Pandas Cheat Sheet

Preliminaries

This cheat sheet was written for pandas version 1.4.2. It assumes you are using Python 3.9 or higher. The cheat sheet uses the Calibri and Consolas fonts. If not already installed on your device, you should download these fonts.

What is pandas?

Pandas is a powerful and useful Python library for manipulating tabular data (like a spreadsheet).

But some things differ from how python usually works:

- pandas largely uses a functional programming paradigm, not a pure object-oriented paradigm.
- The floating-point not-a-number (NaN) construct is co-opted to represent missing data.

And there are downsides:

- The pandas library is huge, with a steep learning curve, and multiple ways to do the same thing.
- Your data needs to be small enough to fit into RAM memory on your machine (under 1GB if you have 8GB RAM. Under 10GB on a 64GB machine).

Getting started

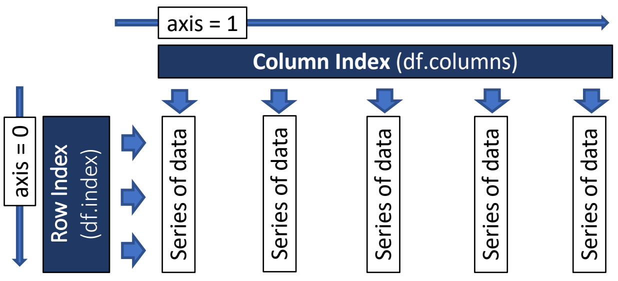
You must import the pandas library into your python code. Typically, one also imports numpy, matplotlib and seaborn.

```
import pandas as pd                 # required
import numpy as np                  # optional
import matplotlib as mpl            # optional
import matplotlib.pyplot as plt     # optional
import seaborn as sns               # optional
```

Core concepts

Pandas provides three core objects:

- <u>**Index**</u> which are like an array of numbers, strings, or dates/times, and which are used to reference the data in Series and DataFrame objects.
- <u>**Series**</u> which comprises an ordered array of data (all of one data-type), and an index to reference that data.
- <u>**DataFrame**</u> which is a table of data, where every column is like a pandas Series. A DataFrame has two indexes – one each for the rows and the columns)



Pandas aligns the index and vectorizes all arithmetic on pandas Series and DataFrame objects. For example:

```
s1 = pd.Series([1, 2, 3], index=[1, 2, 3])
s2 = pd.Series([2, 3, 4], index=[2, 3, 4])

(s1 + s2).to_dict() # {1: NaN, 2: 4, 3: 6, 4:NaN}
```

Getting your data into pandas

Pandas has methods to import data from most interfaces and common data storage formats, including

```
df = pd.read_csv('file.csv', header=0,
    index_col=0, quotechar='"', sep=':',
    na_values = ['na', '-', '.', ''])

df = pd.read_html(url/html_string)
df = pd.read_json(path/JSON_string)
df = pd.read_sql(query, connection)
df = pd.read_excel('filename.xlsx')
df = pd.read_sas(filepath)
df = pd.read_parquet(filepath)
df = pd.read_clipboard() # eg from Excel copy
```

<u>**Note**</u>: see the documentation for the data formats that can be accessed, and for the detailed method arguments.

You can also get data from elsewhere in python

```
df = pd.DataFrame(python_dictionary_of_lists)
df = pd.DataFrame(python_list_of_lists)
df = pd.DataFrame(numpy_2D_matrix)
```

You can place your data as inline text in your python code

```
from io import StringIO

text = """, Animal,     Cuteness, Desirable
A,          dog,        8.7,      True
B,          cat,        9.5,      False
C,          gold fish,  2.3,      False"""

df = pd.read_csv(StringIO(text), header=0,
     index_col=0, sep=',', skipinitialspace=True)
```

Standard statistical data sets are in Seaborn

```
print(sns.get_dataset_names())
iris = sns.load_dataset('iris')
```

You can fake up some random data (for testing)

```
df = pd.DataFrame(np.random.normal(0, 1,(500,4)))
s = pd.Series(np.random.choice(list('abcd'),500))
```

And you can instantiate an empty DataFrame

```
df = pd.DataFrame()
```

Getting your data out of pandas

To a python object

```
python_dictionary = df.to_dict()     # DataFrames
numpy_matrix = df.to_numpy()

python_dictionary = s.to_dict()      # Series
numpy_array = s.to_numpy()
python_list = s.to_list()
```

Also, among many other options …

```
df.to_csv('filename.csv', encoding='utf-8')
df.to_excel('filename.xlsx')

csv = df.to_csv()        # defaults to string
html = df.to_html()      # to string or file
md = df.to_markdown()    # to string or file
json = df.to_json()      # to string or file
df.to_clipboard()        # then paste into Excel
```

<u>**Note**</u>: refer to the pandas documentation for all options.

Working with DataFrames

About the DataFrame

```
df.shape      # tuple - number of (rows, columns)
df.size       # row-count * column-count
df.dtypes     # Series of data types for columns
df.empty      # True if DataFrame is empty

df.info()     # prints info on DataFrame
```

DataFrame memory usage

```
df.memory_usage() # usage in bytes by column
df.memory_usage().sum() / (1024 ** 2) # total MB
df.memory_usage().sum() / (1024 ** 3) # total GB
```

Copy the DataFrame

```
df = df.copy()          # copy a DataFrame
```

Adding, dropping, and renaming rows and columns

Adding columns to a DataFrame

```
df['new column'] = df.sum(axis=1) #add row-totals
```

Adding rows to a DataFrame

```
df = pd.concat([df, additional_rows_as_df])
```

Hint: convert new row(s) to a DataFrame and then use the pandas concat() function. Note: the argument is a list.
Trap: All DataFrames must have the same column labels.

Dropping columns (by label)

```
df = df.drop(columns=col_label)
df = df.drop(columns=[col1, col2]) # multi-drop
```

Dropping rows

```
df = df.drop(row_label)
df = df.drop([row1, row2]) # multi-row drop

# dropping rows with missing data
df = df.dropna(subset='col_name')
df = df.dropna(subset=['c1', 'c2'], how='any')
df = df.dropna(thresh=2) # drop rows with 2+ NaNs

# dropping duplicate rows
df = df.drop_duplicates(keep='first')

# dropping rows with duplicate indexes
df = df[~df.index.duplicated(keep='last')]
```

Rename column labels in a DataFrame

```
df.columns = ['list', 'of', 'col', 'names']
df = df.rename(columns={'old':'new','a':'1'})

df = df.add_prefix('x') # add prefix to col names
df = df.add_suffix('x') # add suffix to col names
```

Change the row labels in a DataFrame

```
df.index = idx                # new ad hoc index
df.index = range(len(df))     # set with iterable

df = df.set_index('A')        # index set to col A
df = df.set_index(['A', 'B'])# multi-level index

df = df.reset_index(drop=True) # with default idx
# if drop=False, the old index stored as a col

df = df.rename(index={'old':'new','a':'1'}) #rows
```

Summarising DataFrames (into Series objects)

By default you get a column-wise summary. But with the axis=1 argument, you get a row-wise summary.

```
s = df.count()        # count non-missing values
s = df.min()
s = df.max()
s = df.sum()
s = df.prod()

s = df.mean()
s = df.median()       # middle value when sorted
s = df.mode()         # most often occurring
s = df.nunique()      # number unique elements
```

Note: missing data (NaN) ignored by default (skipna=True).
Note: these are also called aggregating functions.

