My study notes from the book "Python for Data Analysis, by Wes McKinney"

Link for the book: https://wesmckinney.com/book/ (free access)
Repository that contains the book materials: https://github.com/wesm/pydata-book

Chapter 1 to 3: Basics

Note 1: Referencing an existing object in Python

When assigning an object to a new variable, you are copying the reference to the object. Not a copy of the existing object.
Example:
a = [1, 3, 5]
a = b
a.append(7)
print(b)
output >> [1, 3, 5, 7]

Note 2: Modifying tuples pt1

If an object inside a tuple is mutable you can modify it.
Example:
tup = tuple(['foo', [1, 2], True])
tup[1].append(3)
print(tup)
output >> ('foo', [1, 2,3], True)

Note 3: Modifying tuples pt2

You can alongate tuples.
Example:
tup1 = (1,2,3)
tup2 = (4,5)
tup3 = tup1 + tup2
tup4 = tup2 3
print(tup3)
output >> (1,2,3,4,5)

print(tup3)
output >> (4,5,4,5,4,5)*

Note 4: Use collections.deque

Insert is computationally expensive compared with append. When in need to insert elements at the start and end of a sequence use: **collections.deque**, a double-ended queue, optimized for this purpose.

Note 5: is in {} >> [ ]

Checking whether a list contains a value is a lot slower than doing so with dictionaries and sets as Python makes a linear scan across the values of the list.

Note 6: str.extend() >> str + str

Concatenation by addition is a comparatively expensive operation since a new list must be created and the objects copied over. Using extend to append elements to an existing list is preferable.
Example:
*everything = []
for chunk in list_of_lists:
everything.extend(chunk)

is faster than the concatenative alternative:

everything = []
for chunk in list_of_lists:
everything = everything + chunk

Note 7: types of nested list comprehensions

You can have 2 types of nesting in list comprehensions.
Example:
type1)
In: [x for tup in some_tuples for x in tup]
Out: [1, 2, 3, 4, 5, 6, 7, 8, 9]
type2)
In: [[x for x in tup] for tup in some_tuples]
Out: [[1, 2,3], [4, 5, 6], [7, 8, 9]]

Chapter 4: NumPy

Note 1: Built-in lists x Numpy Array's

An important distinction between Python's built-in lists and numpy array's is that the **slices are views on the original array.**
This means that the data is not copied: **any modifications to the view will be reflected in the source array.**
If you really want a copy, you need to: *array_copy = array[4:6].copy()*
ps: pandas also work like this

Note 2: How to select elements in 2+ dimension arrays?

In a 2 or more dimension array you can select elements either using commas or using new brackets. (rows = 1st dim, col = 2st dim, depth = 3st dim, time = 4st dim? kekw (joke))
Example:
array [0] [2] == array [0 , 2]

Note 3: FANCY INDEXING

To select a subset of the rows in a particular order, pass a list of integers specifying the desired order:

Example:
arr = array([[0,0,0,0], [1,1,1,1], [2,2,2,2], [3,3,3,3], [4,4,4,4], [5,5,5,5], [6,6,6,6], [7,7,7,7]])

print(arr[[4, 3, 0, 6]])
output >> array([[4,4,4,4], [3,3,3,3], [0,0,0,0], [6,6,6,6]])

BUT... IF YOU PASS MULTIPLE INDEX ARRAYS...
something slightly different will happen; you'll get a one-dimensional array of elements corresponding to each tuple of indices:

Example:
arr = np.arange(32).reshape((8,4))
arr = array([[0,1,2,3], [4,5,6,7], [8,9,10,11], [12,13,14,15], [16,17,18,19], [20,21,22,23], [24,25,26,27], [28,29,30,31]])
print(arr[[1, 5, 7, 2], [0, 3, 1, 2]])
output >> array([4, 23, 29, 10])

The result of fancy indexing with as many integer arrays as there are axes is always one-dimensional.
Also: unlike slicing, fancy indexing always copies the data into a new array. Not a view.

If you want to SLICE the array you should do:
print(arr[[1, 5, 7, 2]] [: , [0, 3, 1, 2]])
output >> array([[4, 7, 5, 6], [20, 23, 21, 22], [28, 31, 29, 30], [8, 11, 9, 10]])

Note 4: Universal Functions == ufuncs

A ufunc is a function that performs element-wise operations on data in ndarrays.
They are fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

Unary ufuncs: perform element-wise transformations
Examples: np.sqrt or np.exp

Binary ufuncs: take two arrays and return a single array as the result
Examples: np.add or np.max

PS: While not common, a ufunc can return multiple arrays.
Example: np.modf returns the fractional and integral parts of a floating-point array.
remainder, whole_part = np.modf(array)

Note 5: Matrix multiplication

1. Matrix multiplication is possible when the number of columns in the first matrix is equal to the number of rows in the second matrix.
2. To determine the shape of the resulting matrix, take the number of rows from the first one and the number of columns from the second one.
3. To determine the values in the resulting matrix, sum up the products of corresponding elements in each row of the first matrix with each column of the second matrix.

Note 6: Vectorization

Vectorization is replacing explicit loops with array expressions. Vectorized array operations will usually be significantly faster than their pure Python equivalents.

PS:
The term vectorization is used to describe some other computer science concepts.

Note 7: if-elif-else vectorized

the method numpy.where expresses the conditional logic iin a vectorized way.
How to use: *np.where(condition, array1,array2)*

ps: the 2th and 3rd arguments don’t need to be arrays; one or both of them can be scalars.

Note 8: aggregations and accumulations vectorized

Aggregations (sometimes called reductions) like *sum, mean, and std* can be evoked by calling the array instance method or using the top-level NumPy function.
Example:
*arr.**mean**()* or *np.**mean**(**arr**)*

They can take an optional *axis* argument that computes the statistic over the given axis, resulting in an array with one less dimension.
Example:
*arr.**mean**(**axis=1**)* compute mean across the columns
*arr.**sum**(**axis=0**)* compute sum down the rows.

Accumulations like *cumsum* return an array of the same size but with the partial aggregates computed along the indicated axis.
Example:
_arr.**cumsum(axis=0)** computes the cumulative sum along the rows
arr.**cumsum(axis=1)** computes the sums along the columns

Note 9: numpy.sort and built-in .sorted

The top-level method ***numpy.sort*** returns a sorted copy of an array (like the Python built-in function sorted) instead of modifying the array in place.

Note 10: Matrix inversion

Matrix inversion is the process of finding a matrix B such that when it is multiplied by the original matrix A, it yields the identity matrix.

The identity matrix is a square matrix that, when multiplied by another matrix, leaves the other matrix unchanged. It plays a similar role in matrix operations as the number 1 does in scalar multiplication.

Here's what makes the identity matrix special:

- **Square Matrix:** The identity matrix is always square, meaning it has the same number of rows and columns.
- **Diagonal Ones:** All the elements on the main diagonal (from the top left to the bottom right) are 1.
- **Off-diagonal Zeros:** All other elements (not on the main diagonal) are 0.

Chapter 5: Pandas

Note 1: Series

You can get the array representation and index object of the Series via its array and index attributes.

Example:
obj2 = pd.Series([4, 7, -5, 3], index=["d", "b", "a", "c"])
print(obj2.index)
output >> Index(["d", "b", "a", "c"], dtype='object')

Both the Series object itself and its index have a name attribute

In: obj4.name = "population"
In: obj4.index.name = "state"

In contrast with NumPy arrays you can use labels in the index when selecting a single value or a set of values.

Example:
print(obj2[["c", "a", "d"]])
output >>
d 6
b 7
c 3
dtype: int64

You can create a Series from passing a dictionary.

obj3 = pd.Series(dict_here)

And you can convert back to a dictionary with the *to_dict* method:

dict_here = obj3.to_dict()

Note 2: DataFrames

Transposing discards the column data types if the columns do not all have the same data type.

In this case, transposing and then transposing back may lose the previous type information. The columns become arrays of pure Python objects in this case.

Index objects are immutable and thus can’t be modified by the user.

.reindex method create a new object with the values rearranged to align with the new index.

obj = pd.Series([4.5, 7.2, -5.3, 3.6], index = ["d", "b", "a", "c"])
obj2 = obj.reindex(["a", "b", "c", "d", "e"])

In: obj
Out:
d 4.5
b 7.2
a -5.3
c 3.6
dtype: float64

In: obj2
Out:
a -5.3
b 7.2
c 3.6
d 4.5
e NaN
dtype: float64

.loc and .iloc DataFrames slicing is inclusive

You can also slice with labels, but it works differently from normal Python slicing in that the endpoint is inclusive.
In: obj2.loc["b":"c"]
Out:
b 2
c 3
dtype: int64

Avoid chained indexing when doing assignments and try to use a single loc operation:

Trying this: data.loc[data.three == 5]["three"] = 6
will generate a *SettingWithCopyWarning*, which warns you that you are trying to modify a temporary value.

instead, do this: data.loc[data.three == 5, "three"] = 6.

Adding objects in Pandas: Standard "+" operator

Results in the union of the index pairs and the overlapping values addition. The non overlapping values will become missing values (NaN).

```
 Example:**
In: s1
Out:
a 7.3
c -2.5
d 3.4
e 1.5
dtype: float64

In : s2
Out:
a -2.1
c 3.6
e - 1.5
f 4.0
g 3.1
dtype: float64

In: s1 + s2
Out:
a 5.2
c 1.1
d NaN
e 0.0
f NaN
g NaN
dtype: float64
```

Adding objects in Pandas: fill_value

Using the *add* method on df1, I pass 2 arguments: df2 and *fill_value*. This will substitute the passed value for any missing values in the operation.

