Coding Cheat Sheet for MA705 F19

Nathan Carter
ncarter@bentley.edu
Fall 2019

This summarizes *only the coding part* of the course.
We will cover many high-level tasks not mentioned here.

Introduction to Python

Basics

Comments, which are not executed:

```
# Start with a hash, then explain your code.
```

Print simple data:

```
print( 1 + 5 )
```

Storing data in a variable:

```
num_friends = 1000
```

Data types

Integers and real numbers ("floating point"):

```
0, 20, -3192, 16.51309, 0.003
```

Strings:

```
"You can use double quotes."
'You can use single quotes.'
'Don\'t forget backslashes when needed.'
```

Booleans:

```
True, False
```

Asking Python for the type of a piece of data:

```
type( 5 ), type( "example" ), type( my_data )
```

Converting among data types:

```
str( 5 ), int( "-120" ), float( "0.5629" )
```

Math

Basic arithmetic (+, −, ×, ÷):

```
1 + 2, 1 - 2, 1 * 2, 1 / 2
```

Exponents, integer division, and remainders:

```
1 ** 2, 1 // 2, 1 % 2
```

Lists

Create a list with square brackets:

```
small_primes = [ 2, 3, 5, 7, 11, 13, 17, 19, 23 ]
```

Lists can mix data of any type, even other lists:

```
# Sublists are name, age, height (in m)
heroes = [ [ 'Harry Potter', 11, 1.3 ],
           [ 'Ron Weasley', 11, 1.5 ],
           [ 'Hermione Granger', 11, 1.4 ] ]
```

Accessing elements from the list is zero-based:

```
small_primes[0]   # == 2
small_primes[-1]  # == 23
```

Slicing lists is left-inclusive, right-exclusive:

```
small_primes[2:4]  # == [5,7]
small_primes[:4]   # == [2,3,5,7]
small_primes[4:]   # == [11,13,17,19,23]
```

It can even use a "stride" to count by something other than one:

```
small_primes[0:7:2]     # selects items 0,2,4,6
small_primes[::3]       # selects items 0,3,6
small_primes[::-1]      # selects all, but in reverse
```

If indexing gives you a list, you can index again:

```
heroes[1][0]        # == 'Ron Weasley'
```

Modify an item in a list, or a slice all at once:

```
some_list[5] = 10
some_list[5:10] = [ 'my', 'new', 'entries' ]
```

Adding or removing entries from a list:

```
small_primes += [ 27, 29, 31 ]
small_primes = small_primes + [ 37, 41 ]
small_primes.append( 43 )  # to add just one entry
del( heroes[0] )     # Voldemort's goal
del( heroes[:] )     # or, even better, this
```

Copying or not copying lists:

```
# L will refer to the same list in memory as heroes:
L = heroes
# M will refer to a full copy of the heroes array:
M = heroes[:]
```

Functions

Calling a function and saving the result:

```
lastSmallPrime = max( small_primes )
```

Getting help on a function:

```
help( max )
```

Methods

Methods are functions that belong to an object. (In Python, every piece of data is an object.)
Examples:

```
name = 'jerry'
name.capitalize()             # == 'Jerry'
name.count( 'r' )             # == 2
flavors = [ 'vanilla', 'chocolate', 'strawberry' ]
flavors.index( 'chocolate' )  # == 1
```

Packages

Installing a package from conda:

```
conda install package_name
```

Ensuring conda forge packages are available:

```
conda config --add channels conda-forge
```

Installing a package from pip:

```
pip3 install package_name
```

Importing a package and using its contents:

```
import math
print( math.pi )
# or if you'll use it a lot and want to be brief:
import math as M
print( M.pi )
```

Importing just some functions from a package:

```
from math import pi, degrees
print( "The value of pi in degrees is:" )
print( degrees( pi ) )          # == 180.0
```

NumPy

Creating arrays from Python lists:

```
import numpy as np
a = np.array( [ 5, 10, 6, 3, 9 ] )
```

Elementise computations are supported:

```
a * 2        # == [ 10, 20, 12, 6, 18 ]
a < 10       # == [ True, False, True, True, True ]
```

Use comparisons to subset/select:

```
a[a < 10]    # == [ 5, 6, 3, 9 ]
```

Note: NumPy arrays don't permit mixing data types:

```
np.array( [ 1, "hi" ] )  # converts all to strings
```

NumPy arrays can be 2d, 3d, etc.:

```
a = np.array( [ [ 1, 2, 3, 4 ],
                [ 5, 6, 7, 8 ] ] )
a.shape      # == (2,4)
```

You can index/select with comma notation:

```
a[1,3]       # == 8
a[0:2,0:2]   # == [[1,2],[5,6]]
a[:,2]       # == [3,7]
a[0,:]       # == [1,2,3,4]
```

Fast NumPy versions of Python functions, and some new ones:

```
np.sum( a )
np.sort( a )
np.mean( a )
np.median( a )
np.std( a )
# and others
```

See the Statistical Thinking in Python section for more.

Intermediate Python for Data Science

Plots (matplotlib)

Conventional way to import matplotlib:

```
import matplotlib.pyplot as plt
```

Creating a line plot:

```
plt.plot( x_data, y_data )      # create plot
plt.show()                      # display plot
```

Creating a scatter plot:

```
plt.scatter( x_data, y_data )   # create plot
plt.show()                      # display plot
# or this alternative form:
plt.plot( x_data, y_data, kind='scatter' )
plt.show()
```

Labeling axes and adding title:

```
plt.xlabel( 'x axis label here' )
plt.ylabel( 'y axis label here' )
plt.title( 'Title of Plot' )
```

A few small ways to customize plots:

```
plt.xscale( 'log' )
plt.yticks( [ 0, 5, 10, 20 ] )
plt.grid()
```

To create a histogram:

```
plt.hist( data, bins=10 )       # 10 is the default
plt.show()
```

To "clean up" so you can start a new plot:

```
plt.clf()
```

Write text onto a plot:

```
plt.text( x, y, 'Text to write' )
```

To save a plot to a file:

```
# before plt.show(), call:
plt.savefig( 'filename.png' )  # or .jpg or .pdf
```

Also see the Graphical EDA section for more plotting ideas.

