# Cheat Sheet: The pandas DataFrame

## Preliminaries

This cheat sheet uses the Arial and Consolas fonts. If not already installed, you should download these fonts.

**Start by importing these Python modules**

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

**Check the versions of python/pandas you are using**

```
import platform
print(platform.python_version())
print(pd.__version__)
```

This cheat sheet was written for pandas version 1.4.
It assumes you are using Python 3.9 or higher.

**Limitations**
Pandas is not designed for "big data" (data larger than (say) about 10 gigabytes). All of pandas' calculations are done within RAM memory. It is very efficient with small data (measured in kilobytes). And it is robust and scales well with larger data (measured in megabytes, and even single digit gigabytes if your machine has sufficient RAM). Aim to have 5 to 10 times the RAM memory as the size of the data you are working with.

**Check the RAM available on your machine**

```
import psutil
total = (round(psutil.virtual_memory().total
         / (1024 ** 3), 1))
print(f"{total} GB of RAM available")

# 64.0 GB of RAM available
```

## The conceptual model

Pandas provides two important data types: the array-like **Series** and the tabular **DataFrame**.

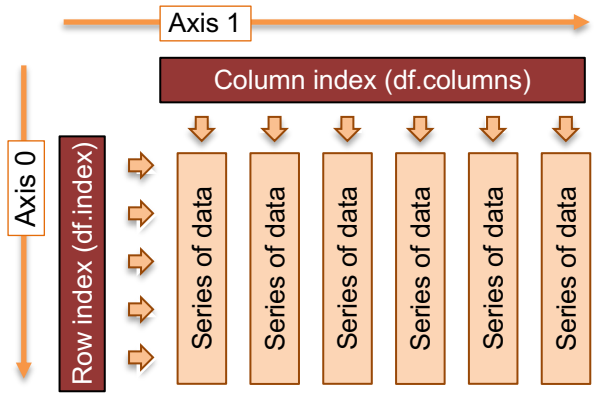
A **Series** is an ordered, one-dimensional array of data with an index. All the data in a Series is of the same pandas data type. The index is also strongly typed. Series arithmetic is vectorized after first aligning the Series index for each of the operands.

**Examples of Series Arithmetic**

```
s1 = pd.Series(range(0, 4))       # 0, 1, 2, 3
s2 = pd.Series(range(1, 5))       # 1, 2, 3, 4
s3 = s1 + s2                      # 1, 3, 5, 7

s4 = pd.Series([1, 2, 3], index=[0, 1, 2])
s5 = pd.Series([1, 2, 3], index=[2, 1, 0])
s6 = s4 + s5                      # 4, 4, 4

s7 = pd.Series([1, 2, 3], index=[1, 2, 3])
s8 = pd.Series([1, 2, 3], index=[0, 1, 2])
s9 = s7 + s8                      # NAN, 3, 5, NAN
```

**Note**: NaNs indicate missing data in pandas.

A **DataFrame** is a two-dimensional table of data with column and row indexes (something like a spread sheet). The columns are made up of Series objects.




**A DataFrame has two Indexes:**

- Typically, the row index (df.index) might be:
  - Integers – for case or row numbers;
  - Strings – for case names; or
  - DatetimeIndex or PeriodIndex – for time series
- Typically, the column index (df.columns) is a list of strings (variable names) or (less commonly) integers

Both the DataFrame and the Series have indexes, which are used to access data, to sort data, and to select data.

## Get your data into a DataFrame

**Getting data from file (many options)**

```
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_hdf(filepath)
df = pd.read_sas(filepath)
df = pd.read_parquet(filepath)
df = pd.read_clipboard() # eg from Excel copy
```

**Note**: pandas also reads compressed files (e.g., zip, gzip); however, you may need to specify a compression argument in the method call.

**Getting 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)
```

**Put your data inline within you python code**

```
from io import StringIO
data = """, Animal, Cuteness, Desirable
A,          dog,    8.7,      True
B,          cat,    9.5,      False"""
df = pd.read_csv(StringIO(data), header=0,
    index_col=0, skipinitialspace=True)
```

**Create an empty DataFrame**

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

### Standard statistical datasets from Seaborn

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

### Fake up some random data – useful for testing

```
df = (pd.DataFrame(np.random.rand(500, 4),
      columns=list('ABCD')) - 0.5).cumsum()
df['Group'] = [np.random.choice(list('abcd'))
      for _ in range(len(df))]
df['Date'] = pd.date_range('1/1/2020',
      periods=len(df), freq='D')
```

## Getting your data out of a DataFrame

### From pandas to a Python object

```
python_dictionary = df.to_dict()
numpy_2d_matrix = df.to_numpy()
```

### 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 (say) paste into Excel
```

**Note**: See the pandas documentation for arguments.

## The whole DataFrame

### About the DataFrame

```
df.shape      # tuple - number of (rows, columns)
df.size       # row-count * column-count
df.dtypes     # Series of data types of columns
df.empty      # True if DataFrame is empty
df.info()     # prints info on DataFrame
df.describe() # summary stats on columns
df.index      # the row index of a DataFrame
df.columns    # the column index of a 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
```