DataFrame's *.apply* method

Will broadcast a function on one-dimensional arrays to each column or row in the DataFrame. The default is applying to the columns, one by one.

```
 **Example:**
In: def func1(x):
    return x.max() - x.min()
In: DataFrameX
Out:
b d e
Utah -0.204708 0.478943 -0.519439
Ohio -0.555730 1.965781 1.393406
Texas 0.092908 0.281746 0.769023
Oregon 1.246435 1.007189 -1.296221

In: DataFrameX.apply(func1)
Out:
b 1.802165
d 1.684034
e 2.689627
dtype: float64
```

BUT...
If you pass *axis="columns"* to the *apply* method, the function will be invoked once per row instead. The book says that a helpful way to think about this is as “apply across the columns”. Didn't help me.

```
**Example:**
In: DataFrameX.apply(func1)
Out:
Utah  0.998382
Ohio 2.521511
Texas 0.676115
Oregon 2.542656
```

ps: Many of the most common array statistics (like sum _and mean) are DataFrame methods, so think of using apply when using a function you defined.

ps: *The function passed to apply need not return a scalar value; it can also return a Series with multiple values.*

```
Example:
In:
def func2(x):
```

```
    return pd.Series(\[x.min(), x.max()], index=\["min", "max"])

DataFrameX.apply(func2)

Out:
b d e
min -0.555730 0.281746 -1.296221
max 1.246435 1.965781 1.393406
```

DataFrame's *.applymap* method

Element-wise Python functions can be used with .applymap. If you wanted to compute a formatted string from each floating-point value in a DataFrame, You could:

```
Example:
In:
def format_DataFrame(x):
    return f"{x:.2f}"

DataFrameX.applymap(format_DataFrame)

Out:
b d e
Utah -0.20 0.48 -0.52
Ohio -0.56 1.97 1.39
Texas 0.09 0.28 0.77
Oregon 1.25 1.01 -1.30
```

DataFrame's sorting

.sort_index() method:

```
The .sort_index() method sorts a axis of the DataFrame lexicographically. It
can be either axis and in either direction, but rows and asceding are the
default.
Example:
DataF.sort_index() >>> sorts rows from A to Z
DataF.sort_index(axis='columns', asceding=False) sorts columns from Z to A.
```

.sort_values() method:

1. To sort a Series by its values, use its .sort_values() method.
   Any missing values are sorted to the end of the Series by default.
   Example:

```
In:
obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
print(obj.sort_values())

Out:
4 -3.0
5 2.0
0 4.0
2 7.0
1 NaN
3 NaN
dtype: float64
```

   ps: Missing values can be sorted to the start instead by
   using the na_position argument (obj.sort_values(na_position="first")).
2. To use the data in one or more columns as the sort keys, pass one or more column names to .sort_values():
   Example:

```
DataF.sort_values(["a", "b"])
```

3. Ranking values assigns ranks from 1 through the number of valid data points in an array, starting from the lowest value or the first in order.
   Example:

```
In:
obj = pd.Series([7, -5, 7, 4, 2, 0, 4]
obj.rank()

Out:
0 6.5
1 1.0
2 6.5
3 4.5
4 3.0
```

```
5 2.0
6 4.5
dtype: float64

To rank according to order use obj.rank(method="first")
```

.corr() and *.corrwith()* methods:

1. *.corr()* computes the correlation in **2 specified Series or the entire DataFrame**.
2. *.corrwith()* computes the correlation in **1 Series against the entire DataFrame**.

.get_indexer() method:
Compute integer indices for each value in an array into another array of distinct values; helpful for data alignment and join-type operations.

```
Example:
   In:
   to_match = pd.Series(["c", "a", "b", "b", "c", "a"])
   unique_vals = pd.Series(["c", "b", "a"])
   indices = pd.Index(unique_vals).get_indexer(to_match)
   print(indices)

   Out:
   array([0, 2, 1, 1, 0, 2])
```

.agg() and .groupby methods:
To create new columns with .agg functions you need to pass tuples of 2 values to a new variable.
The tuple will contain the column to aggregate and the func to use and the new variable will be the name of the new column.

```
new_DataFrame = unemployment.groupby("continent")
.agg(mean_rate_2021=("2021","mean"))
```

Chapter 6: Data Loading, Storage, and File Formats

Note 1: HDF5

```
HDF5 stands for hierarchical data format. Each HDF5 file can store multiple
datasets and supporting metadata. It enables data with repeated patterns to
be stored more efficiently. HDF5 can be a good choice for working with
datasets that don’t fit into memory, as you can efficiently read and write
small sections of much larger arrays.
```

HDF5 is best suited for write-once, read-many datasets.

While data can be added to a file at any time, if multiple writers do so simultaneously, the file can become corrupted.

HDF5 in pandas

Pandas simplifies storing Series and DataFrame objects as HDF5 files. The HDFStore class works like a dictionary and handles the low-level details.
Example:

```
In: frame = pd.DataFrame({"a": np.random.standard_normal(100)})
In: store = pdmydata.h5("examples/mydata.h5")
In: store\["obj1"] = frame
In: store\["obj1_col"] = frame\["a"]
In: store
Out:
<class 'pandas.io.pytables.HDFStore'>
File path: examples/mydata.h5
```

HDFStore supports 2 storage schemas: "fixed" and "table"

the default is "fixed". The "table" format is generally slower, but it supports query operations using a special syntax.
Example:

```
store.put("obj2", frame, format="table")
store.select("obj2", where=\["index >= 10 and index <= 15"])
```

Chapter 7: Data Cleaning and Preparation

Note 1: .dropna() method

Practical example:

```
![[Pasted image 20240304091933.png]]
                ![[Pasted image 20240304092822.png]]
                ![[Pasted image 20240304092916.png]]
                ![[Pasted image 20240304092933.png]]
                _inplace=True modifies the original DataFrame
                inplace=False creates a new one_

                (slide from a different source)
```

On Series, it returns the Series with only the nonnull data and index values.

```
In: data = pd.Series(\[1, np.nan, 3.5, np.nan, 7])
dtype4]: data.dropna()

Out:
0 1.0
2 3.5
4 7.0
dtype: float64
```

On a DataFrames, by default it drops any row containing a missing value.

```
In: data
Out:
0 1 2
0 1.0 6.5 3.0
1 1.0 NaN NaN
2 NaN NaN NaN
3 NaN 6.5 3.0

In: data.dropna()
Out:
0 1 2
0 1.0 6.5 3.0
```

Passing how="all" will drop only rows that are all NA.

In: data.dropna(how="all")

Out:
0 1 2
0 1.0 6.5 3.0
1 1.0 NaN NaN
3 NaN 6.5 3.0

To keep only rows containing at most a certain number of missing observations. Pass the *thresh argument* (tresh=7).

```
In: df.dropna(thresh=2)
Out:
0 1 2
2 0.092908 NaN 0.769023
3 1.246435 NaN -1.296221
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741
```

To drop columns in the same way, pass axis="columns".

In: data
Out:
0 1 2 4
0 1.0 6.5 3.0 NaN
1 1.0 NaN NaN NaN
2 NaN NaN NaN NaN
3 NaN 6.5 3.0 NaN

```
In: data.dropna(axis="columns", how="all")
Out:
0 1 2
0 1.0 6.5 3.0
1 1.0 NaN NaN
2 NaN NaN NaN
3 NaN 6.5 3.0
```

The *.drop_duplicates()* method by default keep the first observed value combination. Passing keep="last" will return the last one.

In: data.drop_duplicates(["k1", "k2"], keep="last")

The data.replace method is distinct from data.str.replace, which performs element-wise string substitution.

Note 2: fillna() method

Pratical example:

Imputing a summary statistic

```
cols_with_missing_values = salaries.columns[salaries.isna().sum() > 0]
print(cols_with_missing_values)
```

```
for col in cols_with_missing_values[:-1]:
    salaries[col].fillna(salaries[col].mode()[0])
```

Imputing by sub-group

```
salaries_dict = salaries.groupby("Experience")["Salary_USD"].median().to_dict()
print(salaries_dict)
```

```
{'Entry': 55380.0, 'Executive': 135439.0, 'Mid': 74173.5, 'Senior': 128903.0}
```

(slide from a different source)

The *.fillna()* method in pandas replaces NaN values with a specified value or method. Key options include:
Value: Replace NaNs with a given value, or use a dictionary/Series for different values per column.
Method: Use `ffill` to forward-fill or `bfill` to backward-fill NaNs.
Axis: Choose axis (`0` for rows, `1` for columns) for filling.
Inplace: Apply changes directly to the original object with `inplace=True`.
Limit: Restrict the number of consecutive NaNs to fill.

