Michael Salmon
Aspiring data scientist. Still figuring my life out.
Jul 6, 2017 · 10 min read

Helpful Python Code Snippets for Data Exploration in Pandas

For anyone new to data exploration, cleaning, or analysis using Python, Pandas will quickly become one of your most frequently used and reliable tools. It is extremely versatile in its ability to work with a wide variety of existing data files (including csv, excel, json, html, and sql, among others), and can easily assemble data from lists or dictionaries into standard “data frames” that effectively display data in tabular form for easy manipulation.

Despite (and perhaps because of) pandas’ versatility in exploring and manipulating data, it can be easy for programmers to find themselves reusing or adapting code used in previous work to perform similar operations for new projects. Rather than revisiting old project files for the appropriate code, it can be handy to maintain a reference file that can house useful code snippets in a singular location.

As such, I wanted to provide some of the most helpful, shorthand pandas code snippets I have utilized in the past. This shouldn’t be taken as a definitive list of pandas code snippets; however, these should help in addressing many of the initial data exploration and manipulation tasks one might perform on data of interest. I have accumulated these from a variety of sources, though I want to give special credit to Joseph

Nelson for providing much of the initial structure for the outlay below. Other great resources for troubleshooting pythonic coding problems include Chris Albon's blog, and the ever faithful Stack Overflow.