### DataFrame utilities

```
df = df.T               # transpose rows and cols
df = df.copy()          # copy a DataFrame
df = df.sort_values(by=col)
df = df.sort_values(by=[col1, col2])
df = df.sort_values(by=row, axis=1)
df = df.sort_index()    # axis=1 to sort cols
df = df.astype(dtype)   # type conversion
```

### DataFrame iteration methods

```
df.iteritems()  # (col_index, series) pairs
df.iterrows()   # (row_index, series) pairs
df.itertuples() # (row_index, ...) tuple
# example ... iterating over columns ...
for (column, series) in df.iteritems():
    print(f'{column}: {series[0]}')
```

**Note**: iteration is rarely the best way to work with your DataFrame. Look for vectorized solutions.

### Maths methods that return a DataFrame

```
df = df.abs()     # absolute values
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.dot(o)    # matrix dot product

df = df.cummax()  # (cols default axis)
df = df.cummin()  # (cols default axis)
df = df.cumsum()  # (cols default axis)
df = df.diff()    # 1st diff (cols default axis)
```

### Summarising maths methods that return a Series

```
s = df.max()      # max of axis (col def)
s = df.mean()     # mean (col default axis)
s = df.median()   # median (col default)
s = df.min()      # min of axis (col def)
s = df.sum()      # sum axis (cols default)
s = df.count()    # non NA/null values
```

## Working with indexes

### A DataFrame has two indexes (for rows/columns)

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

### A Series has one index

```
s.index               # the series index (axis=0)
```

### Each 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
```

### Each index has a number of properties

```
b = df.index.is_monotonic_decreasing
b = df.index.is_monotonic_increasing
b = df.index.has_duplicates
i = df.index.nlevels      # number of index levels
```

### 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
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
```

### Change the column labels in a DataFrame

```
df = df.rename(columns={'old':'new','a':'1'})
df.columns = ['new1', 'new2', 'new3'] # etc.
```

### Change the row labels in a DataFrame / Series

```
df.index = idx                # new ad hoc index
df = df.set_index('A')        # index set to col A
df = df.set_index(['A', 'B']) # multi-level index
df = df.reset_index()         # replace old w new
# by default, the old index stored as a col in df
df.index = range(len(df))     # set with list
df = df.reindex(index=range(len(df)))
df = df.set_index(keys=['r1', 'r2', 'etc'])
```

## Using indexes to select/access 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

### 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" “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 operations

### Select DataFrame rows with .iloc[]

```
# row selection based on row number
df = df.iloc[0]           # the first row
df = df.iloc[[1, 3, 5]]   # rows 2, 3, 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.

## Working with Columns (and pandas Series)

### Peek at the column/Series structure/contents

```
s = df[col].head(i)    # get first i elements
s = df[col].tail(i)    # get last i elements
s = df[col].describe() # summary stats
df[col].dtype          # column data type
```

### Adding new columns to a DataFrame

```
df['new_col'] = range(len(df))
df['new_col'] = np.repeat(np.nan, len(df))
df['random'] = np.random.rand(len(df))
df['index_as_col'] = df.index
df1[['b', 'c']] = df2[['e', 'f']]
```

**Trap**: When adding a new column, only items that have a corresponding index in the DataFrame will be added. The index of the receiving DataFrame is not extended to accommodate all of the series.
**Trap**: when adding a python list or numpy array, the column will be added by integer position.

**Add a mismatched column with an extended index**

```
df = pd.DataFrame([1, 2, 3], index=[1, 2, 3])
s =  pd.Series([2, 3, 4], index=[2, 3, 4])
df = df.reindex(df.index.union(s.index))      #!!!
df['s'] = s # with NaNs where no data
```

**Note**: assumes unique index values

**Dropping (deleting) columns (mostly by label)**

```
df = df.drop(col1, axis=1)
df = df.drop([col1, col2], axis=1)

del df[col]          # classic python works!

df = df.drop(df.columns[0], axis=1)   # first
df = df.drop(df.columns[-2:], axis=1) # last two
```

**Data type conversions**

```
df[col].dtype                 # get data type
s = df[col].astype('float')
s = df[col].astype('int')
s = df[col].astype('str')
s = df[col].astype('string')

s = pd.to_numeric(df[col])
s = pd.to_datetime(df[col])
s = pd.Categorical(df[col])
```

**Hint**: prefer "string" over "str". Str encodes to object, string encodes to the new pandas string data type.
**Note**: although related, python data types and pandas data types are different!

**Swap column contents**

```
df[['B', 'A']] = df[['A', 'B']]
```

**Vectorised arithmetic on columns**

```
df['proportion'] = df['count'] / df['total']
df['percent'] = df['proportion'] * 100.0
```

**Apply numpy mathematical functions to columns**

```
df['log_data'] = np.log(df[col])
```

**Note**: many many more numpy math functions
**Hint**: Prefer pandas math over numpy where you can.

**Set column values based on criteria**

```
df[b] = df[a].where(df[a]>0, other=0)
df[d] = df[a].where(df.b!=0, other=df.c)
```

**Note**: where other can be a Series or a scalar

**Common column-wide methods/attributes**