Transforming DataFrames (into DataFrames)

Simple arithmetic transformations

```
df = df.T                  # transpose rows & cols

df = df.add(o)    # add df, Series or value
df = df.mul(o)    # mul by df Series val
df = df.div(o)    # div by df, Series, value
df = df.div(s, axis=1) # div columns by a Series
df = df.dot(o)    # matrix dot product
df = df.pow(o)    # raise to the power of other

df = df.round(decimals=0) # rounding floats
df = df.abs()             # absolute values
df = df.cumsum()          # cumulative sum
df = df.cumprod()         # cumulative product
df = df.cummin()          # cumulative minimum
df = df.cummax()          # cumulative maximum
df = df.pct_change(periods=4) # percent change

# rolling functions (example) ...
df = df.rolling(window=4, # rolling sum (etc.)
    min_periods=4, center=True).sum()

# often used with time-series data
df = df.diff(periods=1)   # differenced data
df = df.shift(periods=1)  # time-shifted data
```

Note: for +, -, *, / -, default when the second argument is a Series, is to apply the operation elementwise to each row.

Sorting the DataFrame

```
df = df.sort_values(by=col)
df = df.sort_values(by=[col1, col2])
df = df.sort_index()    # axis=1 to sort by cols
```

Conditional transformations

```
df = df.where(df>0, other=0)

df = df.fillna(0)  # fill missing data with value
df = df.ffill()    # forward fill missing data
df = df.bfill()    # back fill missing data
```

Data-type transformations

```
df = df.astype(int)       # change the data-type
```

Hint: str, 'string', int, float, are common transformations.

Other transformations

```
df = df.isna()            # True if NaN
df = df.notna()           # False if NaN
dfx = df.describe()       # Statistical summary
```

Sub-setting DataFrames

Methods that subset the DataFrame by rows

```
df = df.head(i)          # get first i rows
df = df.tail(i)          # get last i rows
df = df.sample(i)        # random sample of i rows
```

Miscellaneous

Find the first row index label for min/max value

```
index_label = df[col].idxmin()
index_label = df[col].idxmax()
```

Iteration

DataFrame iteration

```
df.items()            # (col_index, series) pairs
df.iterrows()         # (row_index, series) pairs
df.itertuples()       # (row_index, ...) tuple

# for example, iterating over columns …
for label, series in df.items():
    print(f'{label}: first element={series[0]}')
```

Hint: itertuples can be up to 50 times faster than iterrows.
Trap: iteration is rarely the fastest or most efficient way to work with your DataFrame. Look for vectorized solutions.

Series iteration

```
for label, value in s.items():
    print(label, value)
```

Working with DataFrame columns and Series

Important: the sub-setting, summarising, and transforming methods noted above can also be applied to individual DataFrame columns and pandas Series. For example:

```
s = df[col].describe()        # descriptive stats

value = df[col].sum()         # sum the column
value = df[col].max()         # maximum value
value = df[col].count()       # count non-missing
```

In addition, there are some methods and attributes that are typically only used with a Series or a DataFrame column.

```
value = df[col].dtype         # Series data type

df[col] = df[col].astype(float)
df[col] = pd.to_numeric(df[col])
df[col] = pd.to_datetime(df[col])
df[col] = pd.Categorical(df[col])

s = df[col].value_counts()
s = df[col].nsmallest(n)     # n smallest values
s = df[col].nlargest(n)      # n largest values
a = df[col].unique()         # array unique items
s = df[col].rank()

if df[col].all(): pass
if df[col].any(): pass
```

Using the numpy maths functions

```
df[col] = np.log(df[col])
df[col] = np.sqrt(df[col])
```

Note: there are many more numpy math functions; see: numpy.org/doc/stable/reference/routines.math.html

Mapping data

A python dictionary or a pandas Series can be used to map the values in a Series or a DataFrame column.

```
Mapper = pd.Series(['red', 'green', 'blue'],
                 index=['r', 'g', 'b'])
s = pd.Series(['r', 'g', 'r', 'b']).map(mapper)
# s contains: ['red', 'green', 'red', 'blue']

mapper = {'Y': True, 'N': False}
df = pd.DataFrame(np.random.choice(list('YN'),
        500, replace=True), columns=['col'])
df['col'] = df['col'].map(mapper)
```

Note: Indexes can also be mapped if needed.

Working with pandas Indexes

DataFrames have two indexes and Series have one.

```
df.index              # the row index (axis=0)
df.columns            # the col index (axis=1)
s.index               # the series index (axis=0)
```

Pandas indexes accept python index and slice operations

```
label = s.index[0]     # the first index label
label = df.columns[n]  # the nth column label
label = df.index[-2]   # the 2nd last index label

idx = idx[[1,3,5]] # the 2nd, 4th and 6th indexes
idx = idx[1:-1]    # all but the first and last
idx = idx[::-1]    # a reversed index
```

Note: consistent with python, index addresses are integers numbered from 0 to n-1, where n is the length of the index.

Every index is strongly typed

```
df.index.dtype        # data type of row index
df.columns.dtype      # data type of col index
s.index.dtype         # data type of series index
s.index = s.index.astype(dtype) # type conversion
```

Note: if the index is dtype('O'), then it is a mix of datatypes.

Each index has a number of properties

```
b = df.index.is_monotonic_decreasing  # Boolean
b = df.index.is_monotonic_increasing  # Boolean
b = df.index.has_duplicates           # Boolean

i = df.index.nlevels     # number of index levels
```

Note: The Boolean properties are either True or False.

Each index has a number of methods

```
idx = idx.astype(dtype)  # change data type
b = idx.equals(o)        # check for equality
idx = idx.union(o)       # union of two indexes
idx = idx.intersection(o)# index intersection
i = idx.nunique()        # number unique labels
label = idx.min()        # minimum label
label = idx.max()        # maximum label
b = idx.duplicated()     # Boolean for duplicates
l = idx.to_list()        # get as a python list
a = idx.to_numpy()       # get as a numpy array
```

Note: string indexes can also access the .str. accessor methods. Date and time indexes can access the .dt. accessor methods.

