PYTHON FOR DATA ANALYSIS USING NUMPY & PANDAS

Notes by Michael Brothers

http://titaniumventures.net/library/

Table of Contents

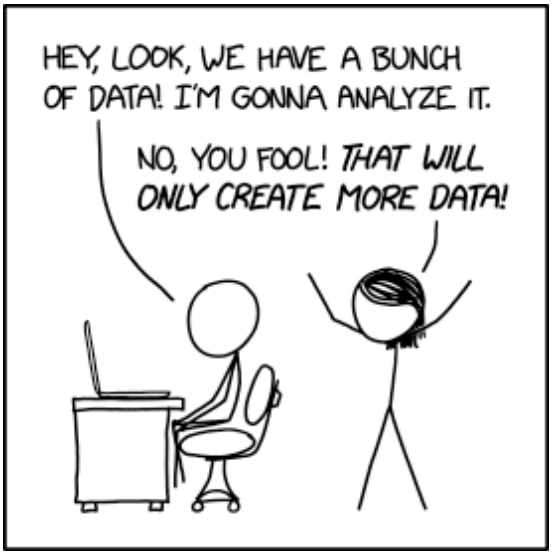
The following courses and resources aided in the creation of this document:

Learning Python for Data Analysis and Visualization by Jose Portilla
https://www.udemy.com/learning-python-for-data-analysis-and-visualization/

Python for Financial Analysis and Algorithmic Trading by Jose Portilla
https://www.udemy.com/python-for-finance-and-trading-algorithms/

Python for Data Science and Machine Learning Bootcamp by Jose Portilla
https://www.udemy.com/course/python-for-data-science-and-machine-learning-bootcamp/

Python for Machine Learning & Data Science Masterclass by Jose Portilla
https://www.udemy.com/course/python-for-machine-learning-data-science-masterclass/



https://xkcd.com/2582/

Related content can be found in these companion files:

Python Programming
Python Visualizations
Python Probability & Statistics

Using this guide

Code is written in a fixed-width font:

`Red` indicates required syntax/new concepts
`Green` represents arbitrary variable assignments
`Blue` indicates program output

Cells may be shaded in light-blue for illustration purposes.

Notes in brown indicate older, deprecated methods.

All code is Python 3.11.0, NumPy 1.24.1 and pandas 1.5.2 unless otherwise noted.

What's What

NumPy – fundamental package for scientific computing & working with array data
Pandas – offers high-performance data structures (Series, DataFrames), built-in visualization, file reading tools
Matplotlib – data visualization package
Seaborn Libraries – specialized visualizations (heatmaps et al)
Beautiful Soup – a web-scraping tool
SciPy – a scientific/technical computing library built on NumPy
SciKit-Learn – a machine learning library

Vocabulary

numpy: An **Array** is numpy's basic data structure. A one-dimensional array is called a *vector*, while a 2D array is a *matrix* (although it is possible for a matrix to have just one row or one column)

pandas: A **Series** is built on top of an array, allowing you to label the data and index it formally
A **DataFrame** is built on top of Series, and is essentially many series put together with different column names but sharing the same index.
Arrays are numpy data types while Series and DataFrame are pandas data types. They have different available methods and attributes.

Jupyter Notebook Tips & Tricks

`Shift+Enter`	run the current cell and move to the next cell (create one if needed)
`Ctrl+Enter`	run the current cell but remain inside it
`pwd`	print working directory
`ls`	print a list of current directory contents
`Ctrl+/`	to comment out a selection of text in a Code cell, repeat to un-comment

To determine what version of Python is running (in jupyter and elsewhere):

```
import sys
print(sys.version)
```

To open a website from inside a Jupyter notebook:

```
import webbrowser
webbrowser.open('https://www.python.org/')
```

To play a YouTube video inside a Jupyter notebook (video owner must permit playing on other websites):

```
from IPython.display import YouTubeVideo
YouTubeVideo('J0Aq44Pze-w')
```

NUMPY *rhymes with "some pie", not "grumpy"*
Documentation https://numpy.org/doc/stable/index.html
Reference https://numpy.org/doc/stable/reference/index.html

From the docs: NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

Standard Import

```
import numpy as np
```
`as np` saves us from writing "numpy" each time

```
np.__version__
```
to see what version of numpy has loaded (optional)

Never do this! `from numpy import *`
For one, it pollutes the global namespace and makes it harder to trace where variables have been defined, and two, it overwrites Python's built-in *sum* function. Use **import numpy as np** and **np.sum** to sum the contents of a numpy array.

Creating Arrays
A common way to create arrays is to cast them from Python lists:

```
my_list = [1, 2, 3]
```
here every element is an integer

```
my_matrix = [[1,2,3],[4,5,6],[7,8,9]]
```
a list of lists, all the same size and data type

```
np.array(my_list)
array([1, 2, 3])
```
creates a 1-dimensional array from a list

```
np.array(my_matrix)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```
creates a 2-dimensional array from a list of lists

```
np.array(([1,2,3], [4,5,6]))
```
creating 2D arrays from scratch requires two parentheses or brackets

```
np.array([[1,2,3], [4,5,6]])
array([[1, 2, 3],
       [4, 5, 6]])
```
these two statements are equivalent

Array Data Types
Unlike Python lists, numpy arrays can contain only *one* type of element. This is one reason numpy is so fast!

```
arr1 = np.array([1,2,3])
arr1
array([1, 2, 3])

arr1.dtype
dtype('int32')
```
In `arr1`, every element is an integer.

```
arr2 = np.array([1,2.5,3])
arr2
array([ 1. ,  2.5,  3. ])

arr2.dtype
dtype('float64')
```
In `arr2`, every element becomes a float.

```
type(arr1)
numpy.ndarray
```
numpy arrays are `ndarray` objects – short for *N-dimensional array*

For more info: https://numpy.org/doc/stable/reference/arrays.ndarray.html

Built-in Array Construction Methods

See https://numpy.org/doc/stable/reference/routines.array-creation.html
and https://numpy.org/doc/stable/user/basics.creation.html

```
np.arange([start,]stop[,step])
```
returns evenly spaced values within a given interval

```
np.arange(0,10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```
like Python `range()`, start is inclusive, stop is exclusive

```
np.arange(0,11,2)
array([ 0,  2,  4,  6,  8, 10])
```

```
np.zeros(5)
array([0., 0., 0., 0., 0.])
```
pass one value as the argument for one-dimensional vectors

```
np.ones((4,4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```
pass a tuple of values as the argument for multi-dimesional matrices (note the double parentheses)

```
np.eye(4)
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```
called the "identity array" (useful in array arithmetic)
takes only one argument as it always produces a square matrix

```
np.linspace(0,10,3)
array([ 0.,  5., 10.]))
```
returns evenly spaced values over a specified interval
unlike `arange()`, the upper bound is *inclusive*

```
np.linspace(0,10,20)
array([ 0.        ,  0.52631579,  1.05263158,  1.57894737,  2.10526316,
        2.63157895,  3.15789474,  3.68421053,  4.21052632,  4.73684211,
        5.26315789,  5.78947368,  6.31578947,  6.84210526,  7.36842105,
        7.89473684,  8.42105263,  8.94736842,  9.47368421, 10.        ])
```

```
np.linspace(0,10,21)
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
        5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5, 10. ])
```

```
np.empty(5)
np.empty((3,4))
```
these resemble zeros arrays, but values are uninitialized & arbitrary (random)
see https://numpy.org/doc/stable/reference/generated/numpy.empty.html

Random Number Arrays

Recent improvements to random number generation are explained here:
https://numpy.org/doc/stable/reference/random/index.html

Generator Objects

See https://numpy.org/doc/stable/reference/random/generator.html

Newer BitGenerators create pseudo random numbers in ways that are statistically better than the RandomState generator used in the past.

```
rng = np.random.default_rng()      construct a Generator object
rng = np.random.default_rng(42)    construct a Generator object with an arbitrary seed value
```