```
value = df[col].dtype      # type of data
value = df[col].size       # col dimensions
value = df[col].count()    # non-NA count
value = df[col].sum()
value = df[col].prod()
value = df[col].min()
value = df[col].max()
value = df[col].mean()     # also median()
value = df[col].cov(df[other_col])

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

**Find first row index label for min/max val in column**

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

**Common column element-wise methods**

```
s = df[col].isna()
s = df[col].notna()    # not isna()
s = df[col].astype('float')
s = df[col].abs()
s = df[col].round(decimals=0)
s = df[col].diff(periods=1)
s = df[col].shift(periods=1)
s = df[col].to_datetime()
s = df[col].fillna(0)    # replace NaN w 0
```

```
s = df[col].cumsum()
s = df[col].cumprod()
s = df[col].pct_change(periods=4)
s = df[col].rolling(window=4,
    min_periods=4, center=False).sum()
```

**Append a column of row sums to a DataFrame**

```
df['Row Total'] = df.sum(axis=1)
```

**Note**: also means, mins, maxs, etc.

**Multiply every column in DataFrame by a Series**

```
df = df.mul(s, axis=0)  # on matched rows
```

**Note**: also add, sub, div, etc.

**Selecting columns with .loc, .iloc**

```
df = df.loc[:, 'col1':'col2'] # inclusive
df = df.iloc[:, 0:2]          # exclusive
```

**Get the integer position of a column index label**

```
i = df.columns.get_loc('col_name')
```

**Test if column index values are unique/monotonic**

```
b = df.columns.is_unique
b = df.columns.is_monotonic_increasing
b = df.columns.is_monotonic_decreasing
```

**Find the largest and smallest values in a column**

```
s = df[col].nlargest(n)
s = df[col].nsmallest(n)
```

**Note**: these methods return a series

**Mapping a DataFrame column or Series**

```
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.

**Sorting the columns of a DataFrame**

```
df = df.sort_index(axis=1, ascending=False)
```

**Note**: the column labels need to be comparable

## Working with rows

**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 to concat is a list of DataFrames. All DataFrames must have the same column labels.

**Append a row of column totals to a DataFrame**

```
df.loc['Total'] = df.sum()
```

**Note**: best if all columns are numeric

**Dropping rows (by label)**

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

**Sorting the rows of a DataFrame by the row index**

```
df = df.sort_index(ascending=False)
```

**Sorting DataFrame rows based on column values**

```
df = df.sort_values(by=df.columns[0],
                    ascending=False)
df = df.sort_values(by=[col1, col2])
```

**Drop duplicates in the row index**

```
df = df[~df.index.duplicated(keep='first')]
```

**Test if two DataFrames have same row index**

```
df.index.equals(other_df.index)
```

**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/mask.

**Get integer position of rows that meet condition**

```
a = np.where(df[col] >= 2) #numpy array
```

**Test if the row index values are unique/monotonic**

```
if df.index.is_unique: pass # ...
b = df.index.is_monotonic_increasing
b = df.index.is_monotonic_decreasing
```

**Find row index duplicates**

```
if df.index.has_duplicates:
    print(df.index.duplicated())
```

**Note**: also similar for column label duplicates.

## Working with missing data

**Working with missing data**
Pandas uses the not-a-number construct (np.nan and float('nan')) to indicate missing data. The Python None can arise in data as well. It is also treated as missing data; as is the pandas not-a-time construct (pd.NaT).

**Missing data in a Series**

```
s = pd.Series([8,  None,  float('nan'),  np.nan])
            #[8,  NaN,   NaN,           NaN]

s.count()    # 1 - count non missing values
s.isna()     #[False,  True,   True,   True]
s.notna()    #[True,   False,  False,  False]
s.fillna(0)  #[8,      0,      0,      0]
s.ffill()    #[8,      8,      8,      8]
```

**Missing data in a DataFrame**

```
df = df.dropna()          # drop all rows with NaN
df = df.dropna(axis=1)    # same for cols

df = df.dropna(how='all') # only drop all-NaN rows
df = df.dropna(thresh=2)  # drop rows with 2+ NaNs

# only drop row if NaN in a specified col
df = df.dropna(df['col'].notnull())
```

**Recode missing data**

```
df = df.fillna(0)            # np.nan → 0
s =  df[col].fillna(0)       # np.nan → 0
df = df.ffill()              # forward fill
df = df.bfill()              # backward fill
df = df.interpolate()        # interpolation
df = df.replace(r'\s+', np.nan, regex=True)
                        # white space → np.nan
```

**Non-finite numbers**
With floating point numbers, pandas provides for positive and negative infinity.

```
s = pd.Series([float('inf'), float('-inf'),
               np.inf, -np.inf])
```

Pandas treats numeric comparisons with plus or minus infinity as expected.

**Testing for finite numbers**
(using the data from the previous example)

```
b = np.isfinite(s)
```

## Working with dates, times and their indexes

Pandas has four date-time like objects that can used for data in a Series or in an Index:

| Concept | Data | Index |
|---|---|---|
| Point | Timestamp | DatetimeIndex |
| Span | Period | PeriodIndex |
| Delta | Timedelta | TimedeltaIndex |
| Offset | DateOffset | None |

**Timestamps**
Timestamps represent a point in time.