Get the integer position of a row or col index label

```
i = df.index.get_loc(row_label)
```

Trap: index.get_loc() returns an integer for a unique match. If not a unique match, may return a slice or a mask.

Using the indexes to access Series and DataFrame data

Select DataFrame columns with []

```
s = df['column_label']        # by label
df = df[[col]]                # by label list
df = df[[col1, col2]]         # by label list
s = df[df.columns[0]]         # by int position
df = df[df.columns[[1, 4]]]   # by int position
df = df[df.columns[[1:4]]]    # by int position
df = df[idx]                  # by a pandas index
df = df[s]                    # by label Series
```

Trap: With [] indexing, a label Series gets/sets columns, but a Boolean Series gets/sets rows

Selecting DataFrame columns with Python attributes

```
s = df.a   # same as s = df['a']
df.existing_column = df.a / df.b
df['new_column'] = df.a / df.b
```

Trap: column names must be valid Python identifiers, but not a DataFrame method or attribute name.
Trap: cannot use attributes to create new columns
Hint: best to avoid using attributes to access data, it is possible to create subtle errors with this feature.

Select DataFrame rows with .loc[]

```
# row selection with row labels
df = df.loc['label']      # by a single row labe
df = df.loc[[row1, row2]] # by list of row labels
df = df.loc['from':'to']  # by slice inclusive-to

# row selection with a Boolean Series
df = df.loc[(df[col1] > 0.5)
            & (df[col2] < 20)]
df = df.loc[df[col].isin([1, 2, 5, 7, 11])]
df = df.loc[~df[col].isin([1, 2, 5, 7, 11])]
# ~ tilde here means not
df = df.loc[df[col].str.contains('string')]

# we can also select rows with a python callable
df = df.loc[lambda frame: frame.index > 300]
```

Avoid: chaining in the form df[col_indexer][row_indexer]
Trap: label slices are inclusive-to, integer slices are exclusive-to
Trap: bitwise "or", "and" and "not" (ie. | & ~) have been co-opted to be Boolean operators on a Series of Boolean. Because of the precedence of these operators, you always need parentheses around comparisons.
Hint: selecting with a callable can be useful in a chained operation.

Select DataFrame rows with .iloc[]

```
# row selection based on integer row number
df = df.iloc[0]         # the first row
df = df.iloc[[1, 3, 5]] # rows 2, 4, 6
df = df.iloc[0:2]       # rows 0 and 1
df = df.iloc[-1:]       # the last row
df = df.iloc[:-1]       # all but the last row
df = df.iloc[::2]       # every 2nd row (0 2 ..)
```

Select rows with query strings

```
# assume our DataFrame has columns 'A' and 'B'
result = df.query('A > B')
result = df.query('B == `col name with spaces`')
result = df.query('A > @python_var')
result = df.query('index >= 20201101')
```

Note: use back ticks around column names with spaces in them. Use @ to access variables in the python environment. The row and column indexes can be accessed with the keywords index and columns.

Select rows with .head(), .tail() or .sample()

```
df.head(n)    # get first n rows
df.tail(n)    # get last n rows
df.sample(n)  # get a random sample of n rows
```

Select individual DataFrame cells with .at[,] or .iat[,]

```
value = df.at[row, col]         # using labels
value = df.iat[irow, icol]      # using integers
```

Select a cross-section with .loc[,] or .iloc[,]

```
# row and col can be scalar, list, slice
xs = df.loc[row, col]           # label accessor
xs = df.iloc[irow, icol]        # integer
accessor

# this cross-section technique can be used to get
an entire column
df_ = df.loc[:, 'A':'C']        # inclusive-to
df_ = df.iloc[:, 0:2]           # exclusive-to

# And it can be used with a callable
df.loc[:, lambda frame: frame.columns == "a"]
```

Note: the indexing attributes (.loc[], .iloc[], .at[] .iat[]) can be used to get and set values in the DataFrame

Filtering for a DataFrame subset (defaults by cols)

```
subset = df.filter(items=['a', 'b']) # by col
subset = df.filter(items=[5], axis=0) #by row
subset = df.filter(like='x') # keep x in col
subset = df.filter(regex='x') # regex in col
```

Conditional replacement

```
df = df.where(conditional, other)
df[col] = df[col].where(conditional, other)
s = s.where(conditional, other)
```

Note: For a DataFrame, the conditional can be either a Boolean DataFrame or a Boolean Series. The default is other=np.nan.

Series element selection with [], .loc[], and .iloc

Series elements are normally selected with []. However, .loc and .iloc methods can be used in a similar manner as above.

Chaining methods

Pandas methods can be chained together using a functional programming style. This can be easier to read and avoids intermediate variables.

Example of method chaining

```
df = (
    sns.load_dataset('titanic')
    .dropna(subset='embark_town')
    .drop(columns=['adult_male', 'alive'])
    .assign(is_adult=lambda df: df['age'] >= 21)
)
```

Hint: long chains are difficult to debug, limit to logical steps. As a rule of thumb, avoid chains longer than 10 methods.
Hint: Bracket chains, with each method call on a new line.

Methods that support chaining

To facilitate chaining, pandas has a few useful methods that take a function as an argument.

```
df.pipe(f) # return the DataFrame from function f
df.assign(c1=f,) # assign series from f to c1 col
df.applymap(f) # apply functn f to df elementwise
df.apply(f) # apply f to the rows/cols of df
df.aggregate(f) apply aggregating f to rows/cols
df.transform(f) apply transforming f to rows/cols
```

Note: in all the above code examples, f is a python function. It can be either an anonymous lambda function, or a normal python function. The function that pipe and assign calls expects a DataFrame as the first argument; the functions for apply, aggregate and transform expect a Series.

Strings

Type conversion

```
df = pd.DataFrame([1, 2, 3], columns=['A'])
df['B'] = df['A'].copy()

df['A'] = df['A'].astype(str)
df['B'] = df['B'].astype('string')

print(df.dtypes)           # A: object; B: string
```

Hint: prefer 'string' to str. Pandas uses its generic object type for the python str type.