Note 3: .cut() method

To do discretization and binning, you have to use pandas.cut(). With a list of values to use as edges of the bins or with a single integer representing the total amount of bins.

```
In: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
In: bins = [18, 25, 35, 60, 100]
In: age_categories = pd.cut(ages, bins)
In: age_categories
Out: [(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60,
100],(35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64, right]): [(18, 25] < (25, 35] < (35, 60] <
(60, 100]]
```

In the string representation of the interval, the parenthesis means "side is open (exclusive)", and the square bracket means "side closed (inclusive)". You can change which side is closed by passing the parameter right=False

```
In [85]: pd.cut(ages, bins, right=False)
Out[85]:
[[18, 25), [18, 25), [25, 35), [25, 35), [18, 25), ...,
[25, 35), [60, 100), [35, 60), [35, 60), [25, 35)]
Length: 12
Categories (4, interval[int64, left]): [[18, 25) < [25, 35) < [35, 60)
< [60, 100)]
```

You can override the default interval-based bin labeling with the labels parameter.

```
In: group_names = ["Youth", "YoungAdult", "MiddleAged", "Senior"]
In: pd.cut(ages, bins, labels=group_names)
Out:
['Youth', 'Youth', 'Youth', 'YoungAdult', 'Youth', ...,
'YoungAdult', 'Senior', '
MiddleAged', 'MiddleAged', 'YoungAdult']
Length: 12
Categories (4, object): ['Youth' < 'YoungAdult' < 'MiddleAged' < 'Senior']
```

The pd.value_counts(categories) method are the bin counts for the result of pandas.cut.

To obtain roughly equally sized bins, you can use the *.qcut()* method, it bins the data based on sample quantiles.

```
In: data = np.random.standard_normal(1000)
In: quartiles = pd.qcut(data, 4, precision=2)
In: quartiles
Out:
[(-0.026, 0.62], (0.62, 3.93], (-0.68, -0.026], (0.62, 3.93],
(-0.026, 0.62], ...(-0.68, -0.026], (-0.68, -0.026],
(-2.96, -0.68], (0.62, 3.93], (-0.68, -0.026]]
Length: 1000
Categories (4, interval[float64, right]): [(-2.96, -0.68] < (-0.68, -0.026]
< (-0.026, 0.62] < (0.62, 3.93]]
In: pd.value_counts(quartiles)
Out:
(-2.96, -0.68] 250
(-0.68, -0.026] 250
(-0.026, 0.62] 250
(0.62, 3.93] 250
dtype: int64
```

aa

Note 4: Outliers

Outlier detection criteria:
A point beyond an inner fence on either side is considered a mild outlier.
A point beyond an outer fence is considered an extreme outlier.

```
lower inner fence: Q1 - 1.5 * IQ
upper inner fence: Q3 + 1.5 * IQ
lower outer fence: Q1 - 3 * IQ
upper outer fence: Q3 + 3 * IQ

 Q1 = 25th percentile]
 Q3 = 75th percentile]
 IQ = Q3 - Q1 (Interquantile Range)
```

To select the outlier values you can do something like:
Presume the outliers are values above 3 or below -3.
Example:

```
In: data[(data.abs() > 3).any(axis="columns")]
Out:
0 1 2 3
41 0.457246 -0.025907 -3.399312 -0.974657
60 1.951312 3.260383 0.963301 1.201206
136 0.508391 -0.196713 -3.745356 -1.520113
235 -0.242459 -3.056990 1.918403 -0.578828
258 0.682841 0.326045 0.425384 -3.428254
322 1.179227 -3.184377 1.369891 -1.074833
544 -3.548824 1.553205 -2.186301 1.277104
635 -0.578093 0.193299 1.397822 3.366626
782 -0.207434 3.525865 0.283070 0.544635
803 -3.645860 0.255475 -0.549574 -1.907459
```

The parentheses around data.abs() > 3 are necessary in order to call the any method on the result of the comparison operation.

To cap outliers you can do something like:
Presume the outliers are values above 3 or below -3.
Example:

```
data[data.abs() > 3] = np.sign(data) * 3
```

The statement np.sign(data) produces 1 and –1 values based on whether the values in data _are positive or negative.

Note 5: Permutation and Random Sampling

Permuting (randomly reordering) the rows or columns in a DataFrame is possible using the *numpy.random.permutation* function.

Calling permutation with the length of the axis you want to permute produces an array of integers indicating the new ordering.
Example:

```
In: df = pd.DataFrame(np.arange(5 * 7).reshape((5, 7)))
In: df
Out:
0 1 2 3 4 5 6
0 0 1 2 3 4 5 6
```

```
1 7 8 9 10 11 12 13
2 14 15 16 17 18 19 20
3 21 22 23 24 25 26 27
4 28 29 30 31 32 33 34
In: sampler = np.random.permutation(5)
In: sampler
Out: array([3, 1, 4, 2, 0])
```

That array can then be used in iloc-based indexing or the equivalent take function:
Example:

In [107]: df.take(sampler) or df.iloc[sampler]
Out[107]:
0 1 2 3 4 5 6
3 21 22 23 24 25 26 27
1 7 8 9 10 11 12 13
4 28 29 30 31 32 33 34
2 14 15 16 17 18 19 20
0 0 1 2 3 4 5 6

To select a random subset without replacement (the same row cannot appear twice),
you can use the *sample* method.

To generate a sample with replacement (to allow repeated choices) pass replace=True.

```
In: choices = pd.Series([5, 7, -1, 6, 4])
In: choices.sample(n=10, replace=True)
Out:
2 -1
0 5
3 6
1 7
4 4
0 5
4 4
0 5
4 4
4 4
dtype: int64
```

Note 6: Computing Indicator/Dummy Variables

A common used type of transformation for statistical modeling or machine learning applications is **converting a categorical variable into a dummy or indicator matrix.**

For a column of categorical variables that has k distinct values, you derive a matrix with k columns containing all 1s and 0s.

You can do this with the *.get_dummies()* method.

Example:

In: df
Out:
key data1
0 b 0
1 b 1
2 a 2
3 c 3
4 a 4
5 b 5

In: pd.get_dummies(df["key"])
Out:
a b c
0 0 1 0
1 0 1 0
2 1 0 0
3 0 0 1
4 1 0 0
5 0 1 0

To merge the dummies matrix with the other data *pandas.get_dummies* has a prefix argument.
Example:

```
In: df
Out:
key data1
 b 0
 b 1
 a 2
 c 3
```

```
 a 4
 b 5

In: dummies = pd.get_dummies(df["key"], prefix="key")
In: df_with_dummy = df[["data1"]].join(dummies)
In: df_with_dummy
Out:
data1 key_a key_b key_c
 0 0 1 0
 1 0 1 0
 2 1 0 0
 3 0 0 1
 4 1 0 0
 5 0 1 0
```

When a row belongs to multiple categories use a different approach to create the dummy variables.
The special Series method *str.get_dummies*() handles this scenario of multiple group membership encoded as a delimited string:

```
In [124]: dummies = movies["genres"].str.get_dummies("|")
In [125]: dummies.iloc[:10, :6]
Out[125]:
Action Adventure Animation Children's Comedy Crime
0 0 1 1 1 0
0 1 0 1 0 0
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
1 0 0 0 0 1
0 0 0 0 1 0
0 1 0 1 0 0
1 0 0 0 0 0
1 1 0 0 0 0
```

For larger data this method is not especially speedy. Its better to write a lower-level function that writes directly to a NumPy array, and then wrap the result in a DataFrame.

A useful recipe for statistical applications is to combine .get_dummies() with a discretization function like .cut().

Example:

```
In: values
Out:
array([0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645, 0.6532,
0.7489, 0.6536])
In: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
In: pd.get_dummies(pd.cut(values, bins))
Out:
(0.0, 0.2] (0.2, 0.4] (0.4, 0.6] (0.6, 0.8] (0.8, 1.0]
0 0 0 0 0 1
1 0 1 0 0 0
2 1 0 0 0 0
3 0 1 0 0 0
4 0 0 1 0 0
5 0 0 1 0 0
6 0 0 0 0 1
7 0 0 0 1 0
8 0 0 0 1 0
9 0 0 0 1 0
```

Note 7: Extension Data Types

Pandas has developed an extension type system allowing for new data types to be added even if they are not supported natively by NumPy. These new data types can be treated as first class alongside data coming from NumPy arrays.

Extension types can be passed to the Series astype() method as part of your data preparation process.

.astype("Int64") **(Capitalization is necessary, otherwise it will be a NumPy-based nonextension type)**
.astype("string")
.astype("boolean")

and others...

These extensions generally use much less memory and are frequently computationally more efficient for doing operations on large datasets. So it's good to know they exist.

Docs for extension types available: https://github.com/pandas-dev/pandas/blob/main/pandas/core/dtypes/dtypes.py

Note 8: RegEx

You can create a reusable regex object with re.compile() and you can then use the methods below in your object.

Example:

```
In: text = "foo bar\t baz \tqux"
In: regex = re.compile(r"\s+")
In: regex.split(text)
Out: ['foo', 'bar', 'baz', 'qux']
```

re.findall(), re.search() and re.match() methods.

Example:
re.findall()
returns all matches in a string
re.search()
returns only the first match
re.match()
More rigidly, match returns only matches at the beginning of the string.

re.sub() will return a new string with occurrences of the pattern replaced by a new string.

Example:

```
In: print(regex.sub("REDACTED", text))
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

The str.extract() method will return the captured groups of a regular expression as a DataFrame.

Example:

```
In: data.str.extract(pattern, flags=re.IGNORECASE)
Out:
0 1 2
Dave dave google com
Steve steve gmail com
Rob rob gmail com
Wes NaN NaN NaN
```

Note 9: Categorical type

In data warehousing, a best practice is to use so-called dimension tables containing the distinct values and storing the primary observations as integer keys referencing the dimension table. *This representation as integers is called the categorical representation, one-hot encoding, or dictionary-encoded representation.* The array of distinct values can be called the categories, dictionary, or levels of the data. The categorical representation can yield significant performance improvements.

This is the base of the categorical extension datatype in pandas.

The Categorical object has categories and codes attributes.

