail() # last five rows

df.sample(5) # random sample of rows

df.shape # number of rows/columns in a tuple

df.describe() # calculates measures of central tendency

df.info() # memory footprint and datatypes

Here’s the head of the pizza DataFrame…

|  | **order\_number** | **date** | **size** | **topping** | **price** | **discount** | **coupon** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | PZZA0001 | 08/21/16 | Small | Anchovies | 12.99 | 3.5 | Yes |
| **1** | PZZA0000 | 09/26/16 | Large | Pepperoni | 14.50 | 0.0 | No |
| **2** | PZZA0001 | 09/27/16 | Extra Large | Bell Pepper | 19.99 | 0.0 | No |
| **3** | PZZA0002 | 09/28/16 | Extra Large | Olives | 20.99 | 5.0 | Yes |
| **4** | PZZA0003 | 09/29/16 | Extra Large | Pepperoni | 21.99 | 0.0 | No |

**3. Adding a New Column to a DataFrame**

The quick and easy way is to just define a new column on the dataframe. This will give us column with the number 23 on every row. Usually, you will be setting the new column with an array or Series that matches the number of rows in the data.

df['new\_column'] = 23

Need to build a new column based on values from other columns?

full\_price = (df.price + df.discount)

df['original\_price'] = full\_price

Need the column in a certain order? The first argument is the position of the column. This will put the column at the begining of the DataFrame.

df.insert(0, 'original\_price', full\_price)

**4. Select a Specific “Cell” Value**

By cell I mean a single row/column intersection, like those in an Excel spreadsheet. You would expect this to be simple, but the syntax is not very obvious. There are three methods in Pandas that almost do the same thing, .loc, iloc, .ix – adding to the confusion for newcomers.

Typically, I use .ix because it allows a mix of integers and strings. Enter the index of the row first, then the column.

df.ix[2, 'topping']

You can also select the column first with dot notation, then the row index, which looks a little cleaner.

df.topping.ix[2]

Either method will return the value of the cell.

>>> 'Bell Pepper'

**5.Filtering DataFrames with Conditional Logic**

Let’s the we need to analyze orders that have pineapple in the topping column.

filtered\_data = df[df.topping == 'pineapple']

Or that meet a certain price threshold

filtered\_data = df[df.price > 11.99 ]

How about both at the same time? Just add the conditions to tuples and connect them with a bitwise operator.

filtered\_data = df[(df.price > 11.99) & (df.topping == 'Pineapple')]

Now we have all the pizzas with a Pineapple topping priced over 11.99.

|  | **order\_number** | **date** | **size** | **topping** | **price** | **discount** | **coupon** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **6** | PZZA0006 | 10/01/16 | Medium | Pineapple | 17.50 | 0.0 | No |
| **9** | PZZA0009 | 10/04/16 | Medium | Pineapple | 12.99 | 2.0 | Yes |

**6. Sorting a DataFrame by a Certain Column**

Pretty self-explanatory, but very useful.

df.sort\_values('price', axis=0, ascending=False)

**7. Apply a Function to Every Row in a Column**

Anonymous lambda functions in Python are useful for these tasks. Let’s say we need to calculate taxes for every row in the DataFrame with a custom function. The pandas apply method allows us to pass a function that will run on every value in a column. In this example, we extract a new taxes feature by running a custom function on the price data.

def calculate\_taxes(price):

taxes = price \* 0.12

return taxes

df['taxes'] = df.price.apply(calculate\_taxes)

|  | **order\_number** | **price** | **taxes** |
| --- | --- | --- | --- |
| **0** | PZZA0000 | 12.99 | 1.5588 |
| **1** | PZZA0001 | 14.50 | 1.7400 |
| **2** | PZZA0002 | 19.99 | 2.3988 |
| **3** | PZZA0003 | 20.99 | 2.5188 |
| **4** | PZZA0004 | 21.99 | 2.6388 |

**8. Add a New Column with Conditional Logic**

The where function in numpy is useful when extracting features with conditional logic. Let’s imagine the pizza parlor is only profitable on sales above $15.00. We create a new column based on this insight like so:

df['profitable'] = np.where(df['price']>=15.00, True, False)