```
t = pd.Timestamp('2019-01-01')
t = pd.Timestamp('2019-01-01 21:15:06')
t = pd.Timestamp(year=2019, month=1,
      day=1, hour=21, minute=15, second=6.49,
      tz='Australia/Adelaide')
      # handles daylight savings time
```

**Note**: the dtype is datetime64[ns] or datetime64[ns,tz]

### Minimum and Maximum dates

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

### Current date and time

```
pd.Timestamp('now')
# Timestamp('2022-02-17 13:28:25.490814')
```

### From a list of strings to a Series of Timestamps

```
dates = pd.Series(['1/1/1919', '02/01/2020'])
ts = pd.to_datetime(dates, dayfirst=True)
print(ts)  # ['1919-01-01', '2020-01-02']

# Formatting non-standard date times...
t = pd.Series(['09:08:55.7654-JAN092002',
               '15:42:02.6589-FEB082016'])
s = pd.to_datetime(t,
                   format="%H:%M:%S.%f-%b%d%Y")
# [2002-01-09 09:08:55.765400,
#  2016-02-08 15:42:02.658900]
```

**Also**: %B = full month name; %m = numeric month; %y = year without century; and more …
**Note**: if we pass the pd.to_datetime() helper function a pandas Series it returns a pandas Series of Timestamps. If we pass it a list of date strings, it returns a pandas DatetimeIndex.

### From python datetime.datetime to Timestamp

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

### Date ranges

The helper function pd.date_range() can be useful for constructing a range of dates (either as a DatetimeIndex, or as a column in a DataFrame)

```
dti = pd.date_range('2015-01', periods=2,
                    freq='M') # end of month
print(dti)  # ['2015-01-31', '2015-02-28']

dti = pd.date_range('2019-01-01', periods=2,
                    freq='D') # daily
print(dti)  # ['2019-01-01', '2019-01-02']
```

### Using the .dt accessor

```
t = pd.Timestamp('2021-03-24')
month = t.dt.month
year = t.dt.year
```

Also .year .day .month_name() .day_name() .hour .minute .second .weekofyear .dayofweek .dayofyear .isocalendar().week .quarter

### Converting to python datetime.datetime objects

A DatetimeIndex can be converted to an array of python datetime objects using the .to_pydatetime() method.

```
dti = pd.DatetimeIndex(pd.date_range(
  start='1/1/2011', periods=4, freq='M'))
a = dti.to_pydatetime()       # numpy array
```

A pandas series of Timestamps can be converted to an array of python datetime objects using the .dt accessor.

```
dates = pd.Series(['1/1/1919', '02/01/2020'])
ts = pd.to_datetime(dates, dayfirst=True)
a = ts.dt.to_pydatetime() # to datetime.datetime
```

### From Timestamps to Python dates or times

```
[x.date() for x in df['TS']]
[x.time() for x in df['TS']]
```

Note: converts to datatime.date or datetime.time

### Period

```
# periods represent time spans
p = pd.Period('2021', freq='Y')         # year
p = pd.Period('2021-Q1', freq='Q')      # quarter
p = pd.Period('2021-01', freq='M')      # month
p = pd.Period('2021-01-01', freq='D')   # day
p = pd.Period('2021-01-01 21:15:06',freq='S') #sec

# periods are the basis for PeriodIndex
ps = df['Date'].dt.to_period(freq='D')
pi = pd.PeriodIndex(df['Date'], freq='M')
pi = pd.period_range('2021-01',periods=3,freq='M')
```

### Period conversions

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

### 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 |
| QS-{JAN, FEB, …} | Quarter start with year starting (QS – December) |
| Q-{JAN, FEB, …} | Quarter end with year ending (Q – December) |
| AS-{JAN, FEB, …} | Year start (AS - December) |
| A-{JAN, FEB, …} | Year end (A - December) |

### Deltas

Timedeltas represent the difference between two pandas timestamps. When we subtract a Timestamp from another Timestamp, we get a Timedelta object in pandas.

```
delta = pd.Timedelta(days=1, hours=1)
ts = pd.Series(pd.date_range('2019-01-01',
                             periods=31, freq='D'))
delta_series = ts.diff(1)
```

### Converting a Timedelta to a numeric

```
l = ['2019-04-01', '2019-09-03']
s = pd.to_datetime(pd.Series(l))

day = pd.Timedelta(days=1)
delta_days = s / day

minute = pd.Timedelta(minutes=1)
delta_minutes = delta / minute
```

**Offsets**

Subtracting a Period from a Period gives an offset.