Dictionaries

Creating a dictionary directly:

```
days_in_month = {
    "january"  : 31,
    "february" : 28,
    "march"    : 31,
    "april"    : 30,
    # and so on, until...
    "december" : 31
}
```

Getting and using keys:

```
days_in_month.keys()      # == ["january",
                          #     "february",...]
days_in_month["april"]    # == 30
```

Updating dictionary and checking membership:

```
days_in_month["february"] = 29   # update for 2020
"tuesday" in days_in_month       # == False
days_in_month["tuesday"] = 9     # a mistake
"tuesday" in days_in_month       # == True
del( days_in_month["tuesday"] )  # delete mistake
"tuesday" in days_in_month       # == False
```

Pandas (DataFrames)

Build manually from dictionary:

```
import pandas as pd
df = pd.DataFrame( {
    "column label 1": [
        "this example uses...",
        "string data here."
    ],
    "column label 2": [
        100.65,   # and numerical data
        -92.04    # here, for example
    ]
    # and more columns if needed
} )
df.index = [
    "put your...",
    "row labels here."
]
```

Import from CSV file:

```
# if row and column headers are in first row/column:
df = pd.read_csv( "/path/to/file.csv",
                  index_col = 0 )
# if no row headers:
df = pd.read_csv( "/path/to/file.csv" )
# if no column headers:
df = pd.read_csv( "/path/to/file.csv",
                  index_col = 0, header = None,
                  names = ['column','names','here'] )
# if any missing data you want to mark as NaN:
# (na_values can be a list of patterns,
# or a dict mapping column names to patterns/lists)
df = pd.read_csv( "/path/to/file.csv",
                  na_values = 'pattern to replace' )
# and many other options!  (see the documentation)
```

Export to CSV or XLSX file:

```
df.to_csv( "/path/to/output_file.csv" )
df.to_excel( "/path/to/output_file.xlsx" )
```

Indexing and selecting data:

```
df["column name"]     # is a "Series" (labeled column)
df["column name"].values()
                      # extract just its values
df[["column name"]]   # is a 1-column dataframe
df[["col1","col2"]]   # is a 2-column dataframe
df[n:m]               # slice of rows, a dataframe
df.loc["row name"]    # is a "Series" (labeled column)
                      # yes, the row becomes a column
df.loc[["row name"]]  # 1-row dataframe
df.loc[["r1","r2","r3"]]
                      # 3-row dataframe
df.loc[["r1","r2","r3"],:]
                      # same as previous
df.loc[:,["c1","c2","c3"]]
                      # 3-column dataframe
df.loc[["r1","r2","r3"],["c1","c2"]]
                      # 3x2 slice of the dataframe
df.iloc[[5]]          # is a "Series" (labeled column)
                      # contains the 6th row's data
df.iloc[[5,6,7]]      # 3-row dataframe (6th-8th)
df.iloc[[5,6,7],:]    # same as previous
df.iloc[:,[0,4]]      # 2-column dataframe
df.iloc[[5,6,7],[0,4]]
                      # 3x2 slice of the dataframe
```

You can also create a plot from a `Series` or dataframe:

```
df.plot()             # or series.plot()
plt.show()
# or to show each column in a subplot:
df.plot( subplots = True )
plt.show()
# or to plot certain columns:
df.plot( x='col name', y='other col name' )
plt.show()
```

Other useful dataframe tools:

```
df.head(5)            # first five rows
df.tail(5)            # last five rows
series.head(5)        # head, tail also work on series
df.info()             # summary of the data types used
df.describe()         # summary statistics
# df.describe() makes calls to df.mean(), df.std(),
# df.median(), df.quantile(), etc...
df.columns = [ 'col name 1', 'col name 2', ... ]
                      # set the column headers
# remove rows or columns:
df = df.drop( [ 'column', 'names' ], axis='columns' )
```

Python relations work on NumPy arrays and Pandas Series:

```
<, <=, >, >=, ==, !=
```

Logical operators can combine the above relations:

```
and, or, not           # use these on booleans
np.logical_and(x,y)    # use these on numpy arrays
np.logical_or(x,y)     # (assuming you have imported
np.logical_not(x)      # numpy as np)
```

Filtering Pandas DataFrames:

```
series = df["column"]
filter = series > some_number
df[filter]  # new dataframe, a subset of the rows
# or all at once:
df[df["column"] > some_number]
# combining multiple conditions:
df[np.logical_and( df["population"] > 5000,
                   df["area"] < 1250 )]
```

Conditional statements:

```
# Take an action if a condition is true:
if put_condition_here:
    take_an_action()
# Take a different action if the condition is false:
if put_condition_here:
    take_an_action()
else:
    do_this_instead()
# Consider multiple conditions:
if put_condition_here:
    take_an_action()
elif other_condition_here:
    do_this_instead()
elif yet_another_condition:
    do_this_instead2()
else:
    finally_this()
```

Looping constructs:

```
while some_condition:
    do_this_repeatedly()
    # as many lines of code here as you like.
    # note that indentation is crucial!
    # be sure to work towards some_condition
    # becoming false eventually!

for item in my_list:
    do_something_with( item )

for index, item in enumerate( my_list ):
    print( "item " + str(index) +
           " is " + str(item) )
```

```
for key, value in my_dict.items():
    print( "key " + str(key) +
           " has value " + str(value) )

for item in my_numpy_array:
    # works if the array is one-dimensional
    print( item )

for item in np.nditer( my_numpy_array ):
    # if it is 2d, 3d, or more
    print( item )

for column_name in my_dataframe:
    work_with( my_dataframe[column_name] )

for row_name, row in my_dataframe.iterrows():
    print( "row " + str(row_name) +
           " has these entries: " + str(row) )

# in dataframes, sometimes you can skip the for loop:
my_dataframe["column"].apply( function )  # a Series
```

Simulation from Random Numbers

Uniform random numbers from NumPy:

```
np.random.seed( my_int )  # choose a random sequence
# (seeds are optional, but ensure reproducibility)
np.random.rand()          # uniform random in [0,1)
np.random.randint(a,b)    # uniform random in a:b
```

See the Statistical Thinking in Python section for more.

