

1. Show installed versions

<code>pd.__version__</code>	Show Python version
<code>pd.show_versions()</code>	Show dependency version

2. Create an example DataFrame

<code>df = pd.DataFrame({'col one':[100, 200], 'col two':[300, 400]})</code>	Pass a dictionary to the DataFrame constructor, keys are the column names and the dictionary are the values
<code>pd.DataFrame(np.random.rand(4, 8))</code>	Use the rand function to create a larger data frame
<code>pd.DataFrame(np.random.rand(4, 8), columns=list('abcdefgh'))</code>	If you want non-numeric column names

3. Rename columns

<code>df = df.rename({'col one':'col_one', 'col two':'col_two'}, axis='columns')</code>	Pass a dictionary; keys are the old names and the values are the new names, and you also specify the axis
<code>df.columns = ['col_one', 'col_two']</code>	if you're going to rename all of the columns at once, a simpler method is just to overwrite the columns attribute of the DataFrame
<code>df.columns = df.columns.str.replace(' ', '_')</code>	To replace spaces with underscores, use the str.replace() method
<code>df.add_prefix('X_')</code>	Add a prefix
<code>df.add_suffix('_Y')</code>	Add a suffix

4. Reverse row order

<code>drinks.loc[::-1]</code>	The most straightforward method is to use the loc accessor
<code>drinks.loc[::-1].reset_index(drop=True)</code>	Reset the index; use reset_index() to drop the old index entirely

5. Reverse column order

<code>drinks.loc[:, ::-1]</code>	Use loc to reverse the left-to-right order of your columns
----------------------------------	--

6. Select columns by data type

<code>drinks.select_dtypes(include='number')</code>	To select only the numeric columns
<code>drinks.select_dtypes(include=['number', 'object', 'category', 'datetime'])</code>	To include multiple data types by passing a list
<code>drinks.select_dtypes(exclude='number')</code>	To exclude certain data types

7. Convert strings to numbers

<code>df.astype({'col_one':'float', 'col_two':'float'}).dtypes</code>	to convert the data types to numeric. You can use the astype() method
<code>pd.to_numeric(df.col_three, errors='coerce').fillna(0)</code>	use the to_numeric() function on. If you know that the NaN values actually represent zeros, you can fill them with zeros using the fillna() method the third column and tell it to convert any invalid input into NaN values
<code>df = df.apply(pd.to_numeric, errors='coerce').fillna(0)</code>	To apply this function to the entire DataFrame all at once by using the apply() method

8. Reduce DataFrame size

<code>drinks.info(memory_usage='deep')</code>	Review the memory usage of a dataframe
<code>cols = ['beer_servings', 'continent'] small_drinks = pd.read_csv('http://bit.ly/drinksbycountry', usecols=cols)</code>	The first step is to only read in the columns that you actually need, which we specify with the "usecols" parameter
<code>dtypes = {'continent':'category'} smaller_drinks = pd.read_csv('http://bit.ly/drinksbycountry', usecols=cols, dtype=dtypes)</code>	The second step is to convert any object columns containing categorical data to the category data type, which we specify with the "dtype" parameter



9. Build a DataFrame from multiple files (row)

<code>from glob import glob</code>	Use the glob module
<code>stock_files = sorted(glob('data/-stocks*.csv'))</code>	In this case, glob is looking in the "data" subdirectory for all CSV files that start with the word "-stocks"; ['data/stocks1.csv', 'data/stocks2.csv', 'data/stocks3.csv']
<code>pd.concat((pd.read_csv(file) for file in stock_files))</code>	use a generator expression to read each of the files using <code>read_csv()</code> and pass the results to the <code>concat()</code> function, which will concatenate the rows into a single DataFrame
<code>pd.concat((pd.read_csv(file) for file in stock_files), ignore_index=True)</code>	There are now duplicate values in the index. To avoid that, we can tell the <code>concat()</code> function to ignore the index and instead use the default integer index