```
offset = pd.DateOffset(days=4)
s = pd.Series(pd.period_range('2019-01-01',
    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**

```
x = offset.n # an individual offset
t = s.apply(lambda z: np.nan if z is np.nan
        else z.n) # convert a Series
```

**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().

**Time zones**

```
t = ['2015-06-30 00:00:00',
      '2015-12-31 00:00:00']
dti = pd.to_datetime(t
      ).tz_localize('Australia/Canberra')
dti = dti.tz_convert('UTC')
ts = pd.Timestamp('now',
      tz='Europe/London')
```

**Note**: by default, Timestamps are created without time zone information.

**Row selection with a time-series index**

```
# start with some play data
n = 48
df = pd.DataFrame(np.random.randint(low=0,
    high=999, size=(n, 5)),
    index=pd.period_range('2015-01',
        periods=n, freq='M'))

february_selector = (df.index.month == 2)
february_data = df[february_selector]

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

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

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

**Also**: year, month, day [of month], hour, minute, second, dayofweek, weekofmonth, weekofyear [numbered from 1], week starts on Monday], dayofyear [from 1], …

**The Series.dt accessor attribute**

DataFrame columns that contain datetime-like objects can be manipulated with the .dt accessor attribute

```
t = ['2012-04-14 04:06:56.307000',
     '2011-05-14 06:14:24.457000',
     '2010-06-14 08:23:07.520000']

# a Series of time stamps
s = pd.Series(pd.to_datetime(t))
print(s.dtype)      # datetime64[ns]
print(s.dt.second)  # 56, 24,  7
print(s.dt.month)   #  4,  5,  6
# a Series of time periods
s = pd.Series(pd.PeriodIndex(t, freq='Q'))
print(s.dtype)      # int64
print(s.dt.quarter) # 2,    2,    2
print(s.dt.year)    # 2012, 2011, 2010
```

## Joining/Combining DataFrames

Three ways to join DataFrames:

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

**Simple concatenation is often what you want**

```
df = pd.concat([df1,df2], axis=0) #top/bottom
df = pd.concat([df1,df2]).sort_index() # t/b

df = pd.concat([df1,df2], axis=1) #left/right
```

**Trap**: can end up with duplicate rows or cols
**Note**: concat has an ignore_index parameter
**Note**: if no axis is specified, defaults to top/bottom.

**Merge**

```
df_new = pd.merge(left=df1, right=df2,
          how='outer', left_index=True,
          right_index=True) # on indexes
df_new = pd.merge(left=df1, right=df2,
          how='left', left_on='col1',
          right_on='col2') # on columns

df_new = df.merge(right=dfg, how='left',
          left_on='Group', right_index=True)
```

**How**: 'left', 'right', 'outer', 'inner' (where outer=union/all; inner=intersection)
**Note**: merge is both a pandas helper-function, and a DataFrame method
**Note**: DataFrame.merge() joins on common columns by default (if left and right not specified)
**Trap:** When joining on column values, the indexes on the passed DataFrames are ignored.
**Trap**: many-to-many merges can result in an explosion of associated data.

**Join on row indexes (another way of merging)**

```
df = df1.join(other=df2, how='outer')
df = df1.join(other=df2, on=['a','b'],
        how='outer')
```

**Note**: DataFrame.join() joins on indexes by default.

**Combine_first**

```
df = df1.combine_first(other=df2)

# multi-combine with python reduce()
df = reduce(lambda x, y:
        x.combine_first(other=y),
            [df1, df2, df3, df4, df5])
```

Combine_first uses the non-null values from df1. Null values in df1 are filled with values from the same location in df2. The index of the combined DataFrame will be the union of the indexes from df1 and df2.

## Groupby: Split-Apply-Combine

**Grouping**

```
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.

**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: 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({
    '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 similar to 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']) >= 11)
df11 = gb.filter(eleven)
```

**Group by a row index (non-hierarchical index)**

```
df = df.set_index(keys='cat')
s = df.groupby(level=0)[col].sum()
dfg = df.groupby(level=0).sum()
```

## Pivot Tables: working with long and wide data

**These features work with and often create hierarchical or multi-level Indexes; (the pandas MultiIndex is powerful and complex).**

**Pivot, unstack, stack and melt**
Pivot tables move from long format to wide format data

```
# Let's start with data in long format
from io import StringIO
data = """Date,Pollster,State,Party,Est
13/03/2014, Newspoll, NSW,   red,   25
13/03/2014, Newspoll, NSW,   blue,  28
13/03/2014, Newspoll, Vic,   red,   24
13/03/2014, Newspoll, Vic,   blue,  23
13/03/2014, Galaxy,   NSW,   red,   23
13/03/2014, Galaxy,   NSW,   blue,  24
13/03/2014, Galaxy,   Vic,   red,   26
13/03/2014, Galaxy,   Vic,   blue,  25
13/03/2014, Galaxy,   Qld,   red,   21
13/03/2014, Galaxy,   Qld,   blue,  27"""
df = pd.read_csv(StringIO(data),
      header=0, skipinitialspace=True)

# pivot to wide format on 'Party' column
# 1st: set up a MultiIndex for other cols
df1 = df.set_index(['Date', 'Pollster',
    'State'])
# 2nd: do the pivot
wide1 = df1.pivot(columns='Party')
```

```
# unstack to wide format on State / Party
# 1st: MultiIndex all but the Values col
df2 = df.set_index(['Date', 'Pollster',
    'State', 'Party'])
# 2nd: unstack a column to go wide on it
wide2 = df2.unstack('State')
wide3 = df2.unstack() # pop last index