Python Data Science Toolbox 1/2

Tuples

Tuples are like lists, but use parentheses, and are immutable.

```
t = ( 6, 1, 7 )          # create a tuple
t[0]                     # == 6
a, b, c = t              # a==6, b==1, c==7
```

Writing your own functions

Syntax for defining a function:
(A function that modifies any global variables needs the Python `global` keyword inside to identify those variables.)

```
def function_name ( arguments ):
    """Write a docstring describing the function."""
    # do some things here.
    # note the indentation!
    # and optionally:
    return some_value
    # to return multiple values: return v1, v2
```

Syntax for calling a function:
(Note the distinction between “arguments” and “parameters.”)

```
# if you do not care about a return value:
function_name( parameters )
# if you wish to store the return value:
my_variable = function_name( parameters )
# if the function returns multiple values:
var1, var2 = function_name( parameters )
```

Defining nested functions:

```
def multiply_by ( x ):
    """Creates a function that multiplies by x"""
    def result ( y ):
        """Multiplies x by y"""
        return x * y
    return result
# example usage:
df["height_in_inches"].apply(
    multiply_by( 2.54 ) )  # result is now in cm
```

Providing default values for arguments:

```
def rand_between ( a=0, b=1 ):
    """Gives a random float between a and b"""
    return np.random.rand() * ( b - a ) + a
```

Accepting any number of arguments:

```
def commas_between ( *args ):
    """Returns the args as a string with commas"""
    result = ""
    for item in args:
        result += ", " + str(item)
    return result[2:]
commas_between(1,"hi",7)    # == "1,hi,7"
```

Accepting a dictionary of arguments:

```
def inverted ( **kwargs ):
    """Interchanges keys and values in a dict"""
    result = {}
    for key, value in kwargs.items():
        result[value] = key
    return result
inverted( jim=42, angie=9 )
        # == { 42 : 'jim', 9 : 'angie' }
```

Anonymous functions:

```
lambda arg1, arg2: return_value_here
# example:
lambda k: k % 2 == 0    # detects whether k is even
```

Some examples in which anonymous functions are useful:

```
list( map( lambda k: k%2==0, [1,2,3,4,5] ) )
                                   # == [False,True,False,True,False]
list( filter( lambda k: k%2==0, [1,2,3,4,5] ) )
                                   # == [2,4]
reduce( lambda x, y: x*y, [1,2,3,4,5] )
                                   # == 120 (1*2*3*4*5)
```

Raising errors if users call your functions incorrectly:

```
# You can detect problems in advance:
def factorial ( n ):
    if type( n ) != int:
        raise TypeError( "n must be an int" )
    if n < 0:
        raise ValueError( "n must be nonnegative" )
    return reduce( lambda x,y: x*y, range( 2, n+1 ) )

# Or you can let Python detect them:
def solve_equation ( a, b ):
    """Solves a*x+b=0 for x"""
    try:
        return -b / a
    except:
        return None
solve_equation( 2, -1 )    # == 0.5
solve_equation( 0, 5 )     # == None
```

Python Data Science Toolbox 2/2

Iterables and Iterators

To convert an iterable to an iterator and use it:

```
my_iterable = [ 'one', 'two', 'three' ]  # example
my_iterator = iter( my_iterable )
first_value = next( my_iterator )         # 'one'
second_value = next( my_iterator )        # 'two'
# and so on
```

To attach indices to the elements of an iterable:

```
my_iterable = [ 'one', 'two', 'three' ]  # example
with_indices = enumerate( my_iterable )
my_iterator = iter( with_indices )
first_value = next( my_iterator )         # (0,'one')
second_value = next( my_iterator )        # (1,'two')
# and so on; see also "Looping Constructs" earlier
```

To join iterables into tuples, use `zip`:

```
iterable1 = range( 5 )
iterable2 = 'five!'
iterable3 = [ 'How', 'are', 'you', 'today', '?' ]
all = zip( iterable1, iterable2, iterable3 )
next( all )     # (0,'f','How')
next( all )     # (1,'i','are')
# and so on, or use this syntax:
for x, y in zip( iterable1, iterable2 ):
    do_something_with( x, y )
```

Think of `zip` as converting a list of rows into a list of columns, a “matrix transpose,” which is its own inverse:

```
row1 = [ 1, 2, 3 ]
row2 = [ 4, 5, 6 ]
cols = zip( row1, row2 )        # swap rows and columns
print( *cols )                  # (1,4) (2,5) (3,6)
cols = zip( row1, row2 )        # restart iterator
undo1, undo2 = zip( *cols )     # swap rows/cols again
print( undo1, undo2 )           # (1,2,3) (4,5,6)
```

Pandas can read CSV files into DataFrames in chunks, creating an iterable out of a file too large for memory:

```
import pandas as pd
for chunk in pd.read_csv( filename, chunksize=100 ):
    process_one_chunk( chunk )
```

List and Dict Comprehensions

List comprehensions build a list from an output expression and a `for` clause:

```
[ n**2 for n in range(3,6) ]      # == [9,16,25]
```

You can nest list comprehensions:

```
[ (i,j) for i in range(3) for j in range(4) ]
  # == [(0,0),(0,1),(0,2),(0,3),
  #     (1,0),(1,1),(1,2),(1,3),
  #     (2,0),(2,1),(2,2),(2,3)]
```

You can put conditions on the “for” clause:

```
[ (i,j) for i in range(3) for j in range(3)
        if i + j > 2 ]  # == [ (1,2), (2,1), (2,2) ]
```

You can put conditions in the output expression:

```
some_data = [ 0.65, 9.12, -3.1, 2.8, -50.6 ]
[ x if x >= 0 else 'NEG' for x in some_data ]
  # == [ 0.65, 9.12, 'NEG', 2.8, 'NEG' ]
```

A dict comprehension creates a dictionary from an output expression in `key:value` form, plus a `for` clause:

```
{ a: a.capitalize() for a in ['one','two','three'] }
  # == { 'one':'One', 'two':'Two', 'three':'Three' }
```

Generators and Generator Functions

Just like list comprehensions, but with parentheses:

```
g = ( n**2 for n in range(3,6) )
next( g )                         # == 9
next( g )                         # == 16
next( g )                         # == 25
```

You can build generators with functions and `yield`:

```
def just_like_range ( a, b ):
    counter = a
    while counter < b:
        yield counter
        counter += 1
list( just_like_range( 5, 9 ) )   # == [5,6,7,8]
```

pandas Foundations

Time Series Data

We need a DataFrame `df` that has a date/time index.
To get a date/time index in the first place:

```
# read as dates any columns that pandas can:
df = pd.read_csv( "/path/to/file.csv",
                  parse_dates = True )
# read as dates just the columns you specify:
df = pd.read_csv( "/path/to/file.csv",
                  parse_dates = ['column','names'] )
# to use one of those columns as a date/time index:
df = pd.read_csv( "/path/to/file.csv",
                  parse_dates = True,
                  index_col = 'Date' )
# combine multiple columns to form a date:
df = pd.read_csv( "/path/to/file.csv",
                  parse_dates = [[column,indices]] )
```

If the dates are already loaded, you can still convert them:

```
dates = pd.to_datetime( df['Date column name'] )
df.set_index( date, inplace=True )
# note that conversions to other types are possible:
nums = pd.to_numeric( df['some other column'] )
```