Passing a seed value lets us generate the same set of "random" values, in case we need reproducible results.

```
rng.random(4)                  returns an array of random samples from a uniform distribution over [0, 1)
array([0.77395605, 0.43887844, 0.85859792, 0.69736803])

rng.standard_normal(10)        returns samples from the "standard normal" distribution [σ = 1]
                               unlike random which is uniform, values closer to zero are more likely to appear
array([-1.95103519, -1.30217951,  0.1278404 , -0.31624259, -0.01680116,
       -0.85304393,  0.87939797,  0.77779194,  0.0660307 ,  1.12724121])

rng.integers(1,100,10)         returns random integers from low (inclusive) to high (exclusive)
array([54, 44, 45, 23, 10, 55, 88,  7, 85, 82], dtype=int64)

rng.random((4,3))              returns a 2D array. requires two sets of parentheses.
array([[0.6316644 , 0.75808774, 0.35452597],
       [0.97069802, 0.89312112, 0.7783835 ],
       [0.19463871, 0.466721  , 0.04380377],
       [0.15428949, 0.68304895, 0.74476216]])

rng.integers(1,100,(4,3))      returns a 2D array
array([[37, 96, 41],
       [33, 90, 37],
       [ 8, 47, 79],
       [19, 46, 13]], dtype=int64)
```

Backward Compatibility with RandomState

Although pseudo random number generation with RandomState is not as statistically sound as with more recent BitGenerators, the following legacy code still works:

```
np.random.seed(42)             seeds the RandomState generator (optional, seed value is arbitrary)

np.random.rand(4)              returns an array of random samples from a uniform distribution over [0, 1)
array([0.37454012, 0.95071431, 0.73199394, 0.59865848])

np.random.randn(10)            returns samples from the "standard normal" distribution [σ = 1]
array([-0.23415337, -0.23413696,  1.57921282,  0.76743473, -0.46947439,
        0.54256004, -0.46341769, -0.46572975,  0.24196227, -1.91328024]))

np.random.randint(1,100,10) returns random integers from low (inclusive) to high (exclusive)
array([89, 49, 91, 59, 42, 92, 60, 80, 15, 62])
```

Array Attributes and Methods

```
arr = np.array([20,7,24,2,18,9,3,38,19,14])
array([20,  7, 24,  2, 18,  9,  3, 38, 19, 14])

arr.max()       arr.argmax()        argmax & argmin return index positions
38              7
arr.min()       arr.argmin()
2               3
```

Note: if an array contains **nan** values, `arr.max()` and `arr.min()` won't work. Pandas handles this better.

```
arr.size                    returns the number of elements in the array, including nan and inf values
10                          there is no "count" method

arr.shape                   describes the shape of the array (rows, columns)
(10,)                       the lack of a second tuple value indicates a one-dimensional array

arr.dtype                   describes the data type of the array
dtype('int32')

arr.astype('float64')       recasts the array in the given data type
array([20.,  7., 24.,  2., 18.,  9.,  3., 38., 19., 14.])
```

Array Arithmetic

```
arr = np.array([[0,1,2], [7,8,9]])      note the double parentheses/brackets
arr
array([[0, 1, 2],
       [7, 8, 9]])
```

Adding arrays:

```
arr + arr
array([[ 0,  2,  4],
       [14, 16, 18]])
```

Multiplying arrays:

```
arr * arr
array([[ 0,  1,  4],
       [49, 64, 81]])
```

Subtracting arrays:

```
arr - arr
array([[0, 0, 0],
       [0, 0, 0]])
```

Dividing arrays: (Float return)

```
arr / arr
array([[ nan,   1.,   1.],
       [  1.,   1.,   1.]])
```

This throws a warning about dividing by zero, but returns *nan*

Array Arithmetic with Scalars

```
arr / 2
array([[0. , 0.5, 1. ],
       [3.5, 4. , 4.5]])

1 / arr
array([[        inf, 1.        , 0.5       ],
       [0.14285714, 0.125     , 0.11111111]])
```

This also throws a warning, but instead returns infinity

```
arr**3
array([[  0,   1,   8],
       [343, 512, 729]], dtype=int32)
```

Broadcasting

The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations. There are, however, cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computation.
See https://numpy.org/doc/stable/user/basics.broadcasting.html

```
arr1 = np.array([1,2,3,4])
arr1
array([1, 2, 3, 4])
```

```
arr2 = np.array([[5],[6],[7]])
arr2
array([[5],
       [6],
       [7]])
```

Adding and multiplying shouldn't be possible since these arrays have different numbers of elements, and yet:

```
arr1 + arr2
array([[ 6,  7,  8,  9],
       [ 7,  8,  9, 10],
       [ 8,  9, 10, 11]])
```

```
arr1 * arr2
array([[ 5, 10, 15, 20],
       [ 6, 12, 18, 24],
       [ 7, 14, 21, 28]])
```

When adding arrays of different shapes, we can think of one array "tiling" across the other.

Broadcasting has its limitations. We can't add `array([1, 2, 3, 4])` to `array([5, 6, 7])` and we can't add `array([1, 2, 3, 4])` to `array([[5, 2], [6, 3], [7, 4]])`

Axis Logic

The shape of an array is given as a tuple of (*rows*, *columns*).
axis=0 points to the zero-index in the tuple, and considers row slices together, while **axis=1** considers columns.

For example, `arr.sum(axis=1)` returns

```
array([(col 0 row 0 + col 1 row 0 + col 2 row 0 + ...),
       (col 0 row 1 + col 1 row 1 + col 2 row 1 + ...),
       (col 0 row 2 + col 1 row 2 + col 2 row 2 + ...)])
```

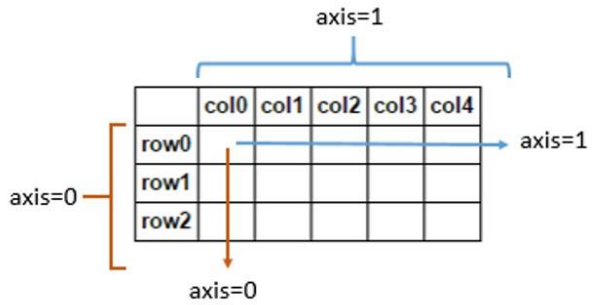


Image source: https://stackoverflow.com/questions/25773245/ambiguity-in-pandas-dataframe-numpy-array-axis-definition

We provide specific examples in the next section.

Summary Statistics on Arrays

```
arr = np.array([[1,2,3],[4,5,6],[7,8,9]])
arr
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

`arr.sum()` returns `45`

`arr.sum(axis=0)` returns `array([12,15,18])`	sums along vertical axes. Can also use arr.sum(0)
`arr.sum(axis=1)` returns `array([ 6,15,24])`	sums along horizontal axes. Can also use arr.sum(1)
`arr.mean()` returns `5.0`	there are no "median" or "mode" methods
`arr.var()` returns `6.666666666666667`	population variance
`arr.var(ddof=1)` returns `7.5`	sample variance (uses *n*-1)
`arr.std()` returns `2.581988897471611`	population standard deviation
`arr.std(ddof=1)` returns `2.7386127875258306`	sample standard deviation (uses *n*-1)
`arr.min()` returns `1`, `.max()` returns `9`	(handles null values poorly)
`arr.argmin()` returns `0`, `.argmax()` returns `8`	returns index positions

`arr.size` returns `9` Note: .size is an attribute. There is no "count" method. Will incl **nan** and **inf** values

Mathematical Functions

See https://numpy.org/doc/stable/reference/routines.math.html

These are available by passing an array into the function. Operations are performed element-wise.

`np.sqrt(arr)`	square-root function
`np.cbrt(arr)`	cube-root function
`np.exp(arr)`	exponential (e^)
`np.sin(arr)`	trigonometric functions
`np.log(arr)`	natural logarithm
`np.abs(arr)`	absolute value
`np.reciprocal(arr)`	*Note: This function is not designed to work with integers*. See docs.
`np.sign(arr)`	returns an array of ones/negative-ones based on the sign of each element
`np.cumsum(arr)`	returns the cumulative sum of the elements along a given axis
`np.cumprod(arr)`	returns the cumulative product of the elements along a given axis
`np.nancumsum(arr)`	nulls are treated as zeros
`np.nancumprod(arr)`	nulls are treated as ones

Consider:

```
arr = np.array([[1,2], [3,np.nan], [5,6]])
np.cumsum(arr)
array([ 1.,  3.,  6., nan, nan, nan])

np.nancumsum(arr)
array([ 1.,  3.,  6.,  6., 11., 17.])

np.nancumsum(arr, axis=0)
array([[1., 2.],
       [4., 2.],
       [9., 8.]])
```

Rounding Functions
See https://numpy.org/doc/stable/reference/routines.math.html#rounding

For values exactly halfway between rounded decimal values, NumPy rounds to the nearest even value.
Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc.