```
Example:
c = df['fruit'].astype('category')
In: c
Out:
0 apple
1 orange
2 apple
3 apple
4 apple
5 orange
6 apple
7 apple
Name: fruit, dtype: category
Categories (2, object): ['apple', 'orange']
```

The values for c are now an instance of pandas.Categorical, which you can access via the .array attribute:

```
Example:
In: c = c.array
In: c.categories
Out: Index(['apple', 'orange'], dtype='object')
In: c.codes
Out: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

A useful trick to get a mapping between codes and categories is:

```
In: dict(enumerate(c.categories))
Out: {0: 'apple', 1: 'orange'}
```

If you have obtained categorical encoded data from another source, you can use the .from_codes() constructor.

Example:

```
In: categories = ['foo', 'bar', 'baz']
In: codes = [0, 1, 2, 0, 0, 1]
In: my_cats_2 = pd.Categorical.from_codes(codes, categories)
In: my_cats_2
Out:
['foo', 'bar', 'baz', 'foo', 'foo', 'bar']
Categories (3, object): ['foo', 'bar', 'baz']
```

When using from_codes or any of the other constructors, you can indicate that the categories have a meaningful ordering.
Example:

```
In: ordered_cat = pd.Categorical.from_codes(codes, categories, ordered=True)
In: ordered_catbazt
Out:
['foo', 'bar', 'baz', 'foo', 'foo', 'bar']
Categories (3, object): ['foo' < 'bar' < 'baz']
```

.remove_unused_categories() method

After filtering large DataFrames, some categories may no longer appear in the data. To optimize for memory savings and performance, use the .remove_unused_categories() method to trim

unobserved categories.
Example:

```
In: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]
In: cat_s3
Out:
0 a
1 b
4 a
5 b
dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']

In: cat_s3.cat.remove_unused_categories()
Out:
0 a
1 b
4 a
5 b
dtype: category
Categories (2, object): ['a', 'b']
```

continue...

Chapter 8: Data Wrangling: Join, Combine, and Reshape

Note 1: Hierarchical Indexing

Enables higher dimensional data in a lower dimensional form using lists of lists as the index for the rows or columns. Either axis can have it.

You can see how many levels an index has by accessing its *nlevels* attribute:

```
Example:
In: frame.index.nlevels
Out: 2
```

The swaplevel method takes two level numbers or names and returns a new object with the levels interchanged (but the data is otherwise unaltered)

DataFrame's *set_index* function will create a new DataFrame using one or more of its columns as the index.

```
 Example:
 In: frame
Out:
a b c d
0 0 7 one 0
1 1 6 one 1
2 2 5 one 2
3 3 4 two 0
4 4 3 two 1
5 5 2 two 2
6 6 1 two 3

In: frame2 = frame.set_index(["c", "d"])
In: frame2
Out:
        a b
c d
one 0 0 7
        1 1 6
        2 2 5
two 0 3 4
        1 4 3
        2 5 2
        3 6 1
```

By default, the columns are removed from the DataFrame, though you can leave them in by passing *drop=False* to set_index.

reset*index does the opposite of _set_index*; the hierarchical index levels are moved into the columns.

```
 Example:
 In: frame2.reset_index()
Out:
c d a b
0 one 0 0 7
1 one 1 1 6
```

```
2 one 2 2 5
3 two 0 3 4
4 two 1 4 3
5 two 2 5 2
6 two 3 6 1
```

Note 2: Combining and Merging Datasets

Data contained in pandas objects can be combined in a number of ways:

pandas.merge

Connect rows in DataFrames based on one or more keys. Similar to SQL or other relational databases.

You must specify which column to join on. If that information is not specified, *pandas.merge* uses the overlapping column names as the keys.
It's a good practice to specify explicitly.
Example:

```
In: pd.merge(df1, df2, on="key")
Out:
key data1 data2
0 b 0 1
1 b 1 1
2 b 6 1
3 a 2 0
4 a 4 0
5 a 5 0
```

By default, *pandas.merge* does an "inner" join; the keys in the result are the intersection, or the common set found in both tables.
Other possible **options are "left", "right", and "outer"**.
Examples:

1# In: pd.merge(df1, df2, how="right")
2# In: pd.merge(df3, df4, left_on="lkey", right_on="rkey", how="outer")

Options Behavior:

how="inner" Use only the key combinations observed in both tables

how="left" Use all key combinations found in the left table
how="right" Use all key combinations found in the right table
how="outer" Use all key combinations observed in both tables together

***Many-to-many merges* form the Cartesian product of the matching keys.**
Example:

In:
df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],
....: "data1": pd.Series(range(6), dtype="Int64")})
In:
df2 = pd.DataFrame({"key": ["a", "b", "a", "b", "d"],
....: "data2": pd.Series(range(5), dtype="Int64")})

In:
df1
Out:
key data1
0 b 0
1 b 1
2 a 2
3 c 3
4 a 4
5 b 5

In:
df2
Out:
key data2
0 a 0
1 b 1
2 a 2
3 b 3
4 d 4

In:
pd.merge(df1, df2, on="key", how="left")
Out:
key data1 data2
0 b 0 1

1 b 0 3
2 b 1 1
3 b 1 3
4 a 2 0
5 a 2 2
6 c 3
7 a 4 0
8 a 4 2
9 b 5 1
10 b 5 3

In merge operations **the treatment of overlapping column names is the suffixes argument.**
Example:

In: pd.merge(left, right, on="key1", suffixes=("_left", "_right"))
Out:
key1 key2_left lval key2_right rval
0 foo one 1 one 4
1 foo one 1 one 5
2 foo two 2 one 4
3 foo two 2 one 5
4 bar one 3 one 6
5 bar one 3 two 7

pandas.concat

Concatenate or “stack” objects together along an axis.

By default, pandas.concat works along axis="index", producing another Series. If you pass axis="columns", the result will instead be a DataFrame.
Example:

In: s1
Out:
a 0
b 1
dtype: Int64

In: s2
Out:
c 2

d 3
e 4
dtype: Int64

In: s3
Out:
f 5
g 6
dtype: Int64

In: pd.concat([s1, s2, s3])
Out:
a 0
b 1
c 2
d 3
e 4
f 5
g 6
dtype: Int64

In: pd.concat([s1, s2, s3], axis="columns")
Out:
0 1 2
a 0
b 1
c 2
d 3
e 4
f 5
g 6

By default, pandas.concat works with outter joins.

You can use the *keys* argument to create a hierarchical index.
Alternatively, you can use dictionaries.
Example:

df1
one two
a 0 1

b 2 3
c 4 5
df2
three four
a 5 6
c 7 8

In: pd.concat([df1, df2], axis="columns", keys=["level1", "level2"])
Out:
level1 level2
one two three four
a 0 1 5.0 6.0
b 2 3 NaN NaN
c 4 5 7.0 8.0

(Alternative: pd.concat({"level1": df1, "level2": df2}, axis="columns"))

combine_first

Splice together overlapping data to fill in missing values in one object with values from another.
How to do it?

a.combine_first(b)

The output of combine_first with DataFrame objects will have the union of all the column names.

Note 3: Reshaping and Pivoting

There are a number of basic operations for rearranging tabular data. These are referred to as reshape or pivot operations.

Reshaping with Hierarchical Indexing

There are two primary actions:
stack
This “rotates” or pivots from the columns in the data to the rows.
Example:

In: data
Out:
number one two three
state

Ohio 0 1 2
Colorado 3 4 5
dtype: int64

In: result = data.stack()
In: result
Out:
state number
Ohio one 0
two 1
three 2
Colorado one 3
two 4
three 5
dtype: int64

unstack
This pivots from the rows into the columns.
Example:

In: data
Out:
state number
Ohio one 0
two 1
three 2
Colorado one 3
two 4
three 5
dtype: int64

In: result = data.unstack()
In: result
Out:
number one two three
state
Ohio 0 1 2
Colorado 3 4 5
dtype: int64

By default, the innermost level is unstacked/stacked. A different level can be chosen by passing a level number or name as argument.
Example:

```
In [131]: result.unstack(level=0)
Out[131]:
state Ohio Colorado
number
one 0 3
two 1 4
three 2 5
```

Stacking/Unstacking filters out missing data by default, so the operation is more easily invertible. The *dropna=False* argument can be passed to negated that.
Example:

```
In: data2.unstack().stack()
Out:
one a 0
b 1
c 2
d 3
two c 4
d 5
e 6
dtype: Int64

In: data2.unstack().stack(dropna=False)
Out:
one a 0
b 1
c 2
d 3
e
two a
b
c 4
d 5
e 6
dtype: Int64
```

Long or stacked format

A common way to store multiple time series in databases and CSV files.
In this format, individual values are represented by a single row in a table rather than multiple values per row.
Each row in the table represents a single observation.
Example:

```
In: long_data[:10]
Out:
date item value
0 1959-01-01 realgdp 2710.349
1 1959-01-01 infl 0.000
2 1959-01-01 unemp 5.800
3 1959-04-01 realgdp 2778.801
4 1959-04-01 infl 2.340
5 1959-04-01 unemp 5.100
6 1959-07-01 realgdp 2775.488
7 1959-07-01 infl 2.740
8 1959-07-01 unemp 5.300
9 1959-10-01 realgdp 2785.204
```

Data is frequently stored this way in relational SQL databases since it allows the number of distinct values in the item column to change as data is added to the table.
In the previous example, date and item would usually be the primary keys, offering both relational integrity and easier joins.