10. Build a DF from multiple files (columns)

<code>drink_files = sorted(glob('data/drinks*.csv'))</code>	
<code>pd.concat((pd.read_csv(file) for file in drink_files), axis='columns')</code>	Tell the <code>concat()</code> function to concatenate along the columns axis

11. Create a DataFrame from the clipboard

<code>df = pd.read_clipboard()</code>	Just select the data and copy it to the clipboard. Then, you can use the <code>read_clipboard()</code> function to read it into a DataFrame
---------------------------------------	---

22. Create a pivot table

<code>titanic.pivot_table(index='Sex', columns='Pclass', values='Survived', aggfunc='mean')</code>	If you often create DataFrames, you might find it more convenient to use the <code>pivot_table()</code> method instead. With a pivot table, you directly specify the index, the columns, the values, and the aggregation function.
--	--

22. Create a pivot table (cont)

<code>titanic.pivot_table(index='Sex', columns='Pclass', values='Survived', aggfunc='mean', margins=True)</code>	An added benefit of a pivot table is that you can easily add row and column totals by setting <code>margins=True</code>
<code>titanic.pivot_table(index='Sex', columns='Pclass', values='Survived', aggfunc='count', margins=True)</code>	Create a cross-tabulation just by changing the aggregation function from "mean" to "count"

23. Convert continuous data into categorical data

<code>pd.cut(titanic.Age, bins=[0, 18, 25, 99], labels=['child', 'young adult', 'adult']).head(10)</code>	Label the age ranges, such as "child", "young adult", and "adult". The best way to do this is by using the <code>cut()</code> function. This assigned each value to a bin with a label. Ages 0 to 18 were assigned the label "child", ages 18 to 25 were assigned the label "young adult", and ages 25 to 99 were assigned the label "adult".
---	---

24. Change display options

<code>pd.set_option('display.float_format', '{:.2f}'.format)</code>	To standardise the display to use 2 decimal places
<code>pd.reset_option('display.float_format')</code>	Reset any option back to its default

25. Style a DataFrame

<code>format_dict = {'Date': '{:%m/%d/%y}', 'Close': '\${:.2f}', 'Volume': '{:}'}</code>	Create a dictionary of format strings that specifies how each column should be formatted
<code>stocks.style.format(format_dict)</code>	Pass it to the DataFrame's <code>style.format()</code> method
<code>(stocks.style.format(format_dict).hide_index().highlight_min('Close', color='red').highlight_max('Close', color='lightgreen'))</code>	We've now hidden the index, highlighted the minimum Close value in red, and highlighted the maximum Close value in green
<code>(stocks.style.format(format_dict).hide_index().background_gradient(subset='Volume', cmap='Blues'))</code>	Highlight the minimum Close value in red, and highlighted the maximum Close value in green

25. Style a DataFrame (cont)

<code>(stocks.style.format(format_dict) .hide_index() .bar("Volume", color='lightblue', align='zero') .set_caption('Stock Prices from October 2016'))</code>	There's now a bar chart within the Volume column and a caption above the DataFrame.
---	---

21. Reshape a MultiIndexed Series

<code>titanic.groupby('Sex').Survived.mean()</code>	If you wanted to calculate the survival rate by a single category such as "Sex", you would use a <code>groupby()</code>
<code>titanic.groupby(['Sex', 'Pclass']).Survived.mean()</code>	If you wanted to calculate the survival rate across two different categories at once, you would <code>groupby()</code> both of those categories
<code>titanic.groupby(['Sex', 'Pclass']).Survived.mean().unstack()</code>	It can be hard to read and interact with data in this format, so it's often more convenient to reshape a MultiIndexed Series into a DataFrame by using the <code>unstack()</code> method

20. Select a slice of rows and columns

<code>titanic.describe()</code>	If you wanted a numerical summary of the dataset, you would use the <code>describe()</code> method
<code>titanic.describe().loc['min':'max']</code>	If you wanted to filter it to only show the "five-number summary", you can use the <code>loc</code> accessor and pass it a slice of the "min" through the "max" row labels
<code>titanic.describe().loc['min':'max', 'Pclass':'Parch']</code>	And if you're not interested in all of the columns, you can also pass it a slice of column labels