Vectorized string functions (including regular expression (regex) manipulations)

```
s = pd.Series(['first', '生日快乐', ' thirst '])
i = s.str.len() # size of strings in characters

s = s.str.lower()
s = s.str.upper()
s = s.str.title()
s = s.str.capitalize()
s = s.str.casefold()
s = s.str.swapcase()

s = s.str.split(pat, n, expand)
s = s.str.join()
s = s.str.cat() # concatenate

s = s.str.lstrip() # remove left-hand white space
s = s.str.rstrip() # remove right white space
s = s.str.strip()  # remove white space

s = s.str.replace('x','y', case=True, regex=True)
i = s.str.count(pat) # count matches
i = s.str.find(sub_string) # left find
i = s.str.rfind(substring) # right find
x = s.str.findall(pat)
b = s.str.fullmatch(pat, case=True) #pat is regex

b = s.str.contains(pat, case=True, regex=True)
b = s.str.startswith(pat)
b = s.str.endswith(pat)

s = s.str.removeprefix(prefix)
s = s.str.removesuffix(suffix)
s = s.str.repeat(repeats)

s = pd.Series(['a', ' ', '\t', 'h2', '5', 'H',])
b = s.str.isalnum()     # is alpha-numeric
b = s.str.isalpha()
b = s.str.isdigit()
```

Vectorized string functions (continued)

```
b = s.str.isspace()
b = s.str.islower()
b = s.str.isupper()
b = s.str.istitle()
b = s.str.isnumeric()
b = s.str.isdecimal()

df = s.str.get_dummies() # create dummy variables
```

Note: More string methods in the pandas documentation.

Categorical data

Categorical data

The pandas Series has a factors-like (from the R language) data type for encoding categorical data. This can save a lot of memory space when used appropriately with large data sets.

```
df = pd.DataFrame(list('abcdefuanduanducdadcaa'),
                  columns=['Group'],
                  dtype='category')

df['Cat'] = df['Group'].astype('category')
```

Note: the key here is to specify the "category" data type.
Note: categories will be ordered on creation if they are sortable. This can be turned off. See ordering below.

Get the category codes used by pandas

```
s = pd.Series(list('abacba'), dtype='category')
print(s.cat.codes)
print({a: b for a, b in zip(s, s.cat.codes)})
```

Convert categories back to their original data type

```
s = pd.Series(list('abacba'), dtype='category')
s = s.astype('string') # or str
print(s.dtype)
```

Ordering, reordering and sorting

```
s = pd.Series(list('abacba'), dtype='category')
s = s.cat.as_ordered()   # make cats ordered
print(s.cat.categories)  # list of the categories
print(s.cat.ordered)     # True if ordered

# re-order the categories
s = s.cat.reorder_categories(['b', 'c', 'a'])

# sort the categories
s = s.sort_values()

s = s.cat.as_unordered() # make cats unordered
print(s.cat.ordered)
```

Renaming categories

```
s = pd.Series(list('abacba'), dtype='category')
s = s.cat.rename_categories(['dog','cat','bee'])
print(s)

# using a comprehension and direct assignment
s.cat.categories = ['Group ' + str(i)
                    for i in s.cat.categories]
```

Trap: categories must be uniquely named
Hint: can also use a python dictionary to rename categories
Adding and removing new categories

```
s = s.cat.add_categories([7, 8, 9])
s = s.cat.remove_categories([7, 9])
s.cat.remove_unused_categories()       # inplace
```

Dates, times and time-series data

The key pandas objects for managing dates and times are:

Concept	Data	Index
Point	Timestamp	DatetimeIndex
Span	Period	PeriodIndex
Delta	Timedelta	TimedeltaIndex
Offset	DateOffset	

Timestamps

Timestamps represent a point in time.

```
now = pd.Timestamp('now') # naive local date-time
gmt_now = pd.Timestamp('now', tz='GMT')

t = pd.Timestamp('2022-01-01')
t = pd.Timestamp('2022-01-01 21:15:06')
t = pd.Timestamp(year=2022, month=6,
      day=20, hour=21, minute=15, second=6,
      microsecond=250, nanosecond=0,
      tz='Australia/Canberra')
      # handles daylight savings time
```

Note: Timestamps can be either naive or time-zone aware.

Timestamps from python time

```
from datetime import datetime
d = datetime(year=2021, month=1, day=5)
t = pd.Timestamp(d)
```

Minimum and maximum dates for Timestamps

```
pd.Timestamp.max  # 2262-04-11
pd.Timestamp.min  # 1677-09-21
pd.Timestamp(None)# pd.NaT - like a NaN for times
```

A pandas Series of Timestamps

```
# from a Series of time-strings
s = pd.Series(['1900-01-01', None, 'now',
               '2020-01-01 01:02:03.14159265'])
sot = pd.to_datetime(s, utc=True)

# from day-first dates in a Series
s = pd.Series(['01-01-1963', '01-01-1993'])
sot = pd.to_datetime(s, dayfirst=True)

# from Unix epoch time (seconds since 01-01-1970)
s = pd.Series([1659720105, 1660065705])
sot = pd.to_datetime(s, unit='s', utc=True)

# from python times in a Series
from datetime import datetime
d = datetime(year=2000, month=1, day=1)
s = pd.Series([datetime(year=2000, month=1,
                        day=1)])
sot = pd.to_datetime(s)

# get a range of dates in a Series
sot = pd.date_range('2021-01-01', periods=500,
     freq='D').to_series().reset_index(drop=True)

# Non-standard time-strings in a Series, example
s = pd.Series(['09:08:55.7654-JAN092002',
               '15:42:02.6589-FEB082016'])
sot = pd.to_datetime(t,
                 format="%H:%M:%S.%f-%b%d%Y")
```

Note: if one passes a pandas Series to the pd.to_datetime() helper function it returns a pandas Series of Timestamps. However, if one passes the function a list of date strings, it returns a pandas DatetimeIndex object.

Note: we can only use 'now' as a time-string with the utc=True argument to the pd.to_datetime() function. (This practice differd from the pd.Timestamp() function).
Note: a Series of Timestamps must be either all naive or all time zone aware.
Also: %B = full month name; %m = numeric month; %y = year without century; and more …

The vectorised .dt. accessor methods

```
sot = pd.Series([pd.Timestamp('now')])

date = sot.dt.date               # the date only
time = sot.dt.time               # the times only

year = sot.dt.year
month = sot.dt.month
day = sot.dt.day
hour = sot.dt.hour
minute = sot.dt.minute
second = sot.dt.second

woy = sot.dt.isocalendar().week # of year
doy = sot.dt.dayofyear
quarter = sot.dt.quarter
dow = sot.dt.dayofweek          # 0 = Monday
dim = sot.dt.daysinmonth

b = sot.dt.is_leap_year
b = sot.dt.is_year_start  # Also: month, quarter
b = sot.dt.is_year_end    # Also: month, quarter
i = sot.dt.daysinmonth

text = sot.dt.strftime('%Y-%m-%d') # Unix format
a = sot.dt.to_pydatetime() # convert to python
text = sot.dt.day_name(locale='de_DE.UTF-8')
text = sot.dt.month_name(locale='de_DE.UTF-8')
```

Note: this is not a complete list of .dt. accessor methods.