The .pivot() method performs a transformation from long/stacked format to a DataFrame containing one column per distinct item with the rows values indexed by timestamps in the date column.
Example:

```
In: pivoted = long_data.pivot(index="date", columns="item", values="value")
In: pivoted.head()
Out:
item infl realgdp unemp
date
1959-01-01 0.00 2710.349 5.8
1959-04-01 2.34 2778.801 5.1
1959-07-01 2.74 2775.488 5.3
```

1959-10-01 0.27 2785.204 5.6
1960-01-01 2.31 2847.699 5.2

The first two values passed to the .pivot() method are the columns to be used, respectively, as the row and column index, then finally an optional value column to fill the DataFrame.
Example:
Suppose you had two value columns that you wanted to reshape simultaneously:

```
In: long_data["value2"] = np.random.standard_normal(len(long_data))
In: long_data[:10]
Out:
date item value value2
0 1959-01-01 realgdp 2710.349 0.802926
1 1959-01-01 infl 0.000 0.575721
2 1959-01-01 unemp 5.800 1.381918
3 1959-04-01 realgdp 2778.801 0.000992
4 1959-04-01 infl 2.340 -0.143492
5 1959-04-01 unemp 5.100 -0.206282
6 1959-07-01 realgdp 2775.488 -0.222392
7 1959-07-01 infl 2.740 -1.682403
8 1959-07-01 unemp 5.300 1.811659
9 1959-10-01 realgdp 2785.204 -0.351305
```

By omitting the last argument, you obtain a DataFrame with hierarchical columns:

In [160]: pivoted = long_data.pivot(index="date", columns="item")
In [161]: pivoted.head()
Out[161]:
value value2
item infl realgdp unemp infl realgdp unemp
date
1959-01-01 0.00 2710.349 5.8 0.575721 0.802926 1.381918
1959-04-01 2.34 2778.801 5.1 -0.143492 0.000992 -0.206282
1959-07-01 2.74 2775.488 5.3 -1.682403 -0.222392 1.811659
1959-10-01 0.27 2785.204 5.6 0.128317 -0.351305 -1.313554
1960-01-01 2.31 2847.699 5.2 -0.615939 0.498327 0.174072

The .melt() method performs the inverse operation to pivot.
it merges multiple columns into one, producing a DataFrame that is longer than the input.

Chapter 9: Plotting and Visualization

Note 1: Check Bokeh, Altair, and Plotly libs out

This libs take advantage of modern web technology to create interactive visualizations that integrate well with the Jupyter notebook.

Note 2: Matplotlib Figures and subplots

Creating a figure object

Plots in matplotlib reside within a *Figure object*. You can create a new figure with plt.figure:

```
In:
fig = plt.figure()
```

Adding subplots

You can't make a plot with a blank figure. You have to create one or more subplots using add_subplot:
Example:

In: ax1 = fig.add_subplot(2, 2, 1)
In: ax2 = fig.add_subplot(2, 2, 2)
In: ax3 = fig.add_subplot(2, 2, 3)
In: ax4 = fig.add_subplot(2, 2, 4)

The first line means that the figure should be 2 × 2
(so up to four plots in total), and we're selecting the first of
four subplots (numbered from 1).

The objects returned by fig.add_subplot here are AxesSubplot objects, on which you can directly plot on the other empty subplots by calling each one's instance method:

```
In: ax1.hist(np.random.standard_normal(100), bins=20,
color="black", alpha=0.3);
In: ax2.scatter(np.arange(30), np.arange(30) +
3 * np.random.standard_normal(30));
```

To check the plot types in matplotlib: https://matplotlib.org/stable/plot_types/index

The pythonic way of creating subplots its the following

fig, axes = plt.subplots(2, 3)

The *axes* array can then be indexed like a two-dimensional array. For example, axes[0, 1] refers to the subplot in the top row at the center.

sharex and sharey arguments
To indicate that subplots should have the same x-axis or y-axis use the **sharex** or **sharey** arguments

Editing Figure objects and their subplots

fig.subplots_adjust() method
You can change the spacing using the *subplots_adjust* method on Figure objects.

wspace* and *hspace control the percent of the figure width and figure height

```
Example:
fig.subplots_adjust(wspace=0, hspace=0)
```

axes objects methods
Matplotlib axes objects have methods like **xlim, xticks, and xticklabels**.
These control the plot range, tick locations, and tick labels, respectively.

They can be used in 2 ways:
Called with **no arguments**:
Returns the current parameter value returns the current x-axis plotting range.
Example:
ax.xlim()
Called with **parameters:**
Sets the parameter value.
Example:
ax.xlim([0, 10]) sets the x-axis range to 0 to 10.
ax.set*xticks()*
ax.set_xticklabels(["one", "two", "three", "four",
"five"], rotation=30, fontsize=8)
ax.set_xlabel("Stages")
ax.set_title("My first matplotlibges')
ax.set_title("My first matplotlib plot")

ax.legend method()
You must call _ax.legend to create the legend, whether or not you passed the label options when plotting the data.

The easiest way to create legends is to pass the label argument when adding each piece of the plot!
Example:

In [np.random.randn(1000).cumsum(),
color="black", linestyle.randn(1000).cumsum(),
color="black", label="one");

In [48]: ax.plot(np.random.randn(1000).cumsum(),
color="black", linestyle="dashed, label="two");

In [49]: ax.plot(np.random.randn(1000).cumsum(),
color="black", linestyle="dotted", label="three");

***Once you've done this, you can call ax.legend()* _to automatically create a legend.**

Annotations and Drawing on a Subplot
To draw your own plot annotations, which could consist of text, arrows, or other shapes. You can ause the **text, arrow, and annotate** functions. Drawing at given coordinates (x, y) on the plot with optional custom styling.
Example:

ax.text(x, y, "Hello world!", family="monospace", fontsize=10)

Drawing other shapes requires some more work. Matplotlib has objects that represent many common shapes, referred to as patches. Some of these, like Rectangle and Circle, are found in matplotlib.pyplot, but **the full set is located in matplotlib.patches.**

To add a shape to a plot, you **create the patch object and add it to the subplot with ax.add_patch()**
Example:

fig, ax = plt.subplots()
rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color="black", alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color="blue", alpha=0.3)
pgon = plt.Polygon(0.15, 0.15], [0.35, 0.4], [0.2, 0.6,
color="green", alpha=0.5)
ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)

To modify the default configuration use the rc method.
Example:
to set the global default figure size to be 10 × 10, you could enter:
plt.rc("figure", figsize=(10, 10))
All of the current configuration settings are found in the plt.rcParams dictionary.

To restore to their default values call the plt.rcdefaults() function.

Note 3: Plotting with pandas and seaborn

The pandas *.plot()* attribute contains a "family" of methods for different plot type

The default *df.plot()* is equivalent to *df.plot.line()*.

DataFrame-specific plot arguments:
subplots:
Plot each DataFrame column in a separate subplot
layouts:
2-tuple (rows, columns) providing layout of subplots
sharex:
If subplots=True, share the same x-axis, linking ticks and limits
sharey:
If subplots=True, share the same y-axis
legend:
Add a subplot legend (**True by default**)
sort_columns:
Plot columns in alphabetical order **by default uses existing order**

Bar Plots (vertical,horizontal,stacked)
The plot.bar() and plot.barh() make vertical and horizontal bar
Example:

In: fig, axes = plt.subplots(2, 1)
In: data = pd.Series(np.random.uniform(size=16),
index=list("abcdefghijklmnop"))
In: data.plot.bar(ax=axes[0], color="black", alpha=0.7)
Out: <AxesSubplot:>
In: data.plot.barh(ax=axes[1], color="black", alpha=0.7)

*We create stacked bar plots from a DataFrame by passing **stacked=True**, resulting in the value in each row being stacked together horizontally*

A useful recipe for bar plots is to visualize a Series's value frequency is the use of value_counts
Example:

```
s.value_counts().plot.bar()
```

The ***pandas.crosstab*** **method** is a convenient way to compute a simple frequency table from 2 DataFrame columns.
Example:

In: tips.head()
Out:
tot_bill tip smoker day time size
0 16.99 1.01 No Sun Dinner 2
1 10.34 1.66 No Sun Dinner 3
2 21.01 3.50 No Sun Dinner 3
3 23.68 3.31 No Sun Dinner 2
4 24.59 3.61 No Sun Dinner 4

In: party_counts = pd.crosstab(tips["day"], tips["size"])
In: party_counts = party_counts.reindex(index=["Thur", "Fri", "Sat", "Sun"])

In: party_counts
Out:
size 1 2 3 4 5 6
day
Thur 1 48 4 5 1 3
Fri 1 16 1 1 0 0
Sat 2 53 18 13 1 0
Sun 0 39 15 18 3 1

In: party_counts = party_counts.loc[:, 2:5]
In: party_pcts = party_counts.div(party_counts.sum(axis="columns"),
axis="index")
In: party_pcts
Out:
size 2 3 4 5
day
Thur 0.827586 0.068966 0.086207 0.017241
Fri 0.888889 0.055556 0.055556 0.000000

Sat 0.623529 0.211765 0.152941 0.011765
Sun 0.520000 0.200000 0.240000 0.040000

In: party_pcts.plot.bar(stacked=True)

Density plot (KDE plot)
A density plot, also known as kernel density estimate (KDE) plot, is formed by computing an estimate of a continuous probability distribution that might have generated the observed data.
Example:

```
In: tips["tip_pct"].plot.density()
```

Chapter 10: Data Aggregation and Group Operations

Note 1: Nuisance column

Nuisance columns are columns where the aggregation function parsed to a groupby method wouldn't make sense, so they get automatically excluded from the result.
Example:

In: df.groupby("key1").mean()
Out:
key2 data1 data2
key1
a 1.5 0.555881 0.441920
b 1.5 0.705025 -0.144516

In: df.groupby("key2").mean()
Out:
data1 data2
key2
1 0.333636 0.115218
2 -0.038393 0.888106

Note 2: Missing values in group keys

Missing values in a group key get **excluded by default.**
Disabled using *dropna=False* as a groupby parameter.