18. Aggregate by multiple functions

<code>orders[orders.order_id == 1].item_price.sum()</code>	Each order has an <code>order_id</code> and consists of one or more rows. To figure out the total price of an order, you sum the <code>item_price</code> for that <code>order_id</code> . For example, here's the total price of order number 1
<code>orders.groupby('order_id').item_price.sum()</code>	If you wanted to calculate the total price of every order, you would <code>groupby()</code> <code>order_id</code> and then take the sum of <code>item_price</code> for each group
<code>orders.groupby('order_id').item_price.agg(['sum', 'count'])</code>	However, you're not actually limited to aggregating by a single function such as <code>sum()</code> . To aggregate by multiple functions, you use the <code>agg()</code> method and pass it a list of functions such as <code>sum()</code> and <code>count()</code>

19. Combine the output of an aggregation with a DF

<code>orders.groupby('order_id').item_price.sum()</code>	What if we wanted to create a new column listing the total price of each order? Recall that we calculated the total price using the <code>sum()</code> method
<code>len(orders.groupby('order_id').item_price.sum())</code>	In other words, the output of the <code>sum()</code> function
<code>len(orders.item_price)</code>	...is smaller than the input to the function
<code>total_price = orders.groupby('order_id').item_price.transform('sum') len(total_price)</code>	The solution is to use the <code>transform()</code> method, which performs the same calculation but returns output data that is the same shape as the input data

This needs more work!



By **Ianh**
cheatography.com/ianh/

Not published yet.
 Last updated 12th July, 2019.
 Page 3 of 4.

Sponsored by **Readable.com**
 Measure your website readability!
<https://readable.com>

16. Split a string into multiple columns

`df.name.str.split(' ', expand=True)`

What if we wanted to split the "name" column into three separate columns, for first, middle, and last name? We would use the `str.split()` method and tell it to split on a space character and expand the results into a DataFrame

`df[['first', 'middle', 'last']] =`

These three columns can actually be saved to the original DataFrame in a single assignment statement

`df.name.str.split(' ', expand=True) df`

`df.location.str.split(',', expand=True)`

What if we wanted to split a string, but only keep one of the resulting columns? For example, let's split the location column on "comma space"

`df['city'] = df.location.str.split(',', expand=True)[0]`

If we only cared about saving the city name in column 0, we can just select that column and save it to the DataFrame

13. Filter a DataFrame by multiple categories

`movies[(movies.genre == 'Action') | (movies.genre == 'Drama') | (movies.genre == 'Western')]`

If we wanted to filter the DataFrame to only show movies with the genre Action or Drama or Western, we could use multiple conditions separated by the "or" operator

`movies[movies.genre.isin(['Action', 'Drama', 'Western'])]`

However, you can actually rewrite this code more clearly by using the `isin()` method and passing it a list of genres

`movies[~movies.genre.isin(['Action', 'Drama', 'Western'])]`

And if you want to reverse this filter, so that you are excluding (rather than including) those three genres, you can put a tilde in front of the condition

14. Filter a DataFrame by largest categories

`counts.nlargest(3)`

The Series method `nlargest()` makes it easy to select the 3 largest values in this Series

`counts.nlargest(3).index`

And all we actually need from this Series is the index

`movies[movies.genre.isin(counts.nlargest(3).index)]`

Finally, we can pass the index object to `isin()`, and it will be treated like a list of genres

15. Handle missing values

`ufo.isna().sum()`

To find out how many values are missing in each column, you can use the `isna()` method and then take the `sum()`

`ufo.isna().mean()`

Similarly, you can find out the percentage of values that are missing by taking the `mean()` of `isna()`

`ufo.dropna(axis='columns')`

If you want to drop the columns that have any missing values, you can use the `dropna()` method

`ufo.dropna(thresh=len(ufo)*0.9, axis='columns')`

Or if you want to drop columns in which more than 10% of the values are missing, you can set a threshold for `dropna()`