Without further ado, I hope you find some of the following code snippets useful. If nothing else, I know that I can come back here to reference these for myself if need be. Note, that I've abstracted most of these snippets to refer to a data frame as "df", and various columns as "column_x" and so on.

```
#Code snippets for Pandas
import pandas as pd
```

```
'''
Reading Files, Selecting Columns, and Summarizing
'''
```

```
# reading in a file from local computer or directly from a
URL
# various file formats that can be read in out wrote out
'''
Format Type     Data Description      Reader
Writer
text                  CSV            read_csv
to_csv
text                 JSON            read_json
to_json
text                 HTML            read_html
to_html
text             Local clipboard  read_clipboard
to_clipboard
binary             MS Excel          read_excel
to_excel
binary            HDF5 Format        read_hdf
to_hdf
binary           Feather Format     read_feather
to_feather
binary              Msgpack         read_msgpack
to_msgpack
binary               Stata           read_stata
to_stata
binary                SAS             read_sas
binary        Python Pickle Format   read_pickle
to_pickle
SQL                   SQL             read_sql
to_sql
SQL             Google Big Query      read_gbq
to_gbq
'''
#to read about different types of files, and further
functionality of reading in files, visit:
http://pandas.pydata.org/pandas-docs/version/0.20/io.html
```

```
df = pd.read_csv('local_path/file.csv')
df = pd.read_csv('https://file_path/file.csv')
```

```
# when reading in tables, can specify separators, and note a
column to be used as index separators can include tabs
(“\t”), commas(“,”), pipes (“|”), etc.
```

```
df = pd.read_table(‘https://file_path/file', sep=’|’,
index_col=’column_x’)
```

```
# examine the df data
df           # print the first 30 and last 30 rows
type(df)     # DataFrame
df.head()    # print the first 5 rows
df.head(10)  # print the first 10 rows
df.tail()    # print the last 5 rows
df.index     # “the index” (aka “the labels”)
df.columns   # column names (which is “an index”)
df.dtypes    # data types of each column
df.shape     # number of rows and columns
df.values    # underlying numpy array — df are stored as
numpy arrays for effeciencies.
```

```
# select a column
df[‘column_y’]         # select one column
type(df[‘column_y’])   # determine datatype of column (e.g.,
Series)
df.column_y            # select one column using the
DataFrame attribute — not effective if column names have
spaces
```

```
# summarize (describe) the DataFrame
df.describe()          # describe all numeric columns
df.describe(include=[‘object’]) # describe all object
columns
df.describe(include=’all’)      # describe all columns
```

```
# summarize a Series
df.column_y.describe()   # describe a single column
df.column_z.mean()       # only calculate the mean
df[“column_z”].mean()    # alternate method for calculating
mean


# count the number of occurrences of each value
df.column_y.value_counts()   # most useful for categorical
variables, but can also be used with numeric variables
```

```
#filter df by one column, and print out values of another
column
#when using numeric values, no quotations
df[df.column_y == “string_value”].column_z
df[df.column_y == 20 ].column_z


# display only the number of rows of the ‘df’ DataFrame
df.shape[0]
```

```
# display the 3 most frequent occurances of column in 'df'
df.column_y.value_counts()[0:3]
```

```
'''
Filtering and Sorting
'''
```

```
# boolean filtering: only show df with column_z < 20
filter_bool = df.column_z < 20    # create a Series of
booleans…
df[filter_bool]               # …and use that Series to
filter rows
df[filter_bool].describe()    # describes a data frame
filtered by filter_bool
df[df.column_z < 20]          # or, combine into a single
step
df[df.column_z < 20].column_x  # select one column from the
filtered results
df[df["column_z"] < 20].column_x     # alternate method
df[df.column_z < 20].column_x.value_counts()   #
value_counts of resulting Series, can also use .mean(), etc.
instead of .value_counts()
```

```
# boolean filtering with multiple conditions; indexes are in
square brackets, conditions are in parens
```

```
df[(df.column_z < 20) & (df.column_y=='string')] # ampersand
for AND condition
df[(df.column_z < 20) | (df.column_z > 60)] # pipe for OR
condition
```

```
# sorting
df.column_z.order()          # sort a column
df.sort_values('column_z')   # sort a DataFrame by a single
column
df.sort_values('column_z', ascending=False)     # use
descending order instead
```

```
# Sort dataframe by multiple columns
df = df.sort(['col1','col2','col3'],ascending=[1,1,0])


# can also filter 'df' using pandas.Series.isin
df[df.column_x.isin(["string_1", "string_2"])]
```

```
'''
Renaming, Adding, and Removing Columns
'''
```

```
# rename one or more columns
df.rename(columns={'original_column_1':'column_x',
'original_column_2':'column_y'}, inplace=True) #saves
changes


# replace all column names (in place)
new_cols = ['column_x', 'column_y', 'column_z']
df.columns = new_cols
```

```
# replace all column names when reading the file
df = pd.read_csv('df.csv', header=0, names=new_cols)
```

```
# add a new column as a function of existing columns
df['new_column_1'] = df.column_x + df.column_y
df['new_column_2'] = df.column_x * 1000   #can create new
columns without for loops
```

```
# removing columns
df.drop('column_x', axis=1)   # axis=0 for rows, 1 for
columns - does not drop in place
df.drop(['column_x', 'column_y'], axis=1, inplace=True) #
drop multiple columns
```

```
# Lower-case all DataFrame column names
df.columns = map(str.lower, df.columns)