```
arr = np.array([1.234,5.555,6.789])
```

```
np.around(arr)
array([1., 6., 7.])
```
returns an array the same size and type as `arr`, with each element rounded to the specified number of decimals. **np.round()** is an alias for np.around()

```
np.around(arr,2)
array([1.23, 5.56, 6.79])
```

```
np.rint(arr)
array([1., 6., 7.])
```
rounds elements of the array to the nearest integer

```
np.rint(arr).astype('int32')
array([1, 6, 7])
```

Binary Functions (require two arrays passed in as arguments)

`np.add(A,B)`	returns sum of matching values of two arrays, broadcasting if necessary
`np.multiply(A,B)`	returns product of matching values of two arrays, broadcasting if necessary
`np.maximum(A,B)`	returns maximum between matching values of two arrays
`np.minimum(A,B)`	returns minimum between matching values of two arrays

Reshaping Arrays
See https://numpy.org/doc/stable/reference/generated/numpy.ndarray.reshape.html

```
arr = np.arange(1,13)
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])

arr.reshape(3,4)
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

returns an array containing the same data in a new shape
but does *not* change the array in place
Equivalent to `numpy.reshape(arr,(3,4))`

From the docs:
Unlike the free function **numpy.reshape**, this method on ndarray allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10,11)` is equivalent to `a.reshape((10,11))`.

```
arr.reshape(1,12)
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]])
```
double brackets indicate a 2D array

```
arr.reshape(12,1)
array([[ 0],
       [ 1],
       [ 2],
       [ 3],
       [ 4],
       [ 5],…

arr.shape
(12,)
arr.reshape(12,1).shape
(12,1)
```

Flattening Arrays
See https://numpy.org/doc/stable/reference/generated/numpy.ndarray.flatten.html

```
arr2 = np.array([[1,2], [3,4], [5,6]])
arr2
array([[1, 2],
       [3, 4],
       [5, 6]])

arr2.flatten()
array([1, 2, 3, 4, 5, 6])

arr2.flatten('F')
array([1, 3, 5, 2, 4, 6])
```

Parameters: `'C'` (the default) means to flatten in row-major (C-style) order.
`'F'` means to flatten in column-major (Fortran- style) order. Refer to the docs for additional options.

Array Slices

Array slices are views of an array, not copies, which avoids memory problems.
As such, arrays are modified in place by slice operations.

```
arr = np.arange(9)
arr
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8])

arr_slice = arr[1:-2:2]      grab every element from index 1 (inclusive) to the second-to-last (exclusive),
arr_slice                    in steps of 2
array([1, 3, 5])

arr_slice[:] = 99            change every element in the slice
arr_slice
array([99, 99, 99])

arr                          show the original array
array([ 0, 99,  2, 99,  4, 99,  6, 7,  8])
```

Note that the changes *also* occur in our original array.

To get a true copy, we need to be explicit:

```
arr_copy = arr.copy()
```

Changes made to `arr_copy` won't be reflected in `arr`, only in `arr_copy`.

Reassign values with broadcasting

```
arr[2:5] = 42                we can't do this with Python lists!
arr
array([ 0, 99, 42, 42, 42, 99,  6, 7,  8])
```

Slicing a 2D Array

```
mat = np.array(([5,10,15],[20,25,30],[35,40,45]))
mat
array([[ 5, 10, 15],
       [20, 25, 30],
       [35, 40, 45]])
```

slicing format follows `mat[row,col]` or `mat[row][col]`

Grab the row at index 1:

```
mat[1]
array([20, 25, 30])
```

Grab the column at index 1:

```
mat[:,1]
array([10, 25, 40])
```

Grab an individual element:

`mat[1,0]` or `mat[1][0]`

```
20
```

Grab a 2x2 slice from the top right corner:

```
mat[:2,1:]
array([[10, 15],
       [25, 30]])
```

Reassign values within a slice

```
mat[:2,1:] = 77
mat
array([[ 5, 77, 77],
       [20, 77, 77],
       [35, 40, 45]])
```

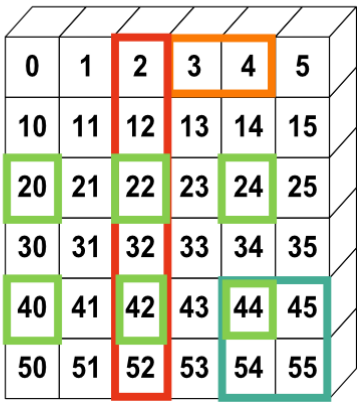
Slicing a 2D Array, cont'd

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



Source: http://www.scipy-lectures.org/_images/numpy_indexing.png

Fancy Indexing

"Fancy" indexing allows a selection *in any order* using embedded brackets (essentially indexing arrays using arrays)

```
arr = np.arange(12).reshape(3,4)
arr
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

arr[[2,1]]                    recall that arr[2,1] just returns 9
array([[ 8,  9, 10, 11],
       [ 4,  5,  6,  7]])
```

Conditional Selection

```
arr = np.arange(1,9)
arr
array([1, 2, 3, 4, 5, 6, 7, 8])

arr > 4                       returns a Boolean array
array([False, False, False, False,  True,  True,  True,  True])

arr[arr > 2]                  casts a Boolean array against the original
array([3, 4, 5, 6, 7, 8])

arr                           so far, the original array is unchanged
array([1, 2, 3, 4, 5, 6, 7, 8])

x=2
arr[arr > x]                  supports variables
array([3, 4, 5, 6, 7, 8])
```

Available comparison operators:

>	greater than	>=	greater than or equal to
<	less than	<=	less than or equal to
==	equal to	just one = sign is an *assignment* operator	
!=	not equal to		

We can perform conditional selection against a collection of values:

```
primes = [2,3,5,7,11]
np.isin(arr, primes)          returns a Boolean array
array([False,  True,  True, False,  True, False,  True, False])

arr[np.isin(arr, primes)]     casts the Boolean array against the original
array([2, 3, 5, 7])

arr[~np.isin(arr, primes)]    the tilde symbol reverses the Boolean array
array([1, 4, 6, 8])           effectively filtering out selected values
```

Any and All for processing Boolean arrays

```
arr = np.array([True,False,True])
arr.any() returns True
arr.all() returns False
```

Random Choice Arrays

See https://numpy.org/doc/stable/reference/random/generated/numpy.random.Generator.choice.html

```
arr = np.array([20,7,24,2,18,9,3,38,19,14])
rng = np.random.default_rng()

rng.choice(arr,3,replace=False)
array([ 7, 20,  3])                results will vary

rng.choice(5,3,replace=True)       if an int is passed, the random sample is generated from np.arange(a)
array([4, 4, 0], dtype=int64)
```

The legacy RandomState method also works:

```
np.random.choice(arr,3,replace=False)
array([ 9, 24, 18])                results will vary
```

Insert elements into an array

See https://numpy.org/doc/stable/reference/generated/numpy.insert.html

```
arr = np.array([[1,1], [2,2], [3,3]])
arr
array([[1, 1],
       [2, 2],
       [3, 3]])

np.insert(arr, 1, 5)              inserts a 5 before index 1 and flattens the array (but not in place!)
array([1, 5, 1, 2, 2, 3, 3])

np.insert(arr, 1, 5, axis=1)      inserts a 5 before index 1 along the vertical axis (but not in place!)
array([[1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])

np.insert(arr, 1, [5,6,7])
array([1, 5, 6, 7, 1, 2, 2, 3, 3])
```

```
np.insert(arr, 1, [5,6,7], axis=1)
array([[1, 5, 1],
       [2, 6, 2],
       [3, 7, 3]])
```

works when dimensions match

```
np.insert(arr, 1, [5,6], axis=1)
```

throws an error

Append elements to an array

See https://numpy.org/doc/stable/reference/generated/numpy.append.html

```
arr1d = np.array([1,2,3])

np.append(arr1d, 4)
array([1, 2, 3, 4])
```