Select contiguous blocks of rows:

```
df.loc["2019-07-21"]             # all rows on this date
df.loc["July 21, 2019"]          # same, alternate form
df.loc["2019-07"]                # all rows in this month
df.loc["2019-07":"2019-09"]      # all rows in this range
```

Reindexing a DataFrame:

```
# attempts to match old data to new index:
df.reindex( some_series )
# for missing rows, fill with earlier ones:
df.reindex( some_series, method="ffill" )
# (there is also a bfill, for back-fill)
```

Resampling a DataFrame:
(Each example computes a statistic using method chaining.)

```
df.resample( 'D' ).mean()     # D = daily
df.resample( '2W' ).count()   # 2W = every 2 weeks
# other sampling frequencies: min, H, B, M, Q, and A
```

Resampling with interpolation:

```
# assume daily data and we want hourly estimates:
df.resample( 'H' ).first().interpolate( 'linear' )
```

Smoothing data with rolling windows:

```
df['my column'].rolling( window=5 ).mean()
# window parameter specifies number of rows.
# data are placed at the end of the window.
```

String manipulations of columns via method chaining:

```
# build a copy of a column in upper case, a series:
df['string col name'].str.upper()
# compute which rows contain a specific substring:
sens = df['Last Name'].str.contains( 'sen' )
df.loc[sens,'Last Name']
# remove whitespace from column headers:
df.columns = df.columns.str.strip()
```

Date/time manipulations of columns via method chaining:

```
# fetch just the hours from a date/time column:
df['date/time column'].dt.hour
# add timezone information to a date/time Series:
times_EST = df['Date'].dt.tz_localize( 'US/Eastern' )
# such Series can then be converted to other zones:
times_GB = times_EST.dt.tz_convert( 'Europe/London' )
```

(A full list of time zone codes is on Wikipedia.)

Manipulating DataFrames with pandas

Indexing

(This builds on the DataCamp Intermediate Python section.)

```
df.iloc[5:7,0:4]      # select ranges of rows/columns
df.iloc[:,0:4]        # select a range, all rows
df.iloc[[5,6],:]      # select a range, all columns
df.iloc[5:,:]         # all but the first five rows
df.loc['A':'B',:]     # colons can take row names too
                      # (but include both endpoints)
df.loc[:,'C':'D']     # ...also column names
df.loc['D':'A':-1]    # rows by name, reverse order
```

Filtering

(This builds on the DataCamp Intermediate Python section.)

```
# avoid using np.logical_and with & instead:
df[(df["population"] > 5000)
 & (df["area"] < 1250 )]
# avoid using np.logical_or with | instead:
df[(df["population"] > 5000)
 | (df["area"] < 1250 )]
# filtering for missing values:
df.loc[:,df.all()]   # only columns with no zeroes
df.loc[:,df.any()]   # only columns with some nonzero
df.loc[:,df.isnull().any()]
                     # only columns with a NaN entry
df.loc[:,df.notnull().all()]
                     # only columns with no NaNs
df.dropna( how='any' )
                     # remove rows with any NaNs
df.dropna( how='all' )
                     # remove rows with all NaNs
```

You can filter one column based on another using these tools.

Transforming

Apply a function to each value, returning a new DataFrame:

```
def example ( x ):
    return x + 1
df.apply( example )    # adds 1 to everything
df.apply( lambda x: x + 1 )    # same
# some functions are built-in:
```

```
df.floordiv( 10 )
# many operators automatically repeat:
df['total pay'] = df['salary'] + df['bonus']
# to extend a dataframe with a new column:
df['new col'] = df['old col'].apply( f )
# slightly different syntax for the index:
df.index = df.index.map( f )
```

You can also map columns through `dict`s, not just functions.

Manipulating

Creating a Series:

```
s = pd.Series( [ 5.0, 3.2, 1.9 ] )   # just data
s = pd.Series( [ 5.0, 3.2, 1.9 ],    # data with...
  index = [ 'Mon', 'Tue', 'Wed' ] )  # ...an index
s.index[2:]                          # sliceable
s.index.name = 'Day of Week'         # index name
```

Column headings are also a series:

```
df.columns                       # is a pd.Series
df.columns.name                  # usually a string
df.columns.values                # column names array
```

Indices

Using an existing column as the index:

```
df.index = df['column name']   # once it's the index,
del df['column name']          # it can be deleted
```

Making an index from multiple columns that, when taken together, uniquely identify rows:

```
df = df.set_index( [ 'last_name', 'first_name' ] )
df.index.name                    # will be None
df.index.names                   # list of strings
df = df.sort_index()             # hierarchical sort
df.loc[('Jones', 'Heide')]       # index rows by tuples
df.loc[('Jones', 'Heide'),       # and you can fetch an
       'birth_date']             # entry that way, too
df.loc['Jones']                  # all rows of Joneses
df.loc['Jones':'Menendez']       # many last names
df.loc[(['Jones','Wu'], 'Heide'), :]
      # get both rows: Heide Jones and Heide Wu
      # (yes, the colon is necessary for rows)
df.loc[(['Jones','Wu'], 'Heide'), 'birth_date']
      # get Heide Jones's and Heide Wu's birth dates
df.loc[('Jones',['Heide','Henry']),:]
      # get full rows for Heide and Henry Jones
df.loc[('Jones',slice('Heide','Henry')),:]
      # 'Heide':'Henry' doesn't work inside tuples
```

Pivoting

If columns A and B together uniquely identify entries in column C, you can create a new DataFrame showing this:

```
new_df = df.pivot( index   = 'A',
                   columns = 'B',
                   values  = 'C' )
# or do this for all columns at once,
# creating a hierarchical column index:
new_df = df.pivot( index   = 'A',
                   columns = 'B' )
```

You can also invert pivoting, which is called “melting:”

```
old_df = pd.melt( new_df,
    id_vars = [ 'A' ],            # old index
    value_vars = [ 'values','of','column','B' ],
        # optional...pandas can often infer it
    var_name = 'B',         # these two lines just
    value_name = 'C' )      # restore column names
```

Convert hierarchical row index to a hierarchical column index:

```
# assume df.index.names is ['A','B','C']
df = df.unstack( level = 'B' )  # or A or C
# equivalently:
df = df.unstack( level = 1 )    # or 0 or 2
# and this can be inverted:
df = df.stack( level = 'B' )    # for example
```

To change the nesting order of a hierarchical index:

```
df = df.swaplevel( levelindex1, levelindex2 )
df = sort_index()               # necessary now
```