**9. Finding the Mean or Standard Deviation of Multiple Columns or Rows**

If you have a DataFrame with the same type of data in every column, possibly a time series with financial data, you may need to find he mean horizontally.

df['mean'] = df.mean(axis=1)

or to find the standard deviation vertically

df.std(axis=0)

**10. Converting a DataFrame to a Numpy Array**

Converting the the values in a DataFrame to an array is simple

df.values

If you want to preserve the table presentation

df.as\_matrix

**11. Combining DataFrames with Concatenation**

You can concatenate rows or columns together, the only requirement is that the shape is the same on corresponding axis. To concat rows vertically:

pd.concat([df\_1, df\_2], axis=0)

Or to concat columns horizontally:

pd.concat([df\_1, df\_2], axis=1)

**12. Combining DataFrames based on an Index Key**

Merging in Pandas works just like SQL. If you you have two DataFrames that share a key, perhaps a pizza ‘order\_id’, you can perform inner, outer, left, right joins just like you would in SQL.

merged\_df = df\_1.merge(df\_2, how='left', on='order\_id')

**13. Converting Dates to their own Day, Week, Month, Year Columns**

First, make sure the data is in datetime format. Then use dt method to extract the data you need.

date = pd.to\_datetime(df.date)

df['weekday'] = date.dt.weekday

df['year'] = date.dt.year

**14. Finding NaNs in a DataFrame**

Count the total number of NaNs present:

df.isnull().sum().sum()

List the NaN count for each column:

df.isnull().sum()

**15. Filling NaNs or Missing Data**

Most machine learning algorithms do not like NaN values, so you’ll probably need to convert them. If the topping column is missing some values, we can fill them a default value.

df.topping = df.topping.fillna('Cheese')

or we can drop any row missing data across the entire DataFrame:

df = df.dropna(axis=0)

**16. Extracting Features by Grouping Columns**

Grouping columns is a great way to extract features from data. This is especially useful when you have data that can be counted or quantified in some way. For example, you might have group pizzas by topping, then calculate the mean for price in each group.

df.groupby('topping')['discount'].apply(lambda x: np.mean(x))

or maybe you want to see the count of a certain value

df.groupby('topping')['discount'].apply(lambda x: x.count())

topping

Anchovies 3

Bell Pepper 1

Cheese 2

Olives 1

Pepperoni 3

Pineapple 2

Veggie 1

Name: discount, dtype: int64

**17.Creating Bins**

Let’s say we want to create 3 separate bins for different price ranges. This is especially useful for simplifying noisy data.

bins = [0, 5, 15, 30]

names = ['Cheap', 'Normal', 'Expensive']

df['price\_point'] = pd.cut(df.price, bins, labels=names)

|  | **order\_number** | **price** | **price\_point** |
| --- | --- | --- | --- |
| **0** | PZZA0000 | 12.99 | Normal |
| **1** | PZZA0001 | 14.50 | Normal |
| **2** | PZZA0002 | 19.99 | Expensive |
| **3** | PZZA0003 | 20.99 | Expensive |
| **4** | PZZA0004 | 21.99 | Expensive |

**18. Creating a new Column by Looping**

Let’s say we want to categorize toppings by ‘vegetable’ or ‘meat’. Dealing with nominal values like these can be handled with a for loop. (Note: you can also use the apply function described earlier to perform this task. )

topping\_type = []

for row in df.topping:

if row in ['pepperoni', 'chicken', 'anchovies']:

topping\_type.append('meat')

else:

topping\_type.append('vegetable')

df['topping\_type'] = topping\_type

**19. Loading Massive Datasets in Smaller Chunks**

Sometimes you might have a massive file that will max out your RAM and crash your system. In that case, you might need to analyze the file in smaller chunks.

chunksize = 500

chunks = []

for chunk in pd.read\_csv('pizza.csv', chunksize=chunksize):

# Do stuff...

chunks.append(chunk)

df = pd.concat(chunks, axis=0)