4 is added to a *copy* of the array; `arr1d` doesn't change

for this reason, `arr1d.append(4)` is not valid code

```
arr2d = np.array([[1,1,1], [2,2,2], [3,3,3]])

np.append(arr2d, [5,6,7])
array([1, 1, 1, 2, 2, 2, 3, 3, 3, 5, 6, 7])
```

elements append to a flattened array

However, when *axis* is specified, *values* must have the correct shape:

```
np.append(arr2d, [[5,6,7]], axis=0)
array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3],
       [5, 6, 7]])

np.append(arr2d, [[5],[6],[7]], axis=1)
array([[1, 1, 1, 5],
       [2, 2, 2, 6],
       [3, 3, 3, 7]])
```

Delete elements from an array

See https://numpy.org/doc/stable/reference/generated/numpy.delete.html

```
arr = np.arange(1,13).reshape(4,3)
arr
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

np.delete(arr,[5,8,11])
array([ 1,  2,  3,  4,  5,  7,  8, 10, 11])
```

deletes elements at *index positions* 5, 8, 11 of the flattened array

```
np.delete(arr,[1,3], axis=0)
array([[1, 2, 3],
       [7, 8, 9]])

np.delete(arr, [1], axis=1)
array([[ 1,  3],
       [ 4,  6],
       [ 7,  9],
       [10, 12]])
```

Array Transposition

See https://numpy.org/doc/stable/reference/generated/numpy.ndarray.transpose.html

```
arr = np.arange(1,13).reshape((3,4))
arr
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

the first row is highlighted to illustrate the dot product below

These are equivalent:

```
arr.T
arr.transpose()
```

these do NOT modify the array in place, and they do not take an "inplace=True" argument. use `arr = arr.transpose()` instead

the second column is highlighted to illustrate the dot product below

```
array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]])
```

Array Dot Products

See https://www.mathsisfun.com/algebra/matrix-multiplying.html for a simple explanation of dot products

```
np.dot(arr,arr.T)
array([[ 30,  70, 110],
       [ 70, 174, 278],
       [110, 278, 446]])
```

30=(1*1)+(2*2)+(3*3)+(4*4)
70=(**1*****5**)+(**2*****6**)+(**3*****7**)+(**4*****8**)
110=(1*9)+(2*10)+(3*11)+(4*12)

```
np.dot(arr.T,arr)
array([[107, 122, 137, 152],
       [122, 140, 158, 176],
       [137, 158, 179, 200],
       [152, 176, 200, 224]])
```

107=(1*1)+(5*5)+(9*9)
122=(1*2)+(5*6)+(9*10)

We can also transpose a 3D matrix:

```
arr3d = np.arange(18).reshape((3,3,2))
```

```
arr3d
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]],

       [[12, 13],
        [14, 15],
        [16, 17]]])
```

```
arr3d.transpose((1,0,2))
array([[[ 0,  1],
        [ 6,  7],
        [12, 13]],

       [[ 2,  3],
        [ 8,  9],
        [14, 15]],

       [[ 4,  5],
        [10, 11],
        [16, 17]]])
```

```
arr3d.transpose((2,0,1))
array([[[ 0,  2,  4],
        [ 6,  8, 10],
        [12, 14, 16]],

       [[ 1,  3,  5],
        [ 7,  9, 11],
        [13, 15, 17]]])
```

Using numpy.where
See https://numpy.org/doc/stable/reference/generated/numpy.where.html

```
A = np.array([1,2,3,4])
B = np.array([100,200,300,400])
condition = np.array([True,True,False,False]) a Boolean array
```

We want to select items from either `A` or `B` depending on the truth value in `condition`.

The slow way: Using a list comprehension

```
answer1 = [(A_val if cond else B_val) for A_val,B_val,cond in zip(A,B,condition)]
answer1
[1, 2, 300, 400] Problems include speed issues and multi-dimensional array issues
```

The numpy.where way:

```
answer2 = np.where(condition,A,B)       follows (test, if true, if false)
answer2
array([  1,   2, 300, 400])
```

Using numpy.where for 2D manipulation:

```
rng = np.random.default_rng(42)
arr = rng.standard_normal([4,4])
arr
array([[ 0.30471708, -1.03998411,  0.7504512 ,  0.94056472],
       [-1.95103519, -1.30217951,  0.1278404 , -0.31624259],
       [-0.01680116, -0.85304393,  0.87939797,  0.77779194],
       [ 0.0660307 ,  1.12724121,  0.46750934, -0.85929246]])

np.where(arr<0, 0, arr)  where array is less than zero, make that value zero, otherwise leave as is
array([[0.30471708, 0.        , 0.7504512 , 0.94056472],
       [0.        , 0.        , 0.1278404 , 0.        ],
       [0.        , 0.        , 0.87939797, 0.77779194],
       [0.0660307 , 1.12724121, 0.46750934, 0.        ]])
```

Sorting arrays
See https://numpy.org/doc/stable/reference/generated/numpy.ndarray.sort.html

```
arr = np.array([[2,3,1,4],[1,4,3,2],[3,2,4,1]])
arr2 = arr.copy()            to retain an unchanged array for step 2
arr
array([[2, 3, 1, 4],
       [1, 4, 3, 2],
       [3, 2, 4, 1]])

arr.sort()                   sorts each row individually, in place
arr
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]])

arr2.sort(axis=0)            sorts array elements vertically in place
arr2
array([[1, 2, 1, 1],
       [2, 3, 3, 2],
       [3, 4, 4, 4]])
```

PANDAS

Documentation https://pandas.pydata.org/docs

Standard Imports

```
import numpy as np                          we'll continue to use NumPy tools
import pandas as pd                         as pd is common practice
from pandas import Series,DataFrame         saves us from typing 'pd.Series' and 'pd.DataFrame' each time
pd.__version__                              to see what version of pandas has loaded (optional)
```

WORKING WITH SERIES

See https://pandas.pydata.org/docs/user_guide/dsintro.html#series

A pandas Series is a one-dimensional array of values and their labels.
The basic method to create a series is to call `Series(data, index=index)`

Creating a Series

```
ser = Series([3,6,9,12])
ser                 "ser" is an arbitrary object name. Try to pick names that best represent the data being stored!
0     3
1     6
2     9
3    12
dtype: int64
```

When no index is passed, pandas assigns a numerical range.

```
ser.values          shows the values as a NumPy array
array([ 3,  6,  9, 12], dtype=int64)

ser.index           shows the index. ser.keys() also works.
RangeIndex(start=0, stop=4, step=1)
```

Creating a Series with axis labels

```
coins = Series([.01,.05,.10,.25],index=['penny','nickel','dime','quarter'])
coins
penny      0.01
nickel     0.05
dime       0.10
quarter    0.25
dtype: float64
```

Basic operations:

`coins['dime']` returns `0.1`
`coins[2]` returns `0.1` positional index arguments are recognized
`'penny' in coins` returns `True`
`.25 in coins.values` returns `True`
`coins.index` returns `Index(['penny', 'nickel', 'dime', 'quarter'], dtype='object')`

We can set a multilevel index by passing two arrays:

```
ser = Series([3,6,9,12], index=[['A','A','B','B'],['x','y','x','y']])
ser
A  x     3
   y     6
B  x     9
   y    12
dtype: int64
```

We go into much greater detail on multilevel indexes in the DataFrame section on index hierarchy.

Creating a Series from a dictionary

```
d = {'b': 1, 'a': 0, 'c': 2}
Series(d)                    follows insertion order of dict keys
b    1
a    0
c    2
dtype: int64
```

Converting a Series to a Python dictionary

```
coins = Series([.01,.05,.10,.25],['penny','nickel','dime','quarter'])
coin_dict = coins.to_dict()
coin_dict
{'penny': 0.01, 'nickel': 0.05, 'dime': 0.1, 'quarter': 0.25}
```

Passing an index with the dictionary can reload a Series in a new order:

```
coinlabels = ['dime','penny','nickel','quarter','dollar']
coins2 = Series(coin_dict,index=coinlabels)  converts it back in coinlabels order
coins2
dime          0.10
penny         0.01
nickel        0.05
quarter       0.25
dollar         NaN      rather than raise an error, 'dollar' is assigned a null value
dtype: float64
```

Adding two Series together

See https://pandas.pydata.org/docs/reference/api/pandas.Series.add.html

```
ser1, ser2 = Series({'a':1,'b':3,'c':5}), Series({'a':2,'b':4,'c':np.nan,'d':8})
ser1 + ser2              adds items by index. null-value items return a NaN sum value
a    3.0                 if an index value appears in one series but not the other, NaN is returned
b    7.0
c    NaN
d    NaN
dtype: float64
```

Use the .add method to assign fill values:

```
ser1.add(ser2, fill_value=0)
a    3.0
b    7.0
c    5.0
d    8.0
dtype: float64
```

Related methods include Series.sub(), Series.mul() and Series.div()
We go into much greater detail in the section on DataFrame Arithmetic.

Naming Series Indexes

```
coins.index.name = 'Coins'
coins
Coins
penny      0.01
nickel     0.05
dime       0.10
quarter    0.25
dtype: float64
```

puts a label above the index list
ser.values does not have a name attribute

Selecting, Changing Series Entries

```
ser = Series(np.arange(4)*2,index=['A','B','C','D'])
ser
A    0
B    2
C    4
D    6
dtype: int32
```

by index label:
`ser['B']` returns `2`

by index value:
`ser[1]` returns `2`

by index slice:

```
ser[0:2]
A    0
B    2
dtype: int32
```

by a list of index labels:

```
ser[['C','A']]
C    4
A    0
dtype: int32
```

by logic:

```
ser[ser>3]
C    4
D    6
dtype: int32
```

We can *change* values the same way. Changes occur in place and cannot be undone.

```
ser[ser>3] = 10
ser
A     0
B     2
C    10
D    10
dtype: int32
```

Checking for Unique Values and their Counts

```
ser1 = Series(list('seeded'))
ser1.unique()          returns array(['s', 'e', 'd'], dtype=object)
ser1.nunique()         returns 3  the number of unique elements
ser1.value_counts()    returns a count of the unique elements as another Series
e    3
d    2
s    1
dtype: int64
```

See the DataFrame value_counts section in this document for more on available parameters, etc.

Removing Elements
See https://pandas.pydata.org/docs/reference/api/pandas.Series.drop.html
and https://pandas.pydata.org/docs/reference/api/pandas.Series.pop.html

```
ser.drop('b')          returns a copy of the series with index 'b' removed
ser.drop(['b','d'])    to remove multiple values use a list
ser.pop('b')           returns the item at index 'b' being removed and permanently changes ser
```

Removing Elements Permanently
Most methods return modified copies of the data unless the argument `inplace=True` is passed into the method. This works, but a better technique is to use reassignment. The code is clearer, and there's a possibility that the `inplace` argument will be deprecated in the future.

Good: `ser.drop('b', inplace=True)` **Better:** `ser = ser.drop('b')`

`_ = ser.pop('b')` also works. Assignment suppresses the item return when unneeded.

Rank and Sort
See https://pandas.pydata.org/docs/reference/api/pandas.Series.sort_index.html
and https://pandas.pydata.org/docs/reference/api/pandas.Series.sort_values.html
and https://pandas.pydata.org/docs/reference/api/pandas.Series.rank.html

```
ser1 = Series([4,5,3],index=['C','A','B'])
ser2 = Series([4,4,5,3],index=['C','A','B','D'])

ser1                                ser2
C    4                              C    4
A    5                              A    4
B    3                              B    5
dtype: int64                        D    3
                                    dtype: int64
```

Sort by index using `.sort_index`

```
ser1.sort_index()
A    5                              Note: this does NOT sort ser1 in place
B    3                              for that use ser1 = ser1.sort_index()
C    4
dtype: int64
```

Sort by value using `.sort_values`

```
ser1.sort_values()
B    3                              Note: this does NOT sort ser1 in place
C    4                              for that use ser1 = ser1.sort_values()
A    5
dtype: int64
```

Rank

```
ser1.rank()        ser2.rank()      assigns numerical values to keys based on value order
C    2.0           C    2.5         ties are resolved with half-steps
A    3.0           A    2.5
B    1.0           B    4.0
dtype: float64     D    1.0
                   dtype: float64
```

WORKING WITH DATAFRAMES
See https://pandas.pydata.org/docs/user_guide/dsintro.html#dataframe

Constructing a DataFrame from a numpy array

```
df = DataFrame(np.arange(12).reshape(4,3))
df
```

	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11

`df` is constructed by casting a 4-row by 3-col numpy array as a pandas DataFrame, pre-filled with values 0-11.
Here the index defaults to [0,1,2,3], the columns to [0,1,2]. Alternatively we can pass in explicit index and column labels:

```
df = DataFrame(np.arange(12).reshape((4,3)),
               index=['a','b','c','d'],
               columns=['Col1','Col2','Col3'])
df
```

	Col1	Col2	Col3
a	0	1	2
b	3	4	5
c	6	7	8
d	9	10	11

Constructing a DataFrame from a dictionary

```
d = {'City':['SF','LA','NYC'],'Population':[837000,3880000,8400000]}
df_cities = DataFrame(d)
df_cities
```

	City	Population
0	SF	837000
1	LA	3880000
2	NYC	8400000

When passing a dictionary, all value arrays must be of the same length. That is:

`DataFrame({'east':['NY','DC'],'west':['LA','SF','LV']})` and
`DataFrame({'east':np.array(['NY','DC']),'west':np.array(['LA','SF','LV'])})`

won't work, but we'll see that

`DataFrame({'east':Series(['NY','DC']),'west':Series(['LA','SF','LV'])})` does.

By default dictionary keys are oriented to columns. To orient keys to rows use:

```
df_cities = DataFrame.from_dict(d, orient='index')
```

	0	1	2
City	SF	LA	NYC
Population	837000	3880000	8400000

Later we'll see that `df_cities = DataFrame(d).T` also works.

Caution: if dictionary values are scaler (not collections) we must also pass an index value:

```
df = DataFrame({'a': 1, 'b': 2}, index=[0])
```

Constructing a DataFrame from a Series

There are several ways to do this depending on the desired result.

```
ser1 = Series([837000,3880000,8400000], index=['SF','LA','NYC'],
              name='Population')
ser2 = Series([16,87,10,6], index=['SF','LA','NYC','BOS'],
              name='Elevation')
```

```
ser1
SF      837000
LA     3880000
NYC    8400000
Name: Population, dtype: int64
```

```
ser2
SF     16
LA     87
NYC    10
BOS     6
Name: Elevation, dtype: int64
```

Convert a Series directly to a DataFrame:

```
df = ser1.to_frame()
df
```

	Population
SF	837000
LA	3880000
NYC	8400000

Create a DataFrame from a list of Series:

```
df = DataFrame([ser1,ser2])
df
```

	SF	LA	NYC	BOS
Population	837000.0	3880000.0	8400000.0	NaN
Elevation	16.0	87.0	10.0	6.0

The assignment of NaN to Boston's Population causes the dtype of the entire DataFrame to become a float.

Create a DataFrame where each Series is a column:

```
df = pd.concat([ser1,ser2], axis=1)
df
```

	Population	Elevation
SF	837000.0	16
LA	3880000.0	87
NYC	8400000.0	10
BOS	NaN	6

The assignment of NaN to Boston's Population causes the dtype of just that column to become a float.

We can use pd.concat() to add a Series to an existing DataFrame:

```
df = ser1.to_frame()
df = pd.concat([df,ser2], axis=1)
df
```

	Population	Elevation
SF	837000.0	16
LA	3880000.0	87
NYC	8400000.0	10
BOS	NaN	6

We go into much greater detail in the CONCATENATE section in this document.

Constructing a DataFrame from a random array

See https://numpy.org/doc/stable/reference/random/index.html

```
rng = np.random.default_rng(101)
```

seeded for reproducibility, value is arbitrary