Note 3: Difference between .size() and .count()

The size method returns the total number of elements **including null (NaN) values.**
The count method counts non-NA/null observations, **excluding null (NaN) values.**

Note 4: Iterating over groups

The object returned by *groupby* supports iteration, generating a sequence of **2-tuples containing the group name along with the chunk of data.**

In the case of multiple keys, the first element in the tuple will be a tuple of key values.

Example:
A recipe that may be useful is:
computing a dictionary of the data pieces as a one-liner

In: pieces = {name: group for name, group in df.groupby("key1")}
In: pieces["b"]
Out:
key1 key2 data1 data2
3 b 2 -0.555730 1.007189
4 b 1 1.965781 -1.296221

Note 5: Selecting columns in groups

_Indexing a GroupBy object created from a DataFrame with a column name or array of column names has the effect of column subsetting for aggregation.

This means that:

`df.groupby("key1")["data1"]` df.groupby("key1")"data2"

are conveniences for:
`df["data1"].groupby(df["key1"])` df"data2".groupby(df["key1"])

Note 6: .groupby() with dictionaries

The groupby method also accepts dictionaries.
Example:
`In: people` Out:
`a b c d e` Joe 1.352917 0.886429 -2.001637 -0.371843 1.669025
`Steve -0.438570 -0.539741 0.476985 3.248944 -1.021228` Wanda -0.577087 NaN NaN 0.523772 0.000940
`Jill 1.343810 -0.713544 -0.831154 -2.370232 -1.860761` Trey -0.860757 0.560145 -1.265934 0.119827 -1.063512

```
In: mapping = {"a": "red", "b": "red", "c": "blue",
"d": "blue", "e": "red", "f" : "orange"}

In: by_column = people.groupby(mapping, axis="columns")

In: by_column.sum()
Out:
blue red
Joe -2.373480 3.908371
Steve 3.725929 -1.999539
Wanda 0.523772 -0.576147
Jill -3.201385 -1.230495
Trey -1.146107 -1.364125

The key "f" is included to highlight that unused grouping keys are OK.
```

Note 7: .groupby() with functions

Any function passed as a group key will be called once per index value (or once per column value if using axis="columns"), with the return values being used as the group names.
Example:
`Consider the DataFrame from the previous note, which has people's first names as index values.

In: people.groupby(len).sum()
Out:
a b c d e
3 1.352917 0.886429 -2.001637 -0.371843 1.669025
4 0.483052 -0.153399 -2.097088 -2.250405 -2.924273
5 -1.015657 -0.539741 0.476985 3.772716 -1.020287

Note 8: Data Aggregation

Aggregations refer to any data transformation that produces scalar values from arrays.

You can pass your own functions to the .agg() method or even a list of functions.

While passing *your own function*, what occurs inside is up to you, but **it must either return a pandas object or a scalar value**

If you pass a list of functions, you get back a DataFrame with column names taken from the functions.
Example:
`In [71]: grouped_pct.agg(["mean", "std", peak_to_peak])` Out[71]:
`mean std peak_to_peak` day smoker
`Fri No 0.151650 0.028123 0.067349` Yes 0.174783 0.051293 0.159925
`Sat No 0.158048 0.039767 0.235193` Yes 0.147906 0.061375 0.290095
`Sun No 0.160113 0.042347 0.193226` Yes 0.187250 0.154134 0.644685
`Thur No 0.160298 0.038774 0.193350` Yes 0.163863 0.039389 0.151240

If you pass a list of (name, function) tuples, the first element of each tuple will be used as the DataFrame column names (you can think of a list of 2-tuples as an ordered mapping).
Example:
`In: grouped_pct.agg([("average", "mean"), ("stdev", np.std)])` Out:
`average stdev` day smoker
`Fri No 0.151650 0.028123` Yes 0.174783 0.051293
`Sat No 0.158048 0.039767` Yes 0.147906 0.061375
`Sun No 0.160113 0.042347` Yes 0.187250 0.154134
`Thur No 0.160298 0.038774` Yes 0.163863 0.039389

Note 9: .transform() method

The built-in method called transform is similar to .apply() but imposes more constraints on the kind of function you can use:

• It can produce a scalar value to be broadcast to the shape of the group.
• It can produce an object of the same shape as the input group.
• It must not mutate its input.

The .transform() method allows us to perform what is called an unwrapped group operation:
Example:
`In [158]: g.transform('mean')` Out[158]:
`0 4.5` 1 5.5
`2 6.5` 3 4.5
`4 5.5` 5 6.5
`6 4.5` 7 5.5
`8 6.5` 9 4.5
`10 5.5` 11 6.5
`` Name: value, dtype: float64

In [159]: normalized = (df['value'] - g.transform('mean'))

/ g.transform('std')
In [160]: normalized
Out[160]:
0 -1.161895
1 -1.161895
2 -1.161895
3 -0.387298
4 -0.387298
5 -0.387298
6 0.387298
7 0.387298
8 0.387298
9 1.161895
10 1.161895
11 1.161895
Name: value, dtype: float64

This is doing arithmetic between the outputs of multiple groupby operations
instead of writing a function and passing it to groupby(...).apply.
That is what is meant by “unwrapped.”

While an unwrapped group operation may involve multiple group aggregations, ***the overall benefit of vectorized operations often outweighs this.***

TLDR: DataFrame.transform >>> DataFrame.apply - where you can use it.

Note 10: .pivot_table()

A pivot table is a data summarization tool frequently found in spreadsheet programs and other data analysis software. It aggregates a table of data by one or more keys, arranging the data in a rectangle with some of the group keys along the rows and some along the columns.

In addition to providing a convenience interface to groupby, pivot_table can add partial totals, also known as margins. ***To include partial totals pass margins=True.***
Example:
`In: tips.pivot_table(index=["time", "day"], columns="smoker",` values=["tip_pct", "size"], margins=True)

Out:
size tip_pct
smoker No Yes All No Yes All
time day

Dinner
Fri 2.000000 2.222222 2.166667 0.139622 0.165347 0.158916
Sat 2.555556 2.476190 2.517241 0.158048 0.147906 0.153152
Sun 2.929825 2.578947 2.842105 0.160113 0.187250 0.166897
Thur 2.000000 NaN 2.000000 0.159744 NaN 0.159744
Lunch
Fri 3.000000 1.833333 2.000000 0.187735 0.188937 0.188765
Thur 2.500000 2.352941 2.459016 0.160311 0.163863 0.161301
All 2.668874 2.408602 2.569672 0.159328 0.163196 0.160803

Note 11: Cross-tabulations

A cross-tabulation (or crosstab for short) is a special case of a pivot table that computes group frequencies.
The first two arguments to crosstab can each be an array or Series or a list of arrays.
Example:
`In: pd.crosstab([tips["time"], tips["day"]], tips["smoker"], margins=True)` Out:
`smoker No Yes All` time day
`Dinner` Fri 3 9 12
`Sat 45 42 87` Sun 57 19 76
`Thur 1 0 1` Lunch
`Fri 1 6 7` Thur 44 17 61
`All` 151 93 244

Chapter 11: Time Series

How you mark and refer to time series data depends on the application, and you may have one of the following:
1) Timestamps
Specific instants in time.
1) Fixed periods
Such as the whole month of January 2017, or the whole year 2020.
1) Intervals of time
Indicated by a start and end timestamp. Periods can be thought of as special cases of intervals.
1) Experiment or elapsed time
Each timestamp is a measure of time relative to a particular start time (e.g., the diameter of a cookie baking each second since being placed in the oven), starting from 0.