If the pivot column(s) aren’t a unique index, use `pivot_table` instead, often with an aggregation function:

```
new_df = df.pivot_table(        # this pivot table
    index   = 'A',              # is a frequency
    columns = 'B',              # table, because
    values  = 'C',              # aggfunc is count
    aggfunc = 'count' )         # (default: mean)
# other aggfuncs: 'sum', plus many functions in
# numpy, such as np.min, np.max, np.median, etc.
# You can also add column totals at the bottom:
new_df = df.pivot_table(
    index   = 'A',
    columns = 'B',
    values  = 'C',
    margins = True )            # add column sums
```

Grouping

Group all columns except column A by the unique values in column A, then apply some aggregation method to each group:

```
# example: total number of rows for each weekday
df.groupby( 'weekday' ).count()
# example: total sales in each city
df.groupby( 'city' )['sales'].sum()
# multiple column names gives a multi-level index
df.groupby( [ 'city', 'state' ] ).mean()
# you can group by any series with the same index;
# here is an example:
series = df['column A'].apply( np.round )
df.groupby( series )['column B'].sum()
```

The `agg` method lets us do even more:

```
# you can do multiple aggregations at once;
# this, too, gives a multi-level index:
df.groupby( 'weekday' ).agg( [ 'max', 'sum' ] )
# or you can pass a user-defined function:
def sum_of_squares ( series ):
    return ( series * series ).sum()
df.groupby( 'weekday' )['column name']
  .agg( sum_of_squares )
# or dictionaries can let us apply different
# aggregations to different columns:
df.groupby( 'weekday' )[['Quantity Ordered',
                         'Total Cost']]
  .agg( { 'Quantity Ordered' : 'median',
          'Total Cost'       : 'sum' } )
```

`transform` is just like `apply`, except that it must convert each value into exactly one other, thus preserving shape.

```
# example: convert values to zscores
from scipy.stats import zscore
df.groupby( 'region' )['gdp'].transform( zscore )
  .agg( [ 'min', 'max' ] )
# example: impute missing values as medians
def impute_median(series):
    return series.fillna(series.median())
grouped = df.groupby( [ 'col B', 'col C' ] )
df['col A'] = grouped['col A']
              .transform( impute_median )
```

Merging DataFrames with pandas

Loading multiple DataFrames

The `glob` module is useful:

```
from glob import glob                # built-in module
filenames = glob( '*.csv' )          # filename list
data_frames = [ pd.read_csv(f)
    for f in filenames ]             # import all files
```

Common indexes across DataFrames

You can reorder the rows in a DataFrame with `reindex`:

```
# example: if an index of month or day names were
# sorted alphabetically as strings
# rather than chronologically:
ordered_days = [ 'Mon', 'Tue', 'Wed', 'Thu',
                 'Fri', 'Sat', 'Sun' ]
df.reindex( ordered_days )
# use this to make two dataframes with a common
# index agree on their ordering:
df1.reindex( df2.index )
# in case the indices don't perfectly match,
# NaN values will be inserted, which you can drop:
df1.reindex( df2.index ).dropna()
```

You can reorder a DataFrame in preparation for reindexing:

```
# sort by index, ascending or descending:
df = df.sort_index()
df = df.sort_index( ascending=False )
# sort by a column, ascending or descending:
df = df.sort_values( 'column name',      # required
                     ascending=False )   # optional
```

Stacking DataFrames Vertically

To add one DataFrame onto the end of another:

```
big_df = df1.append( df2 )  # top: df1, bottom: df2
big_s = s1.append( s2 )     # works for Series, too
# This also stacks indices, so you usually want to:
big_df = big_df.reset_index( drop=True )
```

To add many DataFrames or series on top of one another:

```
big_df = pd.concat( [ df1, df2, df3 ] )
           .reset_index( drop=True )
# equivalently:
big_df = pd.concat( [ df1, df2, df3 ],
                    ignore_index=True )
# or add a hierarchical index to disambiguate:
big_df = pd.concat( [ df1, df2, df3 ],
                    keys=['key1','key2','key3'] )
# equivalently:
big_df = pd.concat( { key1 : df1,
                      key2 : df2,
                      key3 : df3 } )
```

Joining DataFrames Horizontally

If `df2` introduces new columns, and you want to form rows based on common indices, concat by columns:

```
big_df = pd.concat( [ df1, df2 ], axis=1 )
# equivalently:
big_df = pd.concat( [ df1, df2 ], axis='columns' )
# these accept keys=[...] also, or a dict to concat
```

By default, `concat` performs an "outer join," that is, index sets are unioned. To intersect them ("inner join") do this:

```
big_df = pd.concat( [ df1, df2 ], axis=1,
                    join='inner' )
# equivalently:
big_df = df1.join( df2, how='inner' )
```

Inner joins on non-index columns are done with `merge`.

```
# default merges on all columns present
# in both dataframes:
merged = pd.merge( df1, df2 )
# or you can choose your column:
merged = pd.merge( df1, df2, on='colname' )
# or multiple columns:
merged = pd.merge( df1, df2, on=['col1','col2'] )
# if the columns have different names in each df:
merged = pd.merge( df1, df2,
    left_on='col1', right_on='col2' )
# to specify meaningful suffixes to replace the
# default suffixes _x and _y:
merged = pd.merge( df1, df2,
    suffixes=['_from_2011','_from_2012'] )
# you can also specify left, right, or outer joins:
merged = pd.merge( df1, df2, how='outer' )
```

We often have to sort after merging (maybe by a date index), for which there is `merge_ordered`. It most often goes with an outer join, so that's its default.

```
# instead of this:
merged = pd.merge( df1, df2, how='outer' )
           .sorted_values( 'colname' )
# do this, which is shorter and faster:
merged = pd.merge_ordered( df1, df2 )
# it accepts same keyword arguments as merge,
# plus fill_method, like so:
merged = pd.merge_ordered( df1, df2,
           fill_method='ffill' )
```

When dates don't fully match, you can round dates in the right DataFrame up to the nearest date in the left DataFrame:

```
merged = pd.merge_asof( df1, df2 )
```

Intro to SQL for Data Science

SQL ("sequel") means Structured Query Language. A SQL database contains tables, each of which is like a DataFrame.

```
-- A single-line SQL comment

/*
A multi-line
SQL comment
*/
```

Selecting

To fetch one column from a table:

```
SELECT column_name FROM table_name;
```

To fetch multiple columns from a table:

```
SELECT column1, column2 FROM table_name;
SELECT * FROM table_name;    -- all columns
```

To remove duplicates:

```
SELECT DISTINCT column_name
FROM table_name;
```

To count rows:

```
SELECT COUNT(*)
FROM table_name;     -- counts all the rows
SELECT COUNT(column_name)
FROM table_name;     -- counts the non-
     -- missing values in just that column
SELECT COUNT(DISTINCT column_name)
FROM table_name;     -- # of unique entries
```