```
df = DataFrame(rng.random((4,3))*20, index='A B C D'.split(),
               columns='X Y Z'.split())
df
```

	X	Y	Z
A	18.870650	7.188421	15.696108
B	11.825564	5.886571	18.454514
C	17.386631	7.282769	19.463536
D	4.490487	16.109917	13.617925

a random array from a uniform distribution over [0:20)

```
df = DataFrame(rng.integers(0,21,(4,3)),'A B C D'.split(),'X Y Z'.split())
df
```

	X	Y	Z
A	2	9	11
B	0	15	18
C	14	12	20
D	8	8	7

a random array of integers from 0 to 20

```
df = DataFrame(rng.standard_normal((4,3)),'A B C D'.split(),'X Y Z'.split())
df
```

	X	Y	Z
A	-0.340127	-1.207603	-0.091054
B	0.853924	0.193218	-0.002666
C	-1.409222	0.501016	0.485731
D	1.341266	1.489053	0.688583

a random array from a standard normal distribution [σ = 1]
unlike *random* which is uniform, values closer to zero are more likely to appear

Get column and index labels

```
df.columns
```

remember to omit parentheses when calling attributes

```
Index(['X', 'Y', 'Z'], dtype='object')
```

```
df.index
Index(['A', 'B', 'C', 'D'], dtype='object')
```

EXPLORATORY DATA ANALYSIS

Display a specific number of rows

```
df.head()    retrieves the first 5 rows
df.head(3)   retrieves the first 3 rows
df.tail()    retrieves the last 5 rows
```

Display a random collection of rows

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sample.html

```
df.sample()                  retrieves a row at random
df.sample(n)                 retrieves n rows at random, unsorted, defaults to "replace=False"
df.sample(frac=0.2)          retrieves 20% of the dataset
df.sample(random_state=42)   sets an arbitrary seed for reproducibility
```

Selecting columns

A pandas DataFrame is essentially a collection of Series that all share a common index. To select an individual column or collection of columns we use bracket notation:

Using the random set DataFrame 'df' above:

```
df['Z']
A   -0.091054
B   -0.002666
C    0.485731
D    0.688583
Name: Z, dtype: float64
```

```
type(df['Z'])
pandas.core.series.Series

type(df)
pandas.core.frame.DataFrame
```

Quick note: although `df.Z` is valid SQL-like syntax for grabbing a column, it is not recommended as it may get confused with pandas built-in methods.

To select multiple columns, pass a list:

```
df[['X','Z']]
```

	X	Z
A	-0.340127	-0.091054
B	0.853924	-0.002666
C	-1.409222	0.485731
D	1.341266	0.688583

Creating a new column

```
df['new'] = df['X'] + df['Y']
df
```

	X	Y	Z	new
A	-0.340127	-1.207603	-0.091054	-1.547731
B	0.853924	0.193218	-0.002666	1.047142
C	-1.409222	0.501016	0.485731	-0.908207
D	1.341266	1.489053	0.688583	2.830319

Columns can also be added by reindexing – see later section.

Removing a column with drop
See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.drop.html

When massaging data it helps to save a copy of the original:
```
df_save = df.copy()
```

```
df.drop('new', axis=1)
```
without the axis argument it looks to remove a row named 'new'

	X	Y	Z
A	-0.340127	-1.207603	-0.091054
B	0.853924	0.193218	-0.002666
C	-1.409222	0.501016	0.485731
D	1.341266	1.489053	0.688583

Returns a copy of the DataFrame with the selected column(s) removed, does not change `df` in place.
Accepts a collection of columns in any order.

Removing a column with pop
See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pop.html

```
df.pop('new')
```
does not take an axis argument

```
A   -1.547731
B    1.047142
C   -0.908207
D    2.830319
Name: new, dtype: float64
```

Returns the column being removed, and permanently affects `df`. Accepts only one column label, not a collection.

For the next section we need to revert back to the original DataFrame:
```
df = df_save.copy()
```

Permanently removing a column
Either of these methods will work:

```
df = df.drop('new', axis=1)
df.drop('new', axis=1, inplace=True)
df.pop('new') or _ = df.pop('new') use assignment to suppress the column return
del df['new']
```

Selecting rows
See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.loc.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iloc.html

These methods use bracket notation and follow the pattern `[row_indexer,col_indexer]`

```
df.loc['B'] or df.iloc[1]    use loc to search by row name, iloc to search by index position
X    0.853924
Y    0.193218
Z   -0.002666                          Note: df.ix[] has been deprecated
Name: B, dtype: float64                 in favor of df.loc[] and df.iloc[]
```

Selecting subsets of rows and columns
`df.loc[['A','C'],['X','Z']] or df.iloc[[0,2],[0,2]]`

	X	Z
A	-0.340127	-0.091054
C	-1.409222	0.485731

Selecting slices of rows and columns Be careful – `loc` and `iloc` behave differently here!

`df.loc['A':'C','X':'Z']`

	X	Y	Z
A	-0.340127	-1.207603	-0.091054
B	0.853924	0.193218	-0.002666
C	-1.409222	0.501016	0.485731

`df.iloc[0:2,0:2]`

	X	Y
A	-0.340127	-1.207603
B	0.853924	0.193218

Selecting a single value
See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.at.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iat.html

```
df.at['B','Y'] or df.iat[1,1]
0.19321811565038277
```

Similar to `df.loc['B','Y']` and `df.iloc[1,1]`, but more performative.

For more info: https://pandas.pydata.org/docs/user_guide/indexing.html#fast-scalar-value-getting-and-setting

Conditional Selection

```
rng = np.random.default_rng(101)        use this seed to recreate the following example
df = pd.DataFrame(rng.integers(-9,10,(3,3)), list('ABC'),list('XYZ'))
df
```

	X	Y	Z
A	-4	8	4
B	-3	-7	5
C	-2	2	2

Grabbing a row based on min/max values

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.idxmax.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.idxmin.html

```
df.loc[df['Z'].idxmax()]
X   -3
Y   -7
Z    5
Name: B, dtype: int64
```

.idxmin() and .idxmax() return the first row that contains the desired value, in this case row B

```
df.loc[df[['Z']].idxmax()]
```

calling 'Z' with double-brackets returns a DataFrame

	X	Y	Z
B	-3	-7	5

Selections based on comparison operators

```
df > 0
```

returns a Boolean matrix

	X	Y	Z
A	False	True	True
B	False	False	True
C	False	True	True

```
df[df['Y'] > 0]
```

this is an example of filtering

	X	Y	Z
A	-4	8	4
C	-2	2	2

```
df[df > 0]
```

returns only values that meet criteria

	X	Y	Z
A	NaN	8.0	4
B	NaN	NaN	5
C	NaN	2.0	2

```
df[df['Y'] > 0]['Z']
A    4
C    2
Name: Z, dtype: int64
```

commands can be stacked

Available comparison operators:

>	greater than	>=	greater than or equal to
<	less than	<=	less than or equal to
==	equal to		just one = sign is an *assignment* operator
!=	not equal to		

Selections based on two conditions

For two conditions we can use & and | with parentheses:

```
df[(df['Y'] > 0) & (df['Z'] < 3)]
```

	X	Y	Z
C	-2	2	2

```
df[(df['X'] == -4) | (df['Y'] < -4)]
```

	X	Y	Z
A	-4	8	4
B	-3	-7	5

Python's `and` and `or` operators don't work here, because the truth value of a Boolean Series is ambiguous.

Selections based on categorical data

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.isin.html

```
df1 = DataFrame({'weather':['sunny','rainy','sunny','cloudy'],
                 'traffic':[8,5,9,6]})
df1
```

	weather	traffic
0	sunny	8
1	rainy	5
2	sunny	9
3	cloudy	6

```
options = ['cloudy','rainy']
df1[df1['weather'].isin(options)]
```

	weather	traffic
1	rainy	5
3	cloudy	6