Working with time-zones

```
sot = (
    pd.to_datetime(pd.Series(['now']), utc=True)
    .dt.tz_localize(None) # remove tz awareness
    .dt.tz_localize('GMT') # provide tz awareness
    .dt.tz_convert('Australia/Canberra')
)
```

Note: use .dt.tz_localize(time_zone_string) to give a Series of Timestamps time-zone awareness.

DatetimeIndex

If you make a Series of Timestamps the DataFrame or Seroes index, you get a DatetimeIndex. Also, it can be constructed with the pandas pd.to_datetime() helper function. **Note**: all of the .dt. accessor functions and attributes are available directly to a DatetimeIndex object, for example:

```
dti = pd.to_datetime(['2022-06-13 10:15:29'])

dates = dti.date          # array of datetime.date
times = dti.time          # array of datetime.time
day_names = dti.day_name() #pd.Index of day names
```

Populate a DatetimeIndex with dates

```
dti = pd.date_range('2015-01', periods=12,
                    freq='M') # monthly (at end)
dti = pd.date_range('2019-01-01', periods=31,
                    freq='D') # daily
```

Note: many possible options here.

Period and Period Index
Prefer a PeriodIndex (over a DatetimeIndex) for time-series data, particularly for data that is about the number of events in, or the total value of sales over, a period.

Periods and PeriodIndexes are constructed from dates.

```
# periods represent time spans
p = pd.Period('2021', freq='Y')        # year
p = pd.Period('2021-Q1', freq='Q')     # quarter
p = pd.Period('2022-06-13', freq='Q')  # quarter
p = pd.Period('2021-01', freq='M')     # month
p = pd.Period('2021-01-01', freq='D')  # day

# periods are the basis for the PeriodIndex
dates = ['2022-01-01','2022-04-01','2022-07-01']
sot = pd.to_datetime(pd.Series(dates))
ps = sot.dt.to_period()       # Series of Periods

pi = pd.PeriodIndex(dates, freq='Q')
pi = pd.PeriodIndex(sot, freq='M')
pi =pd.period_range('2022-01',periods=3,freq='M')
```

Period frequency constants (not a complete list)

Name	Description
U	**Microsecond**
L	Millisecond
S	Second
T	Minute
H	Hour
D	Calendar day
B	Business day
W-{MON, TUE, …}	Week ending on …
MS	Calendar start of month
M	Calendar end of month
Q-{JAN, FEB, …}	Quarter end with year ending (Q – December)
A-{JAN, FEB, …}	Year end (A - December)

Using .dt. accessors to access data

```
# start with some play data (with a PeriodIndex)
N = 48 # rows of fake data to be created
df = pd.DataFrame(np.random.randint(low=0,
    high=999, size=(N, 5)),
    index=pd.period_range('2015-01',
        periods=n, freq='M'))

# examples of using the .dt. accessor methods
february_selector = (df.index.month == 2)
february_data = df[february_selector]

q1_data = df[(df.index.month >= 1) &
   (df.index.month <= 3)]

may_or_nov_data = df[(df.index.month == 5) |
   (df.index.month == 11)]

year_totals = df.groupby(df.index.year).sum()
```

Conversions

```
ts = ps.dt.to_timestamp()   # Period to Timestamp
ps = ts.dt.to_period(freq='D')#Timestamp toPeriod
pi = pd.PeriodIndex(ps)     # Period to PeriodIdx
dti = pi.to_timestamp()     # PeriodIdx to Tstamp
pi = dti.to_period(freq='D')# DatetimeIndex to PI
```

Timedeltas
When we subtract a Timestamp from another Timestamp, we get a Timedelta object in pandas. We can also instantiate Timedelta objects and add them to Timestamps.

```
today = pd.Timestamp('now')
day = pd.Timedelta(days=1)
tomorrow = today + day
yesterday = today - day

ts1 = pd.Series(pd.date_range(start='2022-01-01',
                    periods=3, freq='D'))
ts2 = pd.Series(pd.date_range(start='2022-01-01',
                   periods=3, freq='MS'))
td = ts2 - ts1 # [0 days, 30 days, 57 days]

# converting Timedelta objects to numeric values
day = pd.Timedelta(days=1)
days = td / day # yields: [0, 30, 57]

hour = pd.Timedelta(hours=1)
hours = td / hour # yields: [0, 720, 1368]
```

Offsets
Subtracting a Period from a Period gives an Offset.

```
offset = pd.DateOffset(days=4)

s = pd.Series(pd.period_range('2022-01-01',
    periods=365, freq='D'))
offset2 = s[4] - s[0]

s = s.diff(1) # s is now a series of offsets
```

Converting an Offset to a numeric value

```
x = offset.n # for an individual offset

# for a series of offsets
t = s.apply(lambda z: np.nan if z is pd.NaT
                        else z.n)
```

Upsampling

```
# fake up some quarterly count data
pi = pd.period_range('1960Q1',
       periods=220, freq='Q')
df = pd.DataFrame(np.random.randint(low=0,
    high=999, size=(len(pi), 5)), index=pi)

# which we can upsample to monthly count data
dfm = df.resample('M').asfreq() # with NAs!

dfm2 = (df.resample('M').asfreq().fillna(0)
    .rolling(window=3, min_periods=3).mean()
    .bfill(limit=2)) # assuming no NA data
```

Note: df.resample(arguments).aggregating_function(). There are lots of options here. See the manual.

Downsampling

```
# downsample from monthly to quarterly counts
dfq = dfm.resample('Q').sum()
```

Note: df.resample(arguments).aggregating_function().

Joining / Combining DataFrames

There are three ways to join DataFrames:

- **concat** – concatenate (or stack) two DataFrames side by side (horizontally), or one on top of the other (vertically).
- **merge** – using a database-like (or an SQL-like) join operation on side-by-side DataFrames.
- **combine_first** – splice two DataFrames together, choosing values from one DataFrames over the other.

Simple concatenation/stacking is often all you need.

```
# Fake-up some data …
import string

def get_random_frame():
    RC = 40, 5 # rows, columns
    names = list(string.ascii_uppercase)[:RC[1]]
    return pd.DataFrame(np.random.randint(low=0,
           high=999, size=RC), columns=names)

df1 = get_random_frame()
df2 = get_random_frame()
df3 = get_random_frame()

# Concatenate or stack …
dfv = pd.concat([df1, df2, df3])     # top/bottom
dfh = pd.concat([df1, df2], axis=1)  # left/right
```

Be careful: it is easy to create duplicate row or column labels (as we did in this example). pd.concat() has arguments for minimising this problem. You can also reindex/rename before/after concatenation.
Be careful: Also, be careful that the column labels or row labels match in the direction you are stacking. Again, pd.concat() has some arguments for managing this.
Note: if no axis is specified, defaults to top/bottom stacking

Merge (analogous to SQL joins)

```
df_new = pd.merge(left=df1, right=df2,
         how='outer', left_index=True,
         right_index=True)          # on indexes

df1['Common'] = range(0, len(df1))
df2['Common'] = range(1, len(df2)+1)
df_new = pd.merge(left=df1, right=df2,
         how='left', left_on='Common',
         right_on='Common')         # on columns
```

How: 'left', 'right', 'outer', 'inner' (where outer=union/all; inner=intersection)
Note: merge is both a pandas helper-function, and a DataFrame method
Trap: many-to-many merges can result in an explosion of associated data.