If a result is huge, you may want just the first few lines:

```
SELECT column FROM table_name
LIMIT 10;            -- only return 10 rows
```

Filtering

(selecting a subset of the rows using the WHERE keyword)
Using the comparison operators <, >, =, <=, >=, and <>, plus the inclusive range filter BETWEEN:

```
SELECT * FROM table_name
WHERE quantity >= 100;  -- numeric filter
SELECT * FROM table_name
WHERE name = 'Jeff';     -- string filter
```

Using range and set filters:

```
SELECT title,release_year FROM films
WHERE release_year BETWEEN 1990 AND 1999;
                         -- range filter
SELECT * FROM employees
WHERE role IN ('Engineer','Sales');
                         -- set filter
```

Finding rows where specific columns have missing values:

```
SELECT * FROM employees
WHERE role IS NULL;
```

Combining filters with AND, OR, and parentheses:

```
SELECT * FROM table_name
WHERE quantity >= 100
  AND name = 'Jeff';     -- one combination
SELECT title,release_year FROM films
WHERE release_year >= 1990
  AND release_year <= 1999
  AND ( language = 'French'
     OR language = 'Spanish' )
  AND gross > 2000000;   -- many
```

Using wildcards (% and _) to filter strings with LIKE:

```
SELECT * FROM employees
WHERE name LIKE 'Mac%'; -- e.g., MacEwan
SELECT * FROM employees
WHERE id NOT LIKE '%00';-- e.g., 352800
SELECT * FROM employees
WHERE name LIKE 'D_n';  -- e.g., Dan, Don
```

Aggregating

We've seen this function before; it is an aggregator:

```
SELECT COUNT(*)
FROM table_name;     -- counts all the rows
```

Some other aggregating functions: SUM, AVG, MIN, MAX. The resulting column name is the function name (e.g., MAX).
To give a more descriptive name:

```
SELECT MIN(salary) AS lowest_salary,
       MAX(salary) AS highest_salary
FROM employees;
```

You can also do arithmetic on columns:

```
SELECT budget/1000 AS budget_in_thousands
FROM projects;       -- convert a column
SELECT hours_worked * hourly_pay
FROM work_log WHERE date > '2019-09-01';
                     -- create a column
SELECT count(start_date)*100.0/count(*)
FROM table_name;     -- percent not missing
```

Sorting

Sorting happens only after selecting:

```
SELECT * FROM employees
ORDER BY name;       -- ascending order
SELECT * FROM employees
ORDER BY name DESC; -- descending order
SELECT name,salary FROM employees
ORDER BY role,name; -- multiple columns
```

Grouping

Grouping happens after selecting but before sorting. It is used when you want to apply an aggregate function like COUNT or AVG not across the whole result set, but to groups within it.

```
-- Compute average salary by role:
SELECT role,AVG(salary) FROM employees
GROUP BY role;
-- How many people are in each division?
-- (sorting results by division name)
SELECT division,COUNT(*) FROM employees
GROUP BY division
ORDER BY division;
```

Every selected column except the one(s) you're aggregating must appear in your GROUP BY.

Filtering Groups

To filter by a condition (like with WHERE but now applied to each group) use the HAVING keyword:

```
-- Same as above, but omit tiny divisions:
SELECT division,COUNT(*) FROM employees
GROUP BY division
HAVING COUNT(*) >= 10
ORDER BY division;
```

Statistical Thinking in Python 1/2

Graphical EDA

Plotting a histogram of your data:

```
import matplotlib.pyplot as plt
plt.hist( df['column of interest'] )
plt.xlabel( 'column name (units)' )
plt.ylabel( 'number of [fill in]' )
plt.show()
```

To change the y axis to probabilities:

```
plt.hist( df['column of interest'], normed=True )
```

Sometimes there is a sensible choice of where to place bin boundaries, based on the meaning of the x axis. Example:

```
plt.hist( df['column of percentages'],
          bins=[0,10,20,30,40,50,60,70,80,90,100] )
```

Change default plot styling to Seaborn:

```
import seaborn as sns
sns.set()
# then do plotting afterwards
```

If your data has observations as rows and features as columns, with two features of interest in columns A and B, you can create a "bee swarm plot" as follows.

```
# assuming your dataframe is called df:
sns.swarmplot( x='A', y='B', data=df )
plt.xlabel( 'explain column A' )
plt.ylabel( 'explain column B' )
plt.show()
```

Box plots are very similar, but using `boxplot`:

```
sns.boxplot( x='A', y='B', data=df )
# then label axes and show plot as above
```

To show a data's distribution as an Empirical Cumulative Distribution Function plot:

```
# the data must be sorted from lowest to highest:
x = np.sort( df['column of interest'] )
# the y values must count evenly from 0% to 100%:
y = np.arange( 1, len(x)+1 ) / len(x)
# then create and show the plot:
plt.plot( x, y, marker='.', linestyle='none' )
plt.xlabel( 'explain column of interest' )
plt.ylabel( 'ECDF' )
plt.margins( 0.02 )  # 2% margin all around
plt.show()
```

Multiple ECDFs on one plot:

```
# prepare the data as before, but now repeatedly:
# (this could be abstracted into a function)
x = np.sort( df['column 1'] )
y = np.arange( 1, len(x)+1 ) / len(x)
plt.plot( x, y, marker='.', linestyle='none' )
x = np.sort( df['column 2'] )
y = np.arange( 1, len(x)+1 ) / len(x)
# and so on, if there were other columns to plot
plt.plot( x, y, marker='.', linestyle='none' )
# and so on if there are more data series
plt.legend( ('explain x1', 'explain x2'),
            loc='lower right')
# then label axes and show plot as usual (not shown)
```

Scatter plot from two quantitative columns in your data:

```
plt.plot( df['column 1'], df['column 2'],
          marker='.', linestyle='none' )
# then label axes and show plot as usual
```

Summary statistics

The mean is the center of mass of the data:

```
np.mean( df['column name'] )
np.mean( series )
```

The median is the 50th percentile, or midpoint of the data:

```
np.median( df['column name'] )
np.median( series )
```

Or you can compute any percentile:

```
quartiles = np.percentile(
    df['column name'], [ 25, 50, 75 ] )
iqr = quartiles[2] - quartiles[0]
```

To overlay percentiles on an ECDF (here using red diamonds):

```
# (assuming you've plotted the ECDF already)
pct = [ 25, 50, 75 ]  # or your chosen percentiles
xs = np.percentile( df['column'], pct )
plt.plot( xs, pct/100,
          marker='D', color='red', linestyle='none' )
# then plt.show() when you're ready
```