```

```
# Even more fancy DataFrame column re-naming
# lower-case all DataFrame column names (for example)
df.rename(columns=lambda x: x.split('.')[-1], inplace=True)



'''
Handling Missing Values
'''
```

```
# missing values are usually excluded by default
df.column_x.value_counts()            # excludes missing
values
df.column_x.value_counts(dropna=False) # includes missing
values
```

```
# find missing values in a Series
df.column_x.isnull()  # True if missing
df.column_x.notnull() # True if not missing
```

```
# use a boolean Series to filter DataFrame rows
df[df.column_x.isnull()]  # only show rows where column_x is
missing
df[df.column_x.notnull()] # only show rows where column_x is
not missing
```

```
# understanding axes
df.sum()       # sums "down" the 0 axis (rows)
df.sum(axis=0) # equivalent (since axis=0 is the default)
df.sum(axis=1) # sums "across" the 1 axis (columns)
```

```
# adding booleans
pd.Series([True, False, True])       # create a boolean
Series
pd.Series([True, False, True]).sum() # converts False to 0
and True to 1
```

```
# find missing values in a DataFrame
df.isnull() # DataFrame of booleans
df.isnull().sum() # count the missing values in each column
```

```
# drop missing values
df.dropna(inplace=True)   # drop a row if ANY values are
missing, defaults to rows, but can be applied to columns
with axis=1
```

```
df.dropna(how='all', inplace=True)  # drop a row only if ALL
values are missing
```

```
# fill in missing values
df.column_x.fillna(value='NA', inplace=True)
# fill in missing values with 'NA'
# value does not have to equal a string — can be set as some
calculated value like df.column_x.mode(), or just a number
like 0



# turn off the missing value filter
df = pd.read_csv('df.csv', header=0, names=new_cols,
na_filter=False)
```

```
'''
Split-Apply-Combine
Diagram: http://i.imgur.com/yjNkiwL.png
'''
```

```
# for each value in column_x, calculate the mean column_y
df.groupby('column_x').column_y.mean()
```

```
# for each value in column_x, count the number of
occurrences
df.column_x.value_counts()
```

```
# for each value in column_x, describe column_y
df.groupby('column_x').column_y.describe()
```

```
# similar, but outputs a DataFrame and can be customized
df.groupby('column_x').column_y.agg(['count', 'mean', 'min',
'max'])
```

```
df.groupby('column_x').column_y.agg(['count', 'mean', 'min',
'max']).sort_values('mean')
```

```
# if you don't specify a column to which the aggregation
function should be applied, it will be applied to all
numeric columns
```

```
df.groupby('column_x').mean()
df.groupby('column_x').describe()
```

```
# can also groupby a list of columns, i.e., for each
combination of column_x and column_y, calculate the mean
column_z
df.groupby([“column_x”,”column_y”]).column_z.mean()
```

```
#to take groupby results out of hierarchical index format
(e.g., present as table), use .unstack() method
df.groupby(“column_x”).column_y.value_counts().unstack()
```

```
#conversely, if you want to transform a table into a
hierarchical index, use the .stack() method
df.stack()
```

```
‘’’
Selecting Multiple Columns and Filtering Rows
‘’’
```

```
# select multiple columns
my_cols = [‘column_x’, ‘column_y’]  # create a list of
column names…
df[my_cols]                  # …and use that list to select
columns
df[[‘column_x’, ‘column_y’]]  # or, combine into a single
step — double brackets due to indexing a list.
```

```
# use loc to select columns by name
df.loc[:, ‘column_x’]    # colon means “all rows”, then
select one column
df.loc[:, [‘column_x’, ‘column_y’]]  # select two columns
df.loc[:, ‘column_x’:’column_y’]     # select a range of
columns (i.e., selects all columns including first through
last specified)
```

```
# loc can also filter rows by “name” (the index)
df.loc[0, :]       # row 0, all columns
df.loc[0:2, :]     # rows 0/1/2, all columns
df.loc[0:2, ‘column_x’:’column_y’] # rows 0/1/2, range of
columns
```

```
# use iloc to filter rows and select columns by integer
position
df.iloc[:, [0, 3]]     # all rows, columns in position 0/3
df.iloc[:, 0:4]        # all rows, columns in position
0/1/2/3
df.iloc[0:3, :]        # rows in position 0/1/2, all columns
```

```
#filtering out and dropping rows based on condition (e.g.,
where column_x values are null)
drop_rows = df[df[“column_x”].isnull()]
new_df = df[~df.isin(drop_rows)].dropna(how=’all’)
```

```
‘’’
Merging and Concatenating Dataframes
‘’’
#concatenating two dfs together (just smooshes them
together, does not pair them in any meaningful way) - axis=1
```

```
concats df2 to right side of df1; axis=0 concats df2 to
bottom of df1
new_df = pd.concat([df1, df2], axis=1)
```

```
#merging dfs based on paired columns; columns do not need to
have same name, but should match values; left_on column
comes from df1, right_on column comes from df2
new_df = pd.merge(df1, df2, left_on='column_x',
right_on='column_y')
```

```
#can also merge slices of dfs together, though slices need
to include columns used for merging
new_df = pd.merge(df1[['column_x1', 'column_x2']], df2,
left_on='column_x2', right_on='column_y')
```

```
#merging two dataframes based on shared index values (left
is df1, right is df2)
new_df = pd.merge(df1, df2, left_index=True,
right_index=True)
```

```
'''
Other Frequently Used Features
'''
```

```
# map existing values to a different set of values
df['column_x'] = df.column_y.map({'F':0, 'M':1})
```

```
# encode strings as integer values (automatically starts at
0)
df['column_x_num'] = df.column_x.factorize()[0]
```

```
# determine unique values in a column
df.column_x.nunique()   # count the number of unique values
df.column_x.unique()    # return the unique values