Combine first (priority splicing)

```
# let's create a DataFrame with missing items
df4 = df2.copy()
N = 29 # holes in DataFrame to create
for r, c in {(np.random.randint(0, len(df4)),
         np.random.randint(0, len(df4.columns)))
            for _ in range(N)}:
    df4.iloc[r, c] = np.nan

# do the combine first and then compare
df = df4.combine_first(other=df1)

print(df == df2) #see where the holes were filled
```

Note: combine_first() uses the non-null values from the method calling DataFrame. Null values in the calling DataFrame are filled with values from the same location in the other DataFrame. The index of the combined DataFrame will be the union of the indexes from the calling and other DataFrames.

Pivot tables

Pivot tables are powerful tools for analysing and summarising data.

The iris dataset has the measurements of petal and sepal lengths and widths for 150 collected flowers from three closely related species of iris. We can use a pivot table to see the average lengths for each species.

```
iris = sns.load_dataset('iris')
display(pd.pivot_table(iris,
                       index='species',
                       aggfunc='mean'))
```

Species	Petal length	Petal width	Sepal length	Sepal width
setosa	1.462	0.246	5.006	3.428
versicolor	4.260	1.326	5.936	2.770
virginica	5.552	2.026	6.588	2.974

The titanic dataset is a subset of 891 passengers and crew on the titanic when it sank on 15 April 1912; (there were more than 2,200 on board when the ship sank). Let's look at average age and survival rates by gender and passenger class in this dataset. We can also look at the total number who survived, and the size of each group.

```
titanic = sns.load_dataset('titanic')
pt = pd.pivot_table(titanic,
                    index=['sex', 'pclass'],
                    values=['survived', 'age'],
                    aggfunc=['mean', 'sum', 'size'])
```

		mean		sum		size
		age	survived	age	survived	0
sex	pclass					
female	1	34.61	0.968	2942	91	94
	2	28.72	0.921	2126	70	76
	3	21.75	0.500	2219	72	144
male	1	41.28	0.369	4169	45	122
	2	30.74	0.157	3043	17	108
	3	26.51	0.135	6706	47	347

Note: this example created hierarchical row and column indexes.

Grouping Data – .groupby() – split-apply-combine

The .groupby() method can be used like a pivot_table to reshape and aggregate or transform data.

For example, we can **split** the iris data by species and **apply** the mean function to each group and then have those means **combined** back into a DataFrame. This code gives much the same result as the above pivot_table.

```
iris = sns.load_dataset('iris')
iris.groupby('species').mean()
```

We can also do the same thing with the titanic dataset as we did above. In this code snippet we group on two columns (sex and passenger class), we then select the two variables we want to summarise, and finally we apply the three aggregating functions.

```
titanic = sns.load_dataset('titanic')
display(
    titanic.groupby(['sex', 'pclass'])
    [['survived', 'age']]
    .aggregate(['mean', 'sum', 'size'])
)
```

Note: in this example we have index-selected the two columns we are interested in by using a list. If we were only interested in one column, we would have used a string.

First you group your data

```
gb = df.groupby(col)           # by one columns
gb = df.groupby([col1, col2]) # by 2 cols
gb = df.groupby(level=0)       # row index groupby
gb = df.groupby(level=['a','b']) # mult-idx gb
print(gb.groups)
```

Note: groupby() returns a pandas groupby object
Note: the groupby object attribute .groups contains a dictionary mapping of the groups.
Trap: NaN values in the group key are automatically dropped – there will never be a NA group.

Then after grouping, you aggregate, transform or filter

Applying an aggregating function

```
# apply to a single column ...
s = gb[col].sum()
s = gb[col].agg(np.sum)

# apply to every column in DataFrame ...
s = gb.count()
df_summary = gb.describe()
df_row_1s = gb.first()
```

Note: .agg() and .aggregate() are the same function
Note: aggregating functions include mean, sum, size, count, std, var, sem (standard error of the mean), describe, first, last, min, max

Applying multiple aggregating functions

```
# apply multiple functions to one column
dfx = gb['col2'].agg([np.sum, np.mean])

# apply to multiple fns to multiple cols
dfy = gb.agg({
    # col:  functions to apply,
    'cat': np.count_nonzero,
    'col1': [np.sum, np.mean, np.std],
    'col2': [np.min, np.max]
})
```

Note: gb['col2'] above is shorthand for df.groupby('cat')['col2'], without the need for regrouping.

Applying transform functions

```
# transform to group z-scores, which have
# a group mean of 0, and a std dev of 1.
zscore = lambda x: (x-x.mean())/x.std()
dfz = gb.transform(zscore)

# replace missing data with the group mean
mean_r = lambda x: x.fillna(x.mean())
df = gb.transform(mean_r) # entire DataFrame
df[col] = gb[col].transform(mean_r) # one col
```

Note: can apply multiple transforming functions in a manner like multiple aggregating functions above,

Applying filtering functions
Filtering functions allow you to make selections based on whether each group meets specified criteria

```
# select groups with more than 10 members
eleven = lambda x: (len(x['col1']) > 10)
df11 = gb.filter(eleven)
```

Hierarchical indexes

Hierarchical indexing allows us to index our data with multiple labels (at multiple levels). Pivot tables and grouping data often creates a hierarchical index in the resulting DataFrame. For example using the titanic data:

```
df = (
    sns.load_dataset('titanic')
    .groupby(['sex', 'pclass'])
    [['survived', 'age']]
    .aggregate(
        {'age': ['mean', 'min', 'max'],
         'survived': ['sum', 'size', 'mean'],}
    )
)
to_round = [('age','mean'), ('survived','mean')]
df[to_round] = df[to_round].round(2)
```

Note: we used tuples for indexes in the above code snippet.
Note: we have a two-level column index and a two-level row index in our data table. The outer/upper most index is at level 0, the next index is at level 1, and so on.

		age			survived		
		mean	min	max	sum	size	mean
sex	pclass						
female	1	34.61	2.00	63.0	91	94	0.97
	2	28.72	2.00	57.0	70	76	0.92
	3	21.75	0.75	63.0	72	144	0.50
male	1	41.28	0.92	80.0	45	122	0.37
	2	30.74	0.67	70.0	17	108	0.16
	3	26.51	0.42	74.0	47	347	0.14

Indexing (using above DataFrame as an example)

```
# use tuples to index a specific row or column
s = df[('survived', 'mean')] # survival rates col

dfs = df['survived'] # all the survived columns

# get the two mean columns
dfs1 = df.loc[:,
          df.columns.get_level_values(1)=='mean']

# get the two first class passenger rows
dfs2 = df.loc[df.index.get_level_values(1)==1]

# get the female passenger rows
dfs3 = df.loc['female']
```

Stack, unstack and melt

The stack method turns column names into index values, and the unstack method turns index values into column names. The melt method reshapes a DataFrame from a wide format to a longer format.

```
df = df.stack(level=0)
df = df.unstack(level=0)
df = df.melt()
```

Note: negative levels select the inner most levels.