Note 1: datetime library

date

Store calendar date (year, month, day) using the Gregorian calendar

time

Store time of day as hours, minutes, seconds, and microseconds

datetime

Store both date and time

timedelta

The difference between two datetime values (as days, seconds, and microseconds)
`Example:
`In: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)` In: delta
`Out: datetime.timedelta(days=926, seconds=56700)` In: delta.days
`Out: 926` In: delta.seconds
`` Out: 56700

tzinfo

Base type for storing time zone information

Note 2: Converting Between String and Datetime

You can use the *strftime* method to convert datetime objects to strings and pass a format specification like this:

```
In: stamp = datetime(2011, 1, 3)
In: stamp.strftime("%Y-%m-%d")
Out: '2011-01-03'
```

You can use the *strptime* method to convert strings to datetime objects and pass a format specification like this:

In: value = "2011-01-03"
In: datetime.strptime(value, "%Y-%m-%d")
Out[30]: datetime.datetime(2011, 1, 3, 0, 0)

In: datestrs = ["7/6/2011", "8/6/2011"]
In: [datetime.strptime(x, "%m/%d/%Y") for x in datestrs]
Out:

[datetime.datetime(2011, 7, 6, 0, 0),
datetime.datetime(2011, 8, 6, 0, 0)]

Below there are the identifies for each type of datetime value.

Types and descriptions

`%Y` Four-digit year
`%y` Two-digit year
`%m` Two-digit month [01, 12]
`%d` Two-digit day [01, 31]
`%H` Hour (24-hour clock) [00, 23]
`%I` Hour (12-hour clock) [01, 12]
`%M` Two-digit minute [00, 59]
`%S` Second [00, 61] (seconds 60, 61 account for leap seconds)
`%f` Microsecond as an integer, zero-padded (from 000000 to 999999)
`%j` Day of the year as a zero-padded integer (from 001 to 336)
`%w` Weekday as an integer [0 (Sunday), 6]
`%u` Weekday as an integer starting from 1, where 1 is Monday.
`%U` Week number of the year [00, 53]; Sunday is considered the first day of the week, and days before the first Sunday of the year are “week 0”
`%W` Week number of the year [00, 53]; Monday is considered the first day of the week, and days before the first Monday of the year are “week 0”
`%z` UTC time zone offset as +HHMM or -HHMM; empty if time zone naive
`%Z` Time zone name as a string, or empty string if no time zone
`%F` Shortcut for %Y-%m-%d (e.g., 2012-4-18)
`%D` Shortcut for %m/%d/%y (e.g., 04/18/12)

pandas.to_datetime() method parses many different kinds of date representations.

Note 3: pandas vs default lib

pandas.Timestamp can store nanosecond precision data, while *datetime* stores only up to microseconds.

pandas.Timestamp can store frequency information and understands how to do time zone conversions and other kinds of manipulations.

Note 4: Frequencies / offsets

Frequencies in pandas are composed of a base frequency (offset) and a multiplier. Base frequencies are typically referred to by a string alias, like "M" for monthly or "H" for

hourly.
You can define a multiple of a base frequency by passing an integer to its alias
like: "H" or "4H".
Example:
`In: pd.date_range("2000-01-01", "2000-01-03 23:59", freq="4H")` Out:
`DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',` '2000-01-01
08:00:00', '2000-01-01 12:00:00',
`'2000-01-01 16:00:00', '2000-01-01 20:00:00',` '2000-01-02 00:00:00', '2000-01-02
04:00:00',
`'2000-01-02 08:00:00', '2000-01-02 12:00:00',` '2000-01-02 16:00:00', '2000-01-02
20:00:00',
`'2000-01-03 00:00:00', '2000-01-03 04:00:00',` '2000-01-03 08:00:00', '2000-01-03
12:00:00',
`'2000-01-03 16:00:00', '2000-01-03 20:00:00'],` dtype='datetime64[ns]', freq='4H')

Some frequencies describe points in time that are not evenly spaced. For example,
"M" (calendar month end) and "BM" (last business/weekday of month).
We refer to these as anchored offsets.

Note 5: .date_range() method

pandas.date_range generates **daily timestamps by default**. You can generate ranges of dates based on different intervals consulting a time series frequencies table in pandas docs.

If you pass only a start or end date, you must pass a number of periods to generate:

Example:
`In: pd.date_range(start="2012-04-01", periods=20)` Out:
`DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-04-04',` '2012-04-
05', '2012-04-06', '2012-04-07', '2012-04-08',
`'2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',` '2012-04-13', '2012-04-14',
'2012-04-15', '2012-04-16',
`'2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],` dtype='datetime64[ns]',
freq='D')

Note 6: Resampling

It's often desirable to work relative to a fixed frequency, such as daily, monthly, or every 15 minutes. You can convert the sample time series to fixed daily frequency by calling .resample():
Example:
`In: resampler = ts.resample("D")
The string "D" is interpreted as daily frequency.

Downsampling: Aggregating higher frequency data to lower frequency

Also done by calling .resample()
Example:
`In [215]: ts` Out[215]:
`2000-01-01 00:00:00 0` 2000-01-01 00:01:00 1
`2000-01-01 00:02:00 2` ...
`2000-01-01 00:09:00 9` 2000-01-01 00:10:00 10
`2000-01-01 00:11:00 11` Freq: T, dtype: int64

Suppose you wanted to aggregate this data into five-minute chunks or bars by taking the sum of each group (**you could also take the mean, std, median or any agg function**):

`In [216]: ts.resample("5min").sum()` Out[216]:
`2000-01-01 00:00:00 10` 2000-01-01 00:05:00 35
`2000-01-01 00:10:00 21` Freq: 5T, dtype: int64

Upsampling: Converting lower frequency to higher frequency

We use the .asfreq() method to convert to the higher frequency without any aggregation.
Example:
`In: df_daily = frame.resample(="D").asfreq()` In: df_daily
`Out:` Colorado Texas New York Ohio
`2000-01-05 -0.896431 0.927238 0.482284 -0.867130` 2000-01-06 NaN NaN NaN NaN
`2000-01-07 NaN NaN NaN NaN` 2000-01-08 NaN NaN NaN NaN
`2000-01-09 NaN NaN NaN NaN` 2000-01-10 NaN NaN NaN NaN
`2000-01-11 NaN NaN NaN NaN` 2000-01-12 0.493841 -0.155434 1.397286 1.507055

Tip: use .ffill() afterwards

`In: frame.resample("D").ffill()` Out:
`Colorado Texas New York Ohio` 2000-01-05 -0.896431 0.927238 0.482284 -0.867130
`2000-01-06 -0.896431 0.927238 0.482284 -0.867130` 2000-01-07 -0.896431 0.927238 0.482284 -0.867130
`2000-01-08 -0.896431 0.927238 0.482284 -0.867130` 2000-01-09 -0.896431 0.927238 0.482284 -0.867130
`2000-01-10 -0.896431 0.927238 0.482284 -0.867130` 2000-01-11 -0.896431 0.927238 0.482284 -0.867130
`` 2000-01-12 0.493841 -0.155434 1.397286 1.507055

.resample() method arguments:

rule
String, DateOffset, or timedelta indicating desired resampled frequency (for example, ‘M', ’5min', or Second(15))
axis
Axis to resample on; default axis=0
fill_method
How to interpolate when upsampling, as in "ffill" or "bfill"; by default does no interpolation
closed
In downsampling, which end of each interval is closed (inclusive), "right" or "left"
label
In downsampling, how to label the aggregated result, with the "right" or "left" bin edge (e.g., the 9:30 to 9:35 five-minute interval could be labeled 9:30 or 9:35)
limit
When forward or backward filling, the maximum number of periods to fill
kind
Aggregate to periods ("period") or timestamps ("timestamp"); defaults to the type of index the time series has
convention
When resampling periods, the convention ("start" or "end") for converting the low-frequency period to high frequency; defaults to "start"
origin
The “base” timestamp from which to determine the resampling bin edges; can also be one of "epoch", "start", "start_day", "end", or "end_day"; see the resample docstring for full details.
offset
An offset timedelta added to the origin; defaults to None

Note 7: Open-high-low-close (OHLC) resampling

In finance, a popular way to aggregate a time series is to compute four values for each bucket: the first (open), last (close), maximum (high), and minimal (low) values.

By using the **.ohlc()** method, you obtain a DataFrame having columns containing these four aggregates, which are efficiently computed in a single call.
Example:
`In: ts = pd.Series(np.random.permutation(np.arange(len(dates))), index=dates)`
In: ts.resample("5min").ohlc()
`Out:` open high low close
`2000-01-01 00:00:00 8 8 1 5` 2000-01-01 00:05:00 6 11 2 2
`2000-01-01 00:10:00 0 7 0 7

Note 8: Shifting (Leading and Lagging) Data

Shifting refers to moving data backward and forward through time. Both Series and DataFrame have a shift method for doing naive shifts forward or backward, leaving the index unmodified.

Because naive shifts (without specified frequency) discards some data and introduces null values, if frequency is known, you should pass it to shift as a parameter to advance the timestamps instead of simply the data:
Example:
Ideal:
`In [95]: ts.shift(2, freq="M")` Out[95]:
`2000-03-31 -0.066748` 2000-04-30 0.838639
`2000-05-31 -0.117388` 2000-06-30 -0.517795
`Freq: M, dtype: float64 Naive:` In: ts
`Out:` 2000-01-31 -0.066748
`2000-02-29 0.838639` 2000-03-31 -0.117388
`2000-04-30 -0.517795` Freq: M, dtype: float64

`In: ts.shift(2)` Out:
`2000-01-31 NaN` 2000-02-29 NaN
`2000-03-31 -0.066748` 2000-04-30 0.838639
`Freq: M, dtype: float64

Note 9: Timezones

In Python, time zone information comes from the *pytz* library which exposes the Olson database, a compilation of world time zone information. the pandas lib wraps *pytz's* functionality so you can ignore its API outside of the time zone names.