```

```
# replace all instances of a value in a column (must match
entire value)
df.column_y.replace('old_string', 'new_string',
inplace=True)
```

```
#alter values in one column based on values in another
column (changes occur in place)
#can use either .loc or .ix methods
```

```
df.loc[df["column_x"] == 5, "column_y"] = 1
```

```
df.ix[df.column_x == "string_value", "column_y"] =
"new_string_value"
```

```
#transpose data frame (i.e. rows become columns, columns
become rows)
df.T
```

```
# string methods are accessed via 'str'
df.column_y.str.upper() # converts to uppercase
df.column_y.str.contains('value', na='False') # checks for a
substring, returns boolean series
```

```
# convert a string to the datetime_column format
df['time_column'] = pd.to_datetime_column(df.time_column)
df.time_column.dt.hour   # datetime_column format exposes
convenient attributes
(df.time_column.max() - df.time_column.min()).days   # also
allows you to do datetime_column "math"
df[df.time_column > pd.datetime_column(2014, 1, 1)]   #
boolean filtering with datetime_column format
```

```
# setting and then removing an index, resetting index can
help remove hierarchical indexes while preserving the table
in its basic structure
df.set_index('time_column', inplace=True)
df.reset_index(inplace=True)
```

```
# sort a column by its index
df.column_y.value_counts().sort_index()
```

```
# change the data type of a column
df['column_x'] = df.column_x.astype('float')
```

```
# change the data type of a column when reading in a file
pd.read_csv('df.csv', dtype={'column_x':float})
```

```
# create dummy variables for 'column_x' and exclude first
dummy column
column_x_dummies = pd.get_dummies(df.column_x).iloc[:, 1:]
```

```
# concatenate two DataFrames (axis=0 for rows, axis=1 for
columns)
df = pd.concat([df, column_x_dummies], axis=1)
```

```
'''
Less Frequently Used Features
'''
```

```
# create a DataFrame from a dictionary
pd.DataFrame({'column_x':['value_x1', 'value_x2',
'value_x3'], 'column_y':['value_y1', 'value_y2',
'value_y3']})
```

```
# create a DataFrame from a list of lists
pd.DataFrame([['value_x1', 'value_y1'], ['value_x2',
'value_y2'], ['value_x3', 'value_y3']], columns=['column_x',
'column_y'])
```

```
# detecting duplicate rows
df.duplicated()       # True if a row is identical to a
```

```
previous row
df.duplicated().sum() # count of duplicates
df[df.duplicated()]   # only show duplicates
df.drop_duplicates()  # drop duplicate rows
df.column_z.duplicated()   # check a single column for
duplicates
df.duplicated([‘column_x’, ‘column_y’, ‘column_z’]).sum()  #
specify columns for finding duplicates
```

```
# Clean up missing values in multiple DataFrame columns
df = df.fillna({
 ‘col1’: ‘missing’,
 ‘col2’: ‘99.999’,
 ‘col3’: ‘999’,
 ‘col4’: ‘missing’,
 ‘col5’: ‘missing’,
 ‘col6’: ‘99’
})
```

```
# Concatenate two DataFrame columns into a new, single
column - (useful when dealing with composite keys, for
example)
df[‘newcol’] = df[‘col1’].map(str) + df[‘col2’].map(str)
```

```
# Doing calculations with DataFrame columns that have
missing values
# In example below, swap in 0 for df[‘col1’] cells that
contain null
df[‘new_col’] = np.where(pd.isnull(df[‘col1’]),0,df[‘col1’])
+ df[‘col2’]


# display a cross-tabulation of two Series
pd.crosstab(df.column_x, df.column_y)
```

```
# alternative syntax for boolean filtering (noted as
“experimental” in the documentation)
df.query(‘column_z < 20’) # df[df.column_z < 20]
df.query(“column_z < 20 and column_y==’string’”)  #
df[(df.column_z < 20) & (df.column_y==’string’)]
df.query(‘column_z < 20 or column_z > 60’)        #
df[(df.column_z < 20) | (df.column_z > 60)]
```

```
# Loop through rows in a DataFrame
for index, row in df.iterrows():
 print index, row[‘column_x’]
```

```
# Much faster way to loop through DataFrame rows if you can
work with tuples
for row in df.itertuples():
 print(row)
```

```
# Get rid of non-numeric values throughout a DataFrame:
for col in df.columns.values:
 df[col] = df[col].replace(‘[⁰-9]+.-’, ‘’, regex=True)
```

```
# Change all NaNs to None (useful before loading to a db)
df = df.where((pd.notnull(df)), None)
```

```
# Split delimited values in a DataFrame column into two new
columns
df['new_col1'], df['new_col2'] =
zip(*df['original_col'].apply(lambda x: x.split(': ', 1)))
```

```
# Collapse hierarchical column indexes
df.columns = df.columns.get_level_values(0)
```

```
# display the memory usage of a DataFrame
df.info()         # total usage
df.memory_usage() # usage by column
```

```
# change a Series to the 'category' data type (reduces
memory usage and increases performance)
df['column_y'] = df.column_y.astype('category')
```

```
# temporarily define a new column as a function of existing
columns
df.assign(new_column = df.column_x + df.spirit +
df.column_y)
```

```
# limit which rows are read when reading in a file
pd.read_csv('df.csv', nrows=10)        # only read first 10
rows
pd.read_csv('df.csv', skiprows=[1, 2]) # skip the first two
rows of data
```

```
# randomly sample a DataFrame
train = df.sample(frac=0.75, random_column_y=1) # will
contain 75% of the rows
test = df[~df.index.isin(train.index)] # will contain the
other 25%
```

```
# change the maximum number of rows and columns printed
('None' means unlimited)
pd.set_option('max_rows', None) # default is 60 rows
pd.set_option('max_columns', None) # default is 20 columns
print df
```

```
# reset options to defaults
pd.reset_option('max_rows')
pd.reset_option('max_columns')
```

```
# change the options temporarily (settings are restored when
you exit the 'with' block)
with pd.option_context('max_rows', None, 'max_columns',
None):
 print df
```

Thanks, all!