# Use stack() to get back to long format
long1 = wide1.stack()
# Then use reset_index() to remove the
# MultiIndex.
long2 = long1.reset_index()

# Or melt() back to long format
# 1st: flatten the column index
wide1.columns = ['_'.join(col).strip()
    for col in wide1.columns.values]
# 2nd: remove the MultiIndex
wdf = wide1.reset_index()
# 3rd: melt away
long3 = pd.melt(wdf, value_vars=
    ['Est_blue', 'Est_red'],
    var_name='Party', id_vars=['Date',
    'Pollster', 'State'])
```

**Note**: See documentation, there are many arguments to these methods.

## Plotting from the DataFrame

### Import matplotlib, choose a matplotlib style

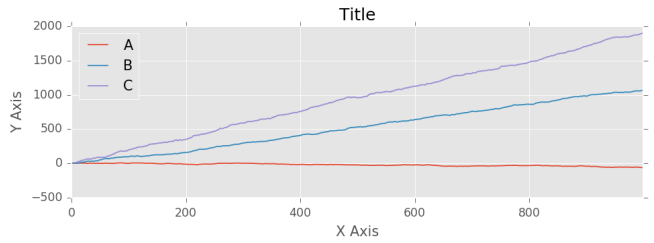
```
import matplotlib.pyplot as plt
print(plt.style.available)
plt.style.use('ggplot')
```

### Fake up some data (which we reuse repeatedly)

```
a = np.random.normal(0, 1, 999)
b = np.random.normal(1, 2, 999)
c = np.random.normal(2, 3, 999)
df = pd.DataFrame([a, b, c]).T
df.columns =['A', 'B', 'C']
```

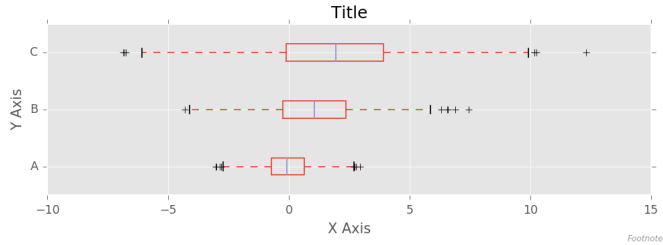
### Line plot

```
df1 = df.cumsum()
ax = df1.plot()

# from here down - standard plot output
ax.set_title('Title')
ax.set_xlabel('X Axis')
ax.set_ylabel('Y Axis')

fig = ax.figure
fig.set_size_inches(8, 3)
fig.tight_layout(pad=1)
fig.savefig('filename.png', dpi=125)

plt.close()
```




### Box plot

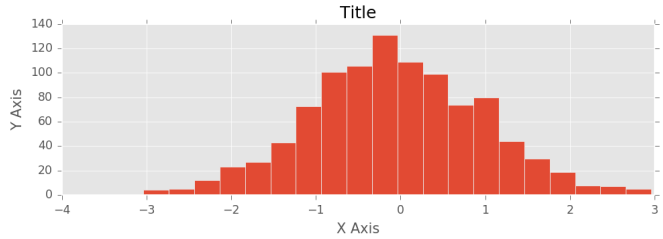
```
ax = df.plot.box(vert=False)
# followed by the standard plot code as above
```




```
ax = df.plot.box(column='c1', by='c2')
```

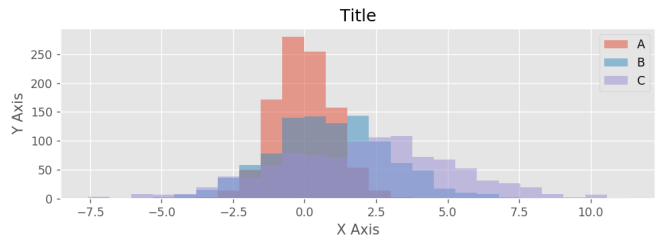
### Histogram

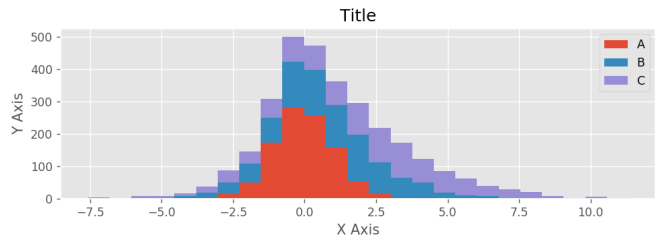
```
ax = df['A'].plot.hist(bins=20)
# followed by the standard plot code as above
```




### Multiple histograms (overlapping or stacked)

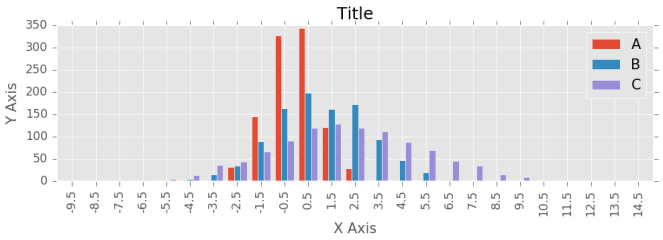
```
ax = df.plot.hist(bins=25, alpha=0.5) # or...
ax = df.plot.hist(bins=25, stacked=True)
# followed by the standard plot code as above
```



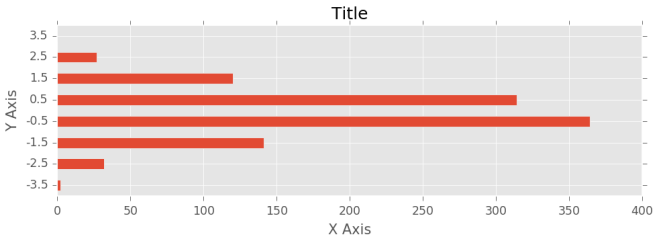
### Bar plots