Timestamps are naive to time zones by default, to work with specifics timezones use:
tz_localize
to define a time zone for a naive timestamp
tz_convert
to convert a time zone to another

Note 10: Moving window functions

Movings windows are calculations across a series of data points in a time series performed by "sliding" a window across the data. Instead of calculating a single average for the entire dataset, you calculate multiple averages each over a subset of data points.

This technique is useful for smoothing noisy data or filling in gaps.

The window can be fixed-size or have exponentially decaying weights where more recent points in the window have a higher weight.

rolling operator:

`close_px["AAPL"].rolling(250).mean().plot()

By default, rolling functions require all of the values in the window to be non-NA.
This behavior can be changed to account for missing data and, in particular, the fact that you will have fewer than window periods of data at the beginning of the time series:
`close_px["AAPL"].rolling(250, min_periods=10).std().plot()

The rolling function also accepts a string indicating a fixed-size time offset:
These are the same alias that you can pass to resample.
For example, we could plot a 20-day rolling mean like:
`close_px.rolling("20D").mean().plot()

expanding operator:

The expanding mean starts the time window from the same point as the rolling window and increases the size of the window until it encompasses the whole series.
To compute an expanding window mean, use the expanding operator:
`std250.expanding().mean().expanding().mean().plot()

ewm operator:

An alternative to using a fixed window size with equally weighted observations is to specify a constant decay factor to give more weight to more recent observations.

Exponentially weighted statistic places more weight on more recent observations and adapts faster to changes compared with the equal-weighted version.

pandas has the **ewm operator (which stands for exponentially weighted moving)**:

`aapl_px.ewm(span=30).mean()

span *is like the window size for a exponentially weighted function*

user defined moving window averages:

The only requirement is that the function produce a single value (a reduction) from each piece of the array.

Example:
`returns["AAPL"].rolling(250).apply(user_defined_function)

Chapter 12: Introduction to Modeling Libraries in Python

Note 1: Feature engineering

Feature engineering in machine learning describes **any data transformation or analytics that extract information from a raw dataset that may be useful in a modeling context.**

Note 2: From data prep to modeling

The point of contact between pandas and other analysis libraries is usually NumPy arrays. To turn a DataFrame into a NumPy array, use the to numpy method.

`In: DataFrame123.to_numpy()

The *to_numpy method* is intended to be used when your data is homogeneous for example, all numeric types.

If you have heterogeneous data, the result will be an ndarray of Python objects.

Note 3: Working with dummies

If we wanted to replace the 'category' column with dummy variables, we create dummy variables, drop the 'category' column, and then join the result.
Example:
`In: data['category'] = pd.Categorical(['a', 'b', 'a', 'a', 'b'],` categories=['a', 'b'])
`In: data` Out:
`x0 x1 y category` 0 1 0.01 -1.5 a
`1 2 –0.01 0.0 b` 2 3 0.25 3.6 a
`3 4 –4.10 1.3 a` 4 5 0.00 -2.0 b

In: dummies = pd.get_dummies(data.category, prefix='c')
In: data_with_dummies = data.drop('category', axis=1).join(dummies)
In: data_with_dummies
In: data_with_dummies
Out:
x0 x1 y cat_a cat_b
0 1 0.01 -1.5 1 0
1 2 -0.01 0.0 0 1
2 3 0.25 3.6 1 0
3 4 -4.10 1.3 1 0
4 5 0.00 -2.0 0 1

Note 4: Patsy

Patsy is a Python library for describing statistical models (especially linear models) with a string-based “formula syntax”.

Patsy’s formulas are a special string syntax that looks like:

`y ~ x0 + x1

The syntax a + b does not mean to add a to b, but rather that these are terms in the design matrix created for the model. The **patsy.dmatrices** function takes a formula string along with a dataset and produces design matrices for a linear model:

`In: import patsy` In: y, X = patsy.dmatrices('y ~ x0 + x1', data)

Now we have:

`In: y` Out:
`DesignMatrix with shape (5, 1)` y
`-1.5` 0.0
`3.6` 1.3
`-2.0` Terms:
`'y' (column 0)

`In: X` Out:
`DesignMatrix with shape (5, 3)` Intercept x0 x1
`1 1 0.01` 1 2 -0.01
`1 3 0.25` 1 4 -4.10
`1 5 0.00` Terms:
`'Intercept' (column 0)` 'x0' (column 1)
`'x1' (column 2)

These Patsy DesignMatrix instances are NumPy ndarrays with additional metadata

You can suppress the intercept by adding the term + 0 to the model:
`In: patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]

Out:
DesignMatrix with shape (5, 2)
x0 x1
1 0.01
2 -0.01
3 0.25
4 -4.10

5 0.00
Terms:
'x0' (column 0)
'x1' (column 1)

The Patsy objects can be passed directly into algorithms like numpy.linalg.lstsq, which performs an ordinary least squares regression:

`In: coef, resid, , = np.linalg.lstsq(X, y)

The model metadata is retained in the design_info attribute, so you can reattach the model column names to the fitted coefficients to obtain a Series.
Example:

In: coef
Out:
array(0.3129], [-0.0791], [-0.2655)
In: coef = pd.Series(coef.squeeze(), index=X.design_info.column_names)
In: coef
Out:
Intercept 0.312910
x0 -0.079106
x1 -0.265464
dtype: float64

You can mix Python code into your Patsy formulas; the library will try to find the functions you use in the enclosing scope:
`In: y, X = patsy.dmatrices('y ~ x0 + np.log(np.abs(x1) + 1)', data)

Some commonly used variable transformations include:
standardizing (to mean 0 and variance 1)
centering (subtracting the mean)

Patsy has built-in functions for this purpose:
`In: y, X = patsy.dmatrices('y ~ standardize(x0) + center(x1)', data)

Stateful transformations:

As part of a modeling process, you may fit a model on one dataset, then evaluate the model based on another.

When applying transformations like center and standardize, you should be careful when using the model to form predications based on new data.

These are called stateful transformations, because you must use statistics like the mean or standard deviation of the original dataset when transforming a new dataset.

The patsy.build_design_matrices function can apply transformations to new out-of-sample data using the saved information from the original in-sample dataset.
Example:

```
In: new_data = pd.DataFrame({ 'x0': [6, 7, 8, 9],
'x1': [3.1, -0.5, 0, 2.3], 'y': [1, 2, 3, 4]})
In: new_X = patsy.build_design_matrices([X.design_info], new_data)
In: new_X
Out:
[DesignMatrix with shape (4, 3)
Intercept standardize(x0) center(x1)
1 2.12132 3.87
1 2.82843 0.27
1 3.53553 0.77
1 4.24264 3.07
Terms:
'Intercept' (column 0)
'standardize(x0)' (column 1)
'center(x1)' (column 2)]
```

I function:

Because the plus symbol (**+**) in the context of Patsy formulas does not mean addition, when you want to add columns from a dataset by name, you must wrap them in the special **I function**.
Example:
`In: y, X = patsy.dmatrices('y ~ I(x0 + x1)', data)

C function:

Numeric columns can be interpreted as categorical with the *C* function:
In: y, X = patsy.dmatrices('v2 ~ C(key2)', data)

See the online documentation for more.

Note 5: statsmodels & scikitlearn

statsmodels

statsmodels contains more “classical” frequentist statistical methods, while Bayesian methods and machine learning models are found in other libraries.

Some kinds of models found in statsmodels include:
• *Linear models, generalized linear models, and robust linear models*
• *Linear mixed effects models*
• *Analysis of variance (ANOVA) methods*
• *Time series processes and state space models*
• *Generalized method of moments*

scikit-learn

scikit-learn is one of the most widely used and trusted general-purpose Python machine learning toolkits.

read the docs, practice with kaggle datasets and dominate the shit out of it
lets go!

missing data

Libraries like statsmodels and scikit-learn generally cannot be fed missing data. There are a number of ways to do missing data imputation, but a simple example is to use the median of the training dataset to fill the nulls.
Example:
`In: impute_value = train['Age'].median()` In: train['Age'] = train['Age'].fillna(impute_value)
`In: test['Age'] = test['Age'].fillna(impute_value)

Chapter 13: Practical examples/tips/extras

"Now that we’ve reached the final chapter of this book, we’re going to take a look at a number of real-world datasets..."

Below the examples I liked:

collections.Counter or pandas.value_counts()

Counting the top 10 time zones and their counts

1. "...If you search the Python standard library, you may find the *collections.Counter class*, which makes this task even simpler..."

```
In: from collections import Counter
In: counts = Counter(time_zones)
```

```
In: counts.most_common(10)
Out[27]:
[('America/New_York', 1251),
('', 521),
('America/Chicago', 400),
('America/Los_Angeles', 382),
('America/Denver', 191),
('Europe/London', 74),
('Asia/Tokyo', 37),
('Pacific/Honolulu', 36),
('Europe/Madrid', 35),
('America/Sao_Paulo', 33)]
```

2. We can also use the pd.value_counts() method for the columns in a DataFrame:

```
In [31]: tz_counts = frame["tz"].value_counts()
In [32]: tz_counts.head()
Out[32]:
America/New_York 1251
521
America/Chicago 400
America/Los_Angeles 382
America/Denver 191
Name: tz, dtype: int64
```

excluding missing data

"...Since some of the agents are missing, we'll exclude these from the data:"
`In: cframe = frame[frame["a"].notna()].copy()

... to finish