Variance is a measure of the spread of the data, the average squared distance from the mean:

```
np.var( df['column name'] )
np.var( series )
```

Standard deviation is the square root of the variance:

```
np.std( df['column name'] )
np.std( series )
```

Covariance measures correlation between two data series.

```
# get a covariance matrix on of these ways:
M = np.cov( df['column 1'], df['column 2'] )
M = np.cov( series1, series2 )
# extract the value you care about, for example:
covariance = M[0,1]
```

The Pearson correlation coefficient normalizes this to $[-1, 1]$:

```
# same as covariance, but using np.corrcoef instead:
np.corrcoef( series1, series2 )
```

Random number generation and simulation

Recall these random number generation basics:

```
np.random.seed( my_int )
np.random.random()       # uniform random in [0,1)
np.random.randint(a,b)   # uniform random in a:b
```

Sampling many times from some distribution:

```
# if the distribution is built into numpy:
results = np.random.random( size=1000 )
# if the distribution is not built into numpy:
simulation_size = 1000    # or any number
results = np.empty( simulation_size )
for i in range( simulation_size ):
    # generate a random number here, however you
    # need to; here is a random example:
    value = 1 - np.random.random() ** 2
    # store it in the list of results:
    results[i] = value
```

Bernoulli trials with probability p:

```
success = np.random.random() < p     # one trial
num_successes = np.random.binomial(
    num_trials, p )                  # many trials
# 1000 experiments, each containing 20 trials:
results = np.random.binomial( 20, p, size=1000 )
```

Poisson distribution:

```
samples = np.random.poisson(
    mean_arrival_rate, size=1000 )  # size optional
```

Normal (Gaussian) distribution:

```
samples = np.random.normal(
    mean, std, size=1000 )          # size optional
```

Exponential distribution (time between events in a Poisson distribution):

```
samples = np.random.exponential(
    mean_waiting_time, size=1000 )  # size optional
```

You can take an array of numbers generated by simulation and plot it as an ECDF, as covered in the Graphical EDA section.

Introduction to Data Visualization with Python

Customizing Plots

Break a plot into an $n \times m$ grid of subplots as follows: (This is preferable to `plt.axes`, not covered here.)

```
# create the grid and begin working on subplot #1:
plt.subplot( n, m, 1 )
plt.plot( x, y )          # this will create plot #1
plt.title( '...' )        # title for plot #1
plt.xlabel( '...' )       # ...and any other options
# keep the same grid and now work on subplot #2:
plt.subplot( n, m, 2 )
# any plot commands here for plot 2,
# continuing for any further subplots, ending with:
plt.tight_layout()
plt.show()
```

Tweak the limits on the axes as follows:

```
plt.xlim( [ min, max ] )  # set x axis limits
plt.ylim( [ min, max ] )  # set y axis limits
plt.axis( [ xmin, xmax, ymin, ymax ] ) # both
```

To add a legend to a plot:

```
# when plotting series, give each a label,
# which will identify it in the legend:
plt.plot( x1, y1, label='first series' )
plt.plot( x2, y2, label='second series' )
plt.plot( x3, y3, label='third series' )
# then add the legend:
plt.legend( loc='upper right' )
# then show the plot as usual
```

To annotate a figure:

```
# add text at some point (here, (10,15)):
plt.annotate( 'text', xy=(10,15) )
# add text at (10,15) with an arrow to (5,15):
plt.annotate( 'text', xytext=(10,15), xy=(5,15),
              arrowprops={ 'color' : 'red' } )
```

Change plot styles globally:

```
plt.style.available       # see list of styles
plt.style.use( 'style' )  # choose one
```

Plotting two-dimensional arrays

To plot a bivariate function using colors:

```
# choose the sampling points in both axes:
u = np.linspace( xmin, xmax, num_xpoints )
v = np.linspace( ymin, ymax, num_ypoints )
# create pairs from these axes:
x, y = np.meshgrid( u, v )
# broadcast a function across those points:
z = x**2 - y**2
# plot it in color:
plt.pcolor( x, y, z )
plt.colorbar()        # optional but helpful
plt.axis( 'tight' )   # remove whitespace
plt.show()
# optionally, the pcolor call can take a color
# map parameter, one of a host of palettes, e.g.:
plt.pcolor( x, y, z, cmap='autumn' )
```

To make a contour plot instead of a color map plot:

```
# replace the pcolor line with this:
plt.contour( x, y, z )
plt.contour( x, y, z, 50 )  # choose num. contours
plt.contourf( x, y, z )     # fill the contours
```

To make a bivariate histogram:

```
# for rectangular bins:
plt.hist2d( x, y, bins=(xbins,ybins) )
plt.colorbar()
# with optional x and y ranges:
plt.hist2d( x, y, bins=(xbins,ybins),
            range=((xmin,xmax),(ymin,ymax)) )
# for hexagonal bins:
plt.hexbin( x, y,
            gridsize=(num_x_hexes,num_y_hexes) )
# with optional x and y ranges:
plt.hexbin( x, y,
            gridsize=(num_x_hexes,num_y_hexes),
            extent=(xmin,xmax,ymin,ymax) )
```

To display an image from a file:

```
image = plt.imread( 'filename.png' )
plt.imshow( image )
plt.axis( 'off' )            # axes don't apply here
plt.show()
# to collapse a color image to grayscale:
gray_img = image.mean( axis=2 )
plt.imshow( gray_img, cmap='gray' )
# to alter the aspect ratio:
plt.imshow( gray_img, aspect=height/width )
```

The Seaborn library

Plotting a linear regression line:

```
import seaborn as sns
sns.lmplot( x='col 1', y='col 2', data=df )
```

Plotting a linear regression line:

```
import seaborn as sns
sns.lmplot( x='col 1', y='col 2', data=df )
plt.show()
# and the corresponding residual plot:
sns.residplot( x='col 1', y='col 2', data=df,
               color='red' )  # color optional
```

Plotting a polynomial regression curve of order n:

```
sns.regplot( x='col 1', y='col 2', data=df,
             order=n )
# this will include a scatter plot, but if you've
# already done one, you can omit redoing it:
sns.regplot( x='col 1', y='col 2', data=df,
             order=n, scatter=None )
```

To do multiple regression plots for each value of a categorical variable in column X, distinguished by color:

```
sns.lmplot( x='col 1', y='col 2', data=df,
            hue='column X', palette='Set1' )
# (many other options exist for palette)
```