Note: See documentation, there are many arguments to these methods.
Trap: stack and unstack will return a Series if it is called in respect of DataFrame with a single index on the relevant axis.

Swapping index levels

```
dfx = df.swaplevel(i=0, j=1, axis=0)
```

Note: negative levels select the inner most levels.

Plotting

Before we start

Import matplotlib.pyplot and choose a plot style

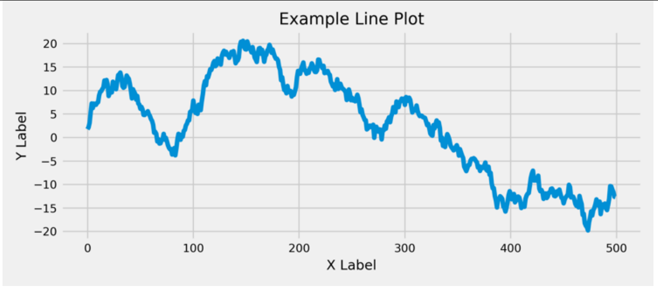
```
import matplotlib.pyplot as plt
print(plt.style.available)
plt.style.use('fivethirtyeight')
```

Let's code a simple plotting utility function (so we don't need to repeat this code with every plotting example).
I often use a small function like this for finalising plots.

```
def and_plot(ax, figsize=(8, 3.5), pad=1, **kw):
    """Plot finalising function, with keyword
       arguments for title, xlabel and ylabel."""

    def get_arg(text):
        return kw[text] if text in kw else None

    ax.set_title(get_arg('title'))
    ax.set_xlabel(get_arg('xlabel'))
    ax.set_ylabel(get_arg('ylabel'))

    fig = ax.figure
    fig.set_size_inches(*figsize)
    fig.tight_layout(pad=pad)

    fn = get_arg('filename')
    fn = fn if fn is not None else 'filename'
    fig.savefig(f'{fn}.png', dpi=300)

    plt.show()
    plt.close()
```

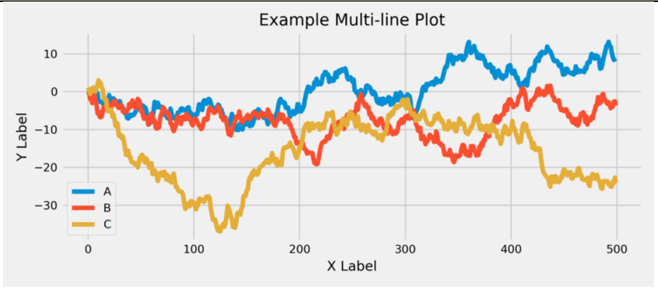
Single line plot

```
# generate some random data
np.random.seed(0)
series = (
    pd.Series(np.random.normal(0, 1, 500))
    .cumsum()
)

# plot that data
ax = series.plot()
and_plot(ax,
         title='Example Line Plot',
         xlabel='X Label',
         ylabel='Y Label', )
```



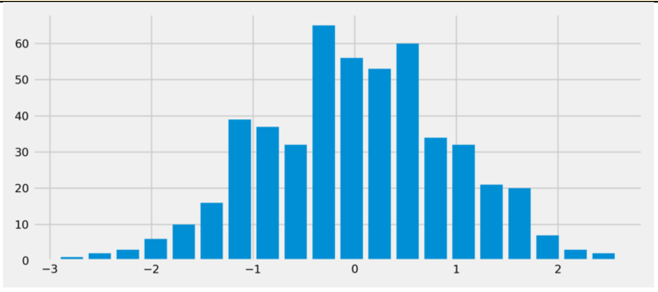
Multiple line plot

```
df = (
    pd.DataFrame(np.random.normal(0, 1, (500,3)),
                 columns=list('ABC'))
    .cumsum()
)
ax = df.plot()
and_plot(ax, title='Example Multi-line Plot',
         xlabel='X Label', ylabel='Y Label')
```



Single histogram

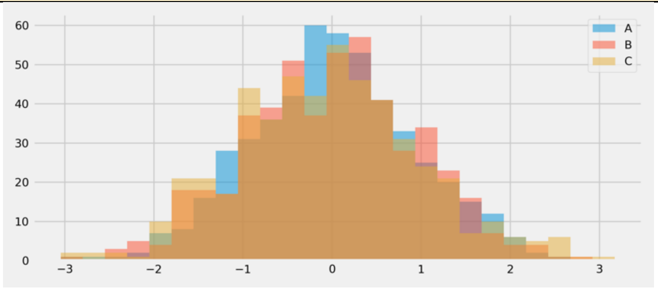
```
# Using the data from the above example
ax = df['A'].diff().plot.hist(bins=20,rwidth=0.8)
and_plot(ax)
```



Note: differencing the data undoes the cumulative sum from when we faked up this data.

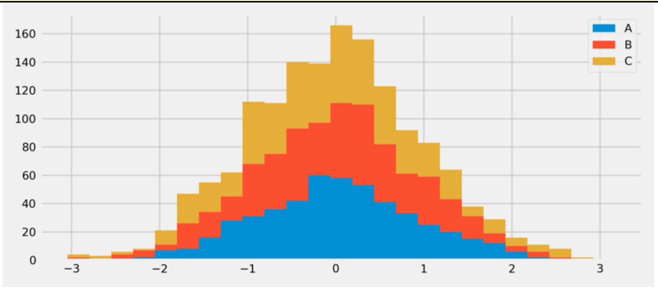
Multiple overlapping histograms

```
ax = df.diff().plot.hist(bins=25, alpha=0.5)
and_plot(ax)
```