```
df1[~df1['weather'].isin(options)]
```

	weather	traffic
0	sunny	8
2	sunny	9

The tilde character reverses the Boolean mask.

Summary Statistics on DataFrames

See:

pandas.DataFrame.sum() pandas.DataFrame.min() pandas.DataFrame.var()
pandas.DataFrame.prod() pandas.DataFrame.max() pandas.DataFrame.std()
pandas.DataFrame.mean() pandas.DataFrame.idxmin() pandas.DataFrame.cumsum()
pandas.DataFrame.median() pandas.DataFrame.idxmax() pandas.DataFrame.cumprod()
pandas.DataFrame.mode()

```
arrs = np.array([[1,2,np.nan],[np.nan,3,3],[4,5,6],[4,5,6]])
df = DataFrame(arrs,['A','B','C','D'],['One','Two','Three'])
df
```

	One	Two	Three
A	1.0	2.0	NaN
B	NaN	3.0	3.0
C	4.0	5.0	6.0
D	4.0	5.0	6.0

```
df.sum()
One        9.0
Two       15.0
Three     15.0
dtype: float64
```

```
df.sum(axis=1)
A     4.0
B     6.0
C    15.0
D    15.0
dtype: float64
```

```
df.prod()
One        16.0
Two       150.0
Three     108.0
dtype: float64
```

```
df. mean()
One       3.00
Two       3.75
Three     5.00
dtype: float64
```

```
df.median()
One       4.0
Two       4.0
Three     6.0
dtype: float64
```

```
df.mode()
```

	One	Two	Three
0	4.0	5.0	6.0

.mean() and .median() ignore null values (the mean of x and null is x)

```
df.min()
One       1.0
Two       2.0
Three     3.0
dtype: float64
```

```
df.idxmin()
One       A
Two       A
Three     B
dtype: object
```

.idxmin() returns the index of the lowest value
.max() and .idxmax() work as expected

```
df['One'].var()          returns 3.0    sample variance (divisor = n-1)
df['One'].var(ddof=0)    returns 2.0    population variance (divisor = n)
df['One'].std()          returns 1.732… sample standard deviation
df['One'].std(ddof=0)    returns 1.414… population standard deviation
```

```
df.cumsum()
```

	One	Two	Three
A	1.0	2.0	NaN
B	NaN	5.0	3.0
C	5.0	10.0	9.0
D	9.0	15.0	15.0

```
df.cumprod()
```

	One	Two	Three
A	1.0	2.0	NaN
B	NaN	6.0	3.0
C	4.0	30.0	18.0
D	16.0	150.0	108.0

Redisplays the DataFrame with cumulative sums / products. Pass axis=1 to sum across columns.
Note: .cumsum() lacks a fill_value parameter, and there is no .nancumsum() method on DataFrames

Caution: operations like `df.sum()` do strange things with categorical data, and may throw an error.
Consider using `df.sum(numeric_only=True)`.

Summary Statistics on DataFrames, cont'd

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.describe.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.info.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.count.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.memory_usage.html

`df.describe()` provides useful summary statistics

	One	Two	Three
count	3.000000	4.00	3.000000
mean	3.000000	3.75	5.000000
std	1.732051	1.50	1.732051
min	1.000000	2.00	3.000000
25%	2.500000	2.75	4.500000
50%	4.000000	4.00	6.000000
75%	4.000000	5.00	6.000000
max	4.000000	5.00	6.000000

`df.describe().T` puts column names on the left with stats across the top (see transposing data)

`df.info()` provides info on index and column dtypes, non-null values and memory usage

```
<class 'pandas.core.frame.DataFrame'>
Index: 4 entries, A to D
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   One     3 non-null      float64
 1   Two     4 non-null      float64
 2   Three   3 non-null      float64
dtypes: float64(3)
memory usage: 300.0+ bytes
```

`df.count()` returns a count of non-null values for each column or row

```
One      3
Two      4
Three    3
dtype: int64
```

`df.count(axis=1)`

```
A    2
B    2
C    3
D    3
dtype: int64
```

`df.memory_usage()` returns the memory usage of each column in bytes

```
Index    204
One       32
Two       32
Three     32
dtype: int64
```

pandas.DataFrame.size() returns the number of elements in the DataFrame
pandas.DataFrame.shape() returns a tuple representing the dimensionality of the DataFrame

Unique Values and Value Counts

See https://pandas.pydata.org/docs/reference/api/pandas.Series.unique.html
and https://pandas.pydata.org/docs/reference/api/pandas.Series.nunique.html
and https://pandas.pydata.org/docs/reference/api/pandas.Series.value_counts.html

A DataFrame column is itself a Series, so Series methods apply:

```
df = DataFrame({'num':[4,6,3,5,4,3,4,np.nan],
                'num1':[4,6,3,5,4,3,2,1]})
```

df

	num	num1
0	4.0	4
1	6.0	6
2	3.0	3
3	5.0	5
4	4.0	4
5	3.0	3
6	4.0	2
7	NaN	1

```
df['num'].unique()
array([ 4.,  6.,  3.,  5., nan])
```
returns an array of unique values in index order

```
df['num'].nunique()
4
```
returns the number of unique non-null values

```
df['num'].value_counts()
4.0    3
3.0    2
6.0    1
5.0    1
Name: num, dtype: int64
```
returns the count from highest to lowest, as a Series

```
df['num'].value_counts(ascending=True)
6.0    1
5.0    1
3.0    2
4.0    3
Name: num, dtype: int64
```
returns the count from lowest to highest

```
df['num'].value_counts(sort=False)
4.0    3
6.0    1
3.0    2
5.0    1
Name: num, dtype: int64
```
returns the count in index order (*not* value order)
for value order use .value_counts().sort_index()

```
df['num'].value_counts(dropna=False)
4.0    3
3.0    2
6.0    1
5.0    1
NaN    1
Name: num, dtype: int64
```
includes a count of null values

```
df['num'].value_counts(normalize=True)
4.0    0.428571
3.0    0.285714
6.0    0.142857
5.0    0.142857
Name: num, dtype: float64
```
returns relative frequencies instead of counts

```
df['num'].value_counts(bins=2)
(2.996, 4.5]    5
(4.5, 6.0]      2
Name: num, dtype: int64
```
we can convert continuous variables to categories

Unique Values and Value Counts, cont'd
See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.nunique.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.value_counts.html

```
df.nunique()
num     4
num1    6
dtype: int64
```
returns a Series of values for each column

```
df.value_counts()
num  num1
3.0  3       2
4.0  4       2
     2       1
5.0  5       1
6.0  6       1
dtype: int64
```
returns a multi-level Series containing counts of unique rows in the DataFrame
by default, rows that contain any NA values are omitted from the result

Identifying, removing duplicate rows
See: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.duplicated.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.drop_duplicates.html

```
df = DataFrame({'key1': ['A','A','B','B','B'],'key2': [2, 2, 2, 3, 3]})
df
```

	key1	key2
0	A	2
1	A	2
2	B	2
3	B	3
4	B	3

```
df.duplicated()
0    False
1     True
2    False
3    False
4     True
dtype: bool
```
identifies duplicates, works top-to-bottom, dupes don't need to be adjacent
the first occurrence is marked False

```
df.drop_duplicates()
```
drops full-record duplicates

	key1	key2
0	A	2
2	B	2
3	B	3

```
df.drop_duplicates(['key1'])
```
keeps only the first occurrence of records from 'key1'

	key1	key2
0	A	2
2	B	2

```
df.drop_duplicates(['key1'],keep='last')
```
keeps the last occurrence

	key1	key2
1	A	2
4	B	3

Filtering using between

See: https://pandas.pydata.org/docs/reference/api/pandas.Series.between.html

```
df = DataFrame(np.arange(1,6),columns=['num'])
df
```

	num
0	1
1	2
2	3
3	4
4	5

```
df[df['num'].between(2,4)]
```

high and low values are inclusive by default
works with string values as well

	num
1	2
2	3
3	4

```
df[df['num'].between(2,4,inclusive='neither')]
```

	num
2	3

Filtering by largest & smallest values

See: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.nlargest.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.nsmallest.html

```
df.nlargest(3,'num')
```

returns rows with the top 3 'num' values
columns not specified are returned as well, but not used for ordering
accepts an optional "keep" argument (first/last/all) to handle duplicate values

	num
4	5
3	4
2	3

```
df.nsmallest(3,'num')
```

	num
0	1
1	2
2	3

Transposing data

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.transpose.html