Now separate plots into columns, rather than all on one plot:

```
sns.lmplot( x='col 1', y='col 2', data=df,
            row='column X' )
sns.lmplot( x='col 1', y='col 2', data=df,
            col='column X' )
```

Strip plots can visualize univariate distributions, especially useful when broken into categories:

```
sns.stripplot( y='data column', x='category column',
               data=df )
# to add jitter to spread data out a bit in x:
sns.stripplot( y='data column', x='category column',
               data=df, size=4, jitter=True )
```

Swarm plots, covered earlier, are very similar, but can also have colors in them to distinguish categorical variables:

```
sns.swarmplot( y='data column', x='category 1',
               hue='category 2', data=df )
# and you can also change the orientation:
sns.swarmplot( y='category 1', x='data column',
               hue='category 2', data=df,
               orient='h' )
```

Violin plots make curves using kernel density estimation:

```
sns.violinplot( y='data column', x='category 1',
                hue='category 2', data=df )
```

Joint plots for visualizing a relationship between two variables:

```
sns.jointplot( x='col 1', y='col 2', data=df )
# and to add smoothing using KDE:
sns.jointplot( x='col 1', y='col 2', data=df,
               kind='kde' )
# other kind options: reg, resid, hex
```

Scatter plots and histograms for all numerical columns in `df`:

```
sns.pairplot( df )            # no grouping/coloring
sns.pairplot( df, hue='A' )   # color by column A
```

Visualize a covariance matrix with a heatmap:

```
M = np.cov( df[['col 1','col 2','col3']], # or more
            rowvar=False )    # vars are in columns
# (or you can use np.corrcoef to normalize np.cov)
sns.heatmap( M )
```

Streamlined Data Ingestion with pandas

Reading flat files

Any file whose rows are on separate lines and whose entries are separated by some delimeter can be read with the same `read_csv` function we've already seen.

```
df = pd.read_csv( "my_csv_file.csv" )    # commas
df = pd.read_csv( "my_tabbed_file.tsv",
                  sep="\t" )             # tabs
```

If you only need some of the data, you can save space:

```
# choose just some columns:
df = pd.read_csv( "my_csv_file.csv", usecols=[
    "use", "only", "these", "columns" ] )
# can also give a list of column indices,
# or a function that filters column names

# choose just the first 100 rows:
df1 = pd.read_csv( "my_csv_file.csv", nrows=100 )
# choose just rows 1001 to 1100,
# re-using the column header from df1:
df2 = pd.read_csv( "my_csv_file.csv",
                   nrows=100, skiprows=1000,
                   header=None,       # skipped it
                   names=list(df1) )  # re-use
```

If pandas is guessing a column's data type incorrectly, you can specify it manually:

```
df = pd.read_csv( "my_geographic_data.csv",
                  dtype={"zipcode":str,
                         "isemployed":bool} )
# to correctly handle bool types:
df = pd.read_csv( "my_geographic_data.csv",
                  dtype={"zipcode":str,
                         "isemployed":bool},
                  true_values=["Yes"],
                  no_values=["No"] )
# note: missing values get coded as True!
# (pandas understands True, False, 0, and 1)
```

If some lines in a file are corrupt, you can ask `read_csv` to skip them and just warn you, importing everything else:

```
df = pd.read_csv( "maybe_corrupt_lines.csv",
                  error_bad_lines=False,
                  warn_bad_lines=True )
```

Reading spreadsheets

If the spreadsheet is a single table of data without formatting:

```
df = pd.read_excel( "my_table.xlsx" )
# nrows, skiprows, usecols, work as before, plus:
df = pd.read_excel( "my_table.xlsx",
                    usecols="C:J,L" )   # excel style
```

If a file contains multiple sheets, choose one by name or index:

```
df = pd.read_excel( "my_workbook.xlsx",
                    sheet_name="budget" )
df = pd.read_excel( "my_workbook.xlsx",
                    sheet_name=3 )
# (the default is the first sheet, index 0)
```

Or load all sheets into an ordered dictionary mapping sheet names to DataFrames:

```
dfs = pd.read_excel( "my_workbook.xlsx",
                     sheet_name=None )
```

Advanced methods of date/time parsing:

```
# standard, as seen before:
df = pd.read_excel( "file.xlsx",
                    parse_dates=True )
# just some cols, in standard date/time format:
df = pd.read_excel( "file.xlsx",
                    parse_dates=["col1","col2"] )
# what if a date/time pair is split over 2 cols?
df = pd.read_excel( "file.xlsx",
                    parse_dates=[
                        "datetime1",
                        ["date2","time2"]
                    ] )
# what if we want to control column names?
df = pd.read_excel( "file.xlsx",
                    parse_dates={
                        "name1":"datetime1",
                        "name2":["date2","time2"]
                    } )
# for nonstandard formats, do post-processing,
# using a strftime format string, like this example:
df["col"] = pd.to_datetime( df["col"],
    format="%m%d%Y %H:%M:%S" )
```

Reading databases

In SQLite, databases are .db files:

```
# prepare to connect to the database:
from sqlalchemy import create_engine
engine = create_engine( "sqlite:///filename.db" )
# fetch a table:
df = pd.read_sql( "table name", engine )
# or run any kind of SQL query:
df = pd.read_sql( "PUT QUERY CODE HERE", engine )
# if the query code is big:
query = """PUT YOUR SQL CODE
           HERE ON AS MANY LINES
           AS YOU LIKE;"""
df = pd.read_sql( query, engine )
# or get a list of tables:
print( engine.table_names() )
```

Reading JSON

From a file or string:

```
# from a file:
df = pd.read_json( "filename.json" )
# from a string:
df = pd.read_json( string_containing_json )
# can specify dtype, as with read_csv:
df = pd.read_json( "filename.json",
                   dtype={"zipcode":str} )
# also see pandas documentation for JSON "orient":
# records, columns, index, values, or split
```

From the web with an API:

```
import requests
response = requests.get(
    "http://your.api.com/goes/here",
    headers = {
        "dictionary" : "with things like",
        "username" : "or API key"
    },
    params = {
        "dictionary" : "with options as",
        "required by" : "the API docs"
    } )
data = response.json()  # ignore metadata
result = pd.DataFrame( data )
# or possibly some part of the data, like:
result = pd.DataFrame( data["some key"] )
# (you must inspect it to know)
```

If the JSON has nested objects, you can flatten:

```
from pandas.io.json import json_normalize
# instead of this line:
result = pd.DataFrame( data["maybe a column"] )
# do this:
result = json_normalize( data["maybe a column"],
                         sep="_" )
# (if there is deep nesting, see the record_path,
# meta, and meta_prefix options)
```