Multiple stacked histograms

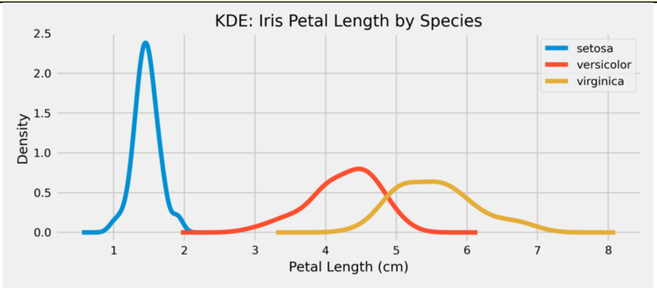
```
ax = df.diff().plot.hist(bins=25, stacked=True)
and_plot(ax)
```



Kernel Density Estimate

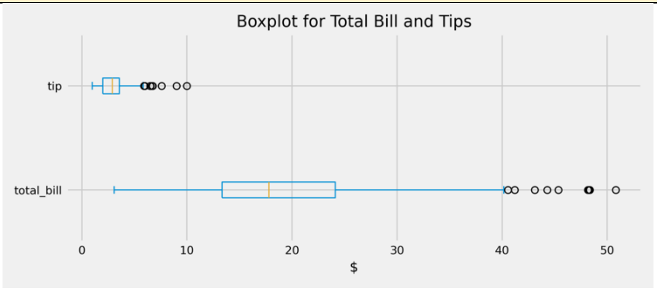
In this example we group the iris data by species, and then plot a kernel density estimate for each species.

```
iris = sns.load_dataset('iris')
ax = None
for label, df in iris.groupby('species'):
    ax = df['petal_length'].plot.kde(label=label,
                                     ax=ax)
ax.legend(loc='best')
and_plot(ax,
       title='KDE: Iris Petal Length by Species',
         xlabel='Petal Length (cm)',
         ylabel='Density', )
```



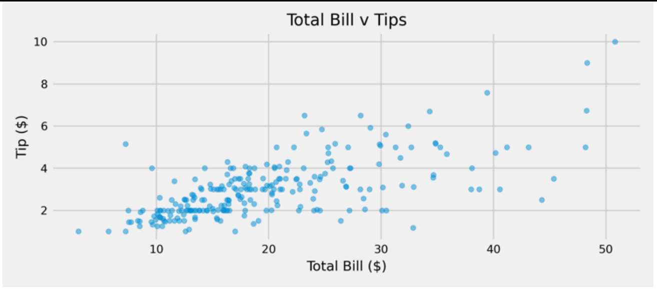
Box plot

```
tips = sns.load_dataset('tips')
ax = tips[['total_bill',
           'tip']].plot.box(vert=False)
and_plot(ax,
         title='Boxplot for Total Bill and Tips',
         xlabel='$')
```



Scatter plot

```
tips = sns.load_dataset('tips')
ax = tips.plot.scatter(x='total_bill', y='tip',
                                     alpha=0.5)
and_plot(ax, title='Total Bill v Tips',
         xlabel='Total Bill ($)',
         ylabel='Tip ($)')
```

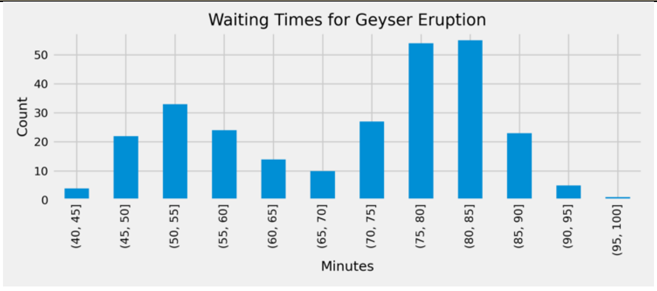


Bar charts

```
# get some data
geyser = sns.load_dataset('geyser')
binned_waiting = (
    pd.cut(geyser['waiting'],
           bins=range(40,101,5))
    .value_counts()-
    .sort_index()
)
```

Bar chart (continued)

```
# plot the bars for our binned data
ax = binned_waiting.plot.bar()
and_plot(ax,
       title='Waiting Times for Geyser Eruption',
       xlabel='Minutes', ylabel='Count', )
```



Basic Statistics

Summary descriptive statistics

```
s = df[col].describe()
df1 = df.describe()
```

Key statistical methods on a DataFrame

```
s = df.value_counts()   # value counts
s = df.mean()           # also median() and mode()

s = df.sem()            # standard error of mean
s = df.std()            # standard deviation
s = df.var()            # variance
s = df.kurtosis()       # kurtosis
s = df.skew()           # skew
s = df.quantile(q=0.5) # get value at quantile(s)
s = df.mad()            # mean absolute deviation
```

Note: these summarising methods can also be used on a DataFrame column or a Series.

Correlation and covariance

```
dfx = df.corr() # pairwise correlation of columns
dfx = df.cov()  # pairwise covariance of columns
```

Cross-tabulation (frequency counts)

```
ct = pd.crosstab(index=df['a'],
        cols=df['b'])
```

Quantiles and ranking

```
quants = [0.025, 0.25, 0.5, 0.75, 0.975]
q = df.quantile(quants)
r = df.rank()
```

Binning data for histograms

```
count, bins = np.histogram(df[col])
count, bins = np.histogram(df[col], bins=5)
count, bins = np.histogram(df[col],
              bins=[-3,-2,-1,0,1,2,3,4])
```

Simple (OLS) linear regression

```
import statsmodels.formula.api as sm
tips = sns.load_dataset('tips')
result = sm.ols(
formula="tip ~ total_bill + sex + smoker + size",
             data=tips).fit()

print (result.params)
print (result.summary())
```

Simple smoothing example using a rolling apply

```
weights = np.array([1,2,3,3,3,2,1]) / 15.0

s = df['A'].rolling(window=len(weights),
    min_periods=len(weights),
    center=True).apply(
         func=lambda x: (x * weights).sum())

# fix the missing end data ... unsmoothed
s = df['A'].where(s.isna(), other=s)
```

Using pandas with a Jupyter Notebook

Truncated tables

Some of the display constraints in a Jupyter Notebook can be annoying. You might want to make these adjustments.

```
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
pd.set_option('display.width', 1000)
pd.set_option('display.max_colwidth', 100)
```

The watermark library

Install the watermark library to get a report on the version numbers for python and the imported packages you have.

```
%load_ext watermark
%watermark -u -n -t -v -iv -w
```

Endnotes

This cheat sheet was cobbled together by tireless bots roaming the dark recesses of the Internet seeking ursine and anguine myths from a fabled land where pandas and pythons gambol together. There is no guarantee the narratives were captured and transcribed accurately. You use these notes at your own risk. I will not be held responsible for whatever happens to you and those you love once you begin to apply what is written here. You have been warned. Caution is advised.

Errors: If you find any errors, please email me at mark.the.graph@gmail.com; (but please do not correct my use of Australian-English spelling conventions).