```
bins = np.linspace(-10, 15, 26)
binned = pd.DataFrame()
for x in df.columns:
    y=pd.cut(df[x],bins,labels=bins[:-1])
    y=y.value_counts().sort_index()
    binned = pd.concat([binned,y],axis=1)
binned.index = binned.index.astype('float')
binned.index += (np.diff(bins) / 2.0)
ax = binned.plot.bar(stacked=False,
    width=0.8) # for bar width
# followed by the standard plot code as above
```



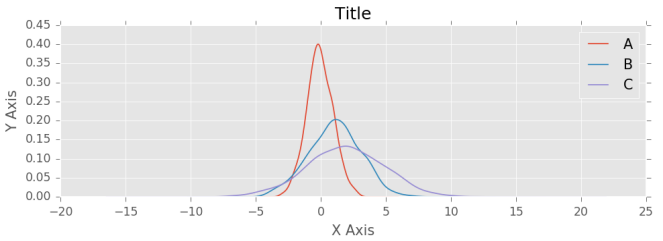

### Horizontal bars

```
ax = binned['A'][(binned.index >= -4) &
    (binned.index <= 4)].plot.barh()
# followed by the standard plot code as above
```



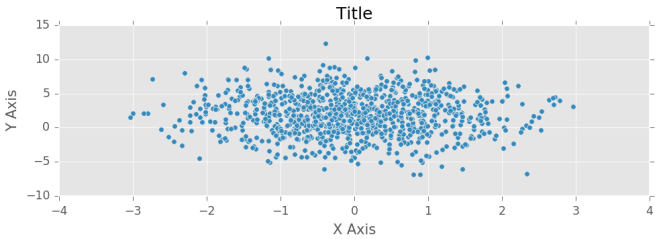

### Density plot

```
ax = df.plot.kde()
# followed by the standard plot code as above
```



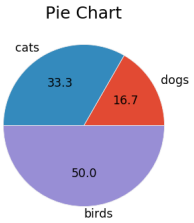

### Scatter plot

```
ax = df.plot.scatter(x='A', y='C')
# followed by the standard plot code as above
```




### Pie chart

```
s = pd.Series(data=[10, 20, 30],
    index = ['dogs', 'cats', 'birds'])
ax = s.plot.pie(autopct='%.1f')

# followed by the standard plot output ...
ax.set_title('Pie Chart')
ax.set_aspect(1) # make it round
ax.set_ylabel('') # remove default

fig = ax.figure
fig.set_size_inches(8, 3)
fig.savefig('filename.png', dpi=125)

plt.close(fig)
```

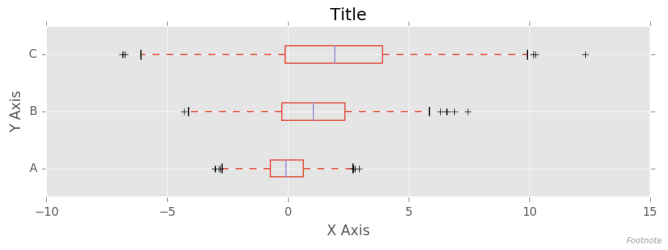



### Change the range plotted

```
ax.set_xlim([-5, 5])

# for some white space on the chart ...
lower, upper = ax.get_ylim()
ax.set_ylim([lower-1, upper+1])
```

### Add a footnote to the chart

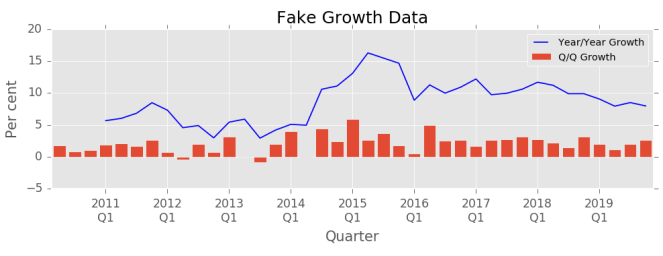
```
# after the fig.tight_layout(pad=1) above
fig.text(0.99, 0.01, 'Footnote',
    ha='right', va='bottom',
    fontsize='x-small',
    fontstyle='italic', color='#999999')
```



**A line and bar on the same chart**

In matplotlib, bar charts visualise categorical or discrete data. Line charts visualise continuous data. This makes it hard to get bars and lines on the same chart. Typically combined charts either have too many labels, and/or the lines and bars are misaligned or missing. You need to trick matplotlib a bit … pandas makes this tricking easier

```
# start with fake percentage growth data
s = pd.Series(np.random.normal(
    1.02, 0.015, 40))
s = s.cumprod()
dfg = (pd.concat([s / s.shift(1),
    s / s.shift(4)], axis=1) * 100) - 100
dfg.columns = ['Quarter', 'Annual']
dfg.index = pd.period_range('2010-Q1',
    periods=len(dfg), freq='Q')

# reindex with integers from 0; keep old
old = dfg.index
dfg.index = range(len(dfg))

# plot the line from pandas
ax = dfg['Annual'].plot(color='blue',
    label='Year/Year Growth')

# plot the bars from pandas
dfg['Quarter'].plot.bar(ax=ax,
    label='Q/Q Growth', width=0.8)

# relabel the x-axis more appropriately
ticks = dfg.index[((dfg.index+0)%4)==0]
labs = pd.Series(old[ticks]).astype('str')
ax.set_xticks(ticks)
ax.set_xticklabels(labs.str.replace('Q',
    '\nQ'), rotation=0)

# fix the range of the x-axis … skip 1st
ax.set_xlim([0.5,len(dfg)-0.5])

# add the legend
l=ax.legend(loc='best',fontsize='small')

# finish off and plot in the usual manner
ax.set_title('Fake Growth Data')
ax.set_xlabel('Quarter')
ax.set_ylabel('Per cent')

fig = ax.figure
fig.set_size_inches(8, 3)
fig.tight_layout(pad=1)
fig.savefig('filename.png', dpi=125)

plt.close()
```




## Working with Categorical Data

**Categorical data**

The pandas Series has an R factors-like data type for encoding categorical data.

```
s = pd.Series(['a','b','a','c','b','d','a'],
                        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.

**Convert back to the original data type**

```
s = pd.Series(['a','b','a','c','b','d','a'],
                        dtype='category')
s = s.astype('str')
```

**Ordering, reordering and sorting**

```
s = pd.Series(list('abc'), dtype='category')
print (s.cat.ordered)
s = s.cat.reorder_categories(['b', 'c', 'a'])
s = s.sort_values()
s.cat.ordered = False
```

**Trap**: category must be ordered for it to be sorted

**Renaming categories**

```
s = pd.Series(list('abc'), dtype='category')
s.cat.categories = [1, 2, 3]      # in place
s = s.cat.rename_categories([4, 5, 6])
# using a comprehension ...
s.cat.categories = ['Group ' + str(i)
    for i in s.cat.categories]
```

**Trap**: categories must be uniquely named

**Adding new categories**

```
s = s.cat.add_categories([7, 8, 9])
```

**Removing categories**

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

## Working with strings

**Working with strings**

```
# quickly let's fake-up some text data
df = pd.DataFrame("Lorem ipsum dolor sit
amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore
magna aliqua".split(), columns=['t'])

# assume that df[col] is series of strings
s1 = df['t'].str.lower()
s2 = df['t'].str.upper()
s3 = df['t'].str.len()
df2 = df['t'].str.split('t', expand=True)

# pandas strings are just like Python strings
s4 = df['t'] + '-suffix'     # concatenate
s5 = df['t'] * 5               # duplicate
```

Most python string functions are replicated in the pandas DataFrame and Series objects.

**Text matching and regular expressions (regex)**

```
s6 = df['t'].str.match('[sedo]+')
s7 = df['t'].str.contains('[em]')
s8 = df['t'].str.startswith('do') # no regex
s8 = df['t'].str.endswith('.') # no regex
s9 = df['t'].str.replace('old', 'new')
s10 = df['t'].str.extract('(pattern)')
```

**Note**: pandas has many more methods.

## Basic Statistics

**Summary statistics**

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

**DataFrame – key stats methods**

```
df.corr()     # pairwise correlation cols
df.cov()      # pairwise covariance cols
df.kurt()     # kurtosis over cols (def)
df.mad()      # mean absolute deviation
df.sem()      # standard error of mean
df.var()      # variance over cols (def)
```

**Value counts**

```
s = df[col].value_counts()
```

**Cross-tabulation (frequency count)**

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

**Quantiles and ranking**

```
quants = [0.05, 0.25, 0.5, 0.75, 0.95]
q = df.quantile(quants)
r = df.rank()
```

**Histogram binning**

```
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])
```

**Regression**

```
import statsmodels.formula.api as sm
result = sm.ols(formula="col1 ~ col2 + col3",
         data=df).fit()
print (result.params)
print (result.summary())
```

**Simple smoothing example using a rolling apply**

```
k3x5 = np.array([1,2,3,3,3,2,1]) / 15.0
s = df['A'].rolling(window=len(k3x5),
    min_periods=len(k3x5),
    center=True).apply(
         func=lambda x: (x * k3x5).sum())
# fix the missing end data ... unsmoothed
s = df['A'].where(s.isna(), other=s)
```

## Cautionary note

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 of milk and honey where it is rumoured pandas and pythons gambol together. There is no guarantee the narratives were captured and transcribed accurately. You use these notes at your own risk. You have been warned. I will not be held responsible for whatever happens to you and those you love once your eyes begin to see what is written here.

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