```
df = DataFrame([[1,2,3],[4,5,np.nan]],['A','B'],['One','Two','Three'])
df
```

	One	Two	Three
A	1	2	3.0
B	4	5	NaN

```
df.T
```

shorthand for `df.transpose()`
use `df = df.T` to make transposition permanent

	A	B
One	1.0	4.0
Two	2.0	5.0
Three	3.0	NaN

Sorting by values along either axis

See: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sort_values.html

```
df = DataFrame({'Col1':[3,1,2],'Col2':[2,np.nan,5],'Col3':[1,4,1],
                'Col4':['c','a','b'],'Col5':['c','a','B']}, ['A','B','C'])
```

df

	Col1	Col2	Col3	Col4	Col5
A	3	2.0	1	c	c
B	1	NaN	4	a	a
C	2	5.0	1	b	B

```
df.sort_values('Col1')
```

	Col1	Col2	Col3	Col4	Col5
B	1	NaN	4	a	a
C	2	5.0	1	b	B
A	3	2.0	1	c	c

```
df.sort_values(['Col3','Col1'])
```

	Col1	Col2	Col3	Col4	Col5
C	2	5.0	1	b	B
A	3	2.0	1	c	c
B	1	NaN	4	a	a

```
df.sort_values('Col2')
```

	Col1	Col2	Col3	Col4	Col5
A	3	2.0	1	c	c
C	2	5.0	1	b	B
B	1	NaN	4	a	a

```
df.sort_values('Col2',na_position='first')
```

	Col1	Col2	Col3	Col4	Col5
B	1	NaN	4	a	a
A	3	2.0	1	c	c
C	2	5.0	1	b	B

```
df.sort_values('Col4')
```

	Col1	Col2	Col3	Col4	Col5
B	1	NaN	4	a	a
C	2	5.0	1	b	B
A	3	2.0	1	c	c

```
df.sort_values('Col5')
```

	Col1	Col2	Col3	Col4	Col5
C	2	5.0	1	b	B
B	1	NaN	4	a	a
A	3	2.0	1	c	c

When sorting `Col5`, pandas sorts "ascii-betically", meaning capital letters sort before lowercase letters.
See the deep dive appendix section on Non-traditional sorting using key for a workaround.

Pandas can't intersort numbers and text when sorting along columns, but it can intersort integers and floats.

```
df[['Col1','Col2','Col3']].sort_values('A',axis=1)
```

	Col3	Col2	Col1
A	1	2.0	3
B	4	NaN	1
C	1	5.0	2

Ranking values

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.rank.html

```
df.rank()
```

assigns numerical values to keys based on value order
ties are resolved with half-steps

	Col1	Col2	Col3	Col4	Col5
A	3.0	1.0	1.5	3.0	3.0
B	1.0	NaN	3.0	1.0	2.0
C	2.0	2.0	1.5	2.0	1.0

INDEXING

Setting a named index

See: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.set_index.html

```
arr = np.append(np.arange(1,4),np.arange(11,17)).reshape(3,3).T
df = DataFrame(arr,columns=['num','A','B'])
df
```

	num	A	B
0	1	11	14
1	2	12	15
2	3	13	16

```
df.set_index('num')
```

	A	B
num		
1	11	14
2	12	15
3	13	16

sets an existing DataFrame column as the new index
this *overwrites* the old index
by default `inplace=False`

Use the optional `verify_integrity=True` argument to check for duplicates (which throws a ValueError).

Resetting an index

See: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.reset_index.html

```
df.reset_index()
```

sets a numerical (zero-based) index and sets the old index as a column in the DataFrame
it will apply the original index name if present, otherwise the new column is named "index"
by default `inplace=False`

INDEX HIERARCHY

See: https://pandas.pydata.org/docs/reference/api/pandas.MultiIndex.html

From the pandas user guide on MultiIndex / advanced indexing:
Hierarchical / Multi-level indexing is very exciting as it opens the door to some quite sophisticated data analysis and manipulation, especially for working with higher dimensional data. In essence, it enables you to store and manipulate data with an arbitrary number of dimensions in lower dimensional data structures like Series (1d) and DataFrame (2d).

Constructing a hierarchical index from a list of arrays

See https://pandas.pydata.org/docs/reference/api/pandas.MultiIndex.from_arrays.html

These steps create a *MultiIndex object*:

```
arrays = [['G1','G1','G1','G2','G2'], [1,2,3,1,2]]
mi = pd.MultiIndex.from_arrays(arrays, names=['A', 'B'])
mi
```

passing a `names` parameter is optional

```
MultiIndex([('G1', 1),
            ('G1', 2),
            ('G1', 3),
            ('G2', 1),
            ('G2', 2)],
           names=['A', 'B'])
```

We intentionally did not represent every combination here.

Constructing a hierarchical index from a list of tuples
See https://pandas.pydata.org/docs/reference/api/pandas.MultiIndex.from_tuples.html

```
outer = ['G1','G1','G1','G2','G2']
inner = [1,2,3,1,2]
mi = list(zip(outer,inner))              start with a list of tuples
mi
[('G1', 1), ('G1', 2), ('G1', 3), ('G2', 1), ('G2', 2)]

mi = pd.MultiIndex.from_tuples(mi, names=['A', 'B'])
mi
MultiIndex([('G1', 1),
            ('G1', 2),
            ('G1', 3),
            ('G2', 1),
            ('G2', 2)],
           names=['A', 'B'])
```

We intentionally did not represent every combination here.

Constructing a hierarchical index from the product of two collections
See https://pandas.pydata.org/docs/reference/api/pandas.MultiIndex.from_product.html

An easier way to construct every combination:

```
outer, inner = ['G1','G2'], [1,2,3]
mi = pd.MultiIndex.from_product([outer, inner], names=['A', 'B'])
mi
MultiIndex([('G1', 1),
            ('G1', 2),
            ('G1', 3),
            ('G2', 1),
            ('G2', 2),
            ('G2', 3)],
           names=['A', 'B'])
```

This has the benefit of ensuring every combination is represented.

MultiIndex object attributes

```
mi.levels                    returns the unique labels for each level
FrozenList([['G1', 'G2'], [1, 2, 3]])

mi.codes                     returns integers for each level designating which label at each location
FrozenList([[0, 0, 0, 1, 1, 1], [0, 1, 2, 0, 1, 2]])
```

Changed in version 0.24.0: `MultiIndex.labels` has been renamed to `MultiIndex.codes`

```
mi.names                     returns index level names, if assigned
FrozenList(['A', 'B'])
```

A MultiIndex object is its own data type:

```
type(mi)
pandas.core.indexes.multi.MultiIndex
```

Once we have a MultiIndex, we can use it to build both Series and DataFrames.

Using a MultiIndex when constructing a DataFrame

See https://pandas.pydata.org/docs/reference/api/pandas.MultiIndex.to_frame.html

```
df = mi.to_frame()
df
```

uses MultiIndex levels as columns
column names are set to integers if index levels are unlabeled

		A	B
A	**B**		
G1	**1**	G1	1
	2	G1	2
	3	G1	3
G2	**1**	G2	1
	2	G2	2
	3	G2	3

To collapse the hierarchical index:

```
df = mi.to_frame(index=False)
df
```

	A	B
0	G1	1
1	G1	2
2	G1	3
3	G2	1
4	G2	2
5	G2	3

To reassign column names:

```
df = mi.to_frame(name=['X','Y'])
df
```

		X	Y
A	**B**		
G1	**1**	G1	1
	2	G1	2
	3	G1	3
G2	**1**	G2	1
	2	G2	2
	3	G2	3

use `name=`, not `names=`

Alternatively we can pass a MultiIndex to any of the previous DataFrame construction methods:

```
df = DataFrame(np.arange(4,16).reshape(2,6).T,index=mi,columns=['X','Y'])
df
```

		X	Y
A	**B**		
G1	**1**	4	10
	2	5	11
	3	6	12
G2	**1**	7	13
	2	8	14
	3	9	15

To construct a MultiIndex from an existing DataFrame's values (not its index - for that use `df.index`):

```
mi2 = pd.MultiIndex.from_frame(df)
```

To remove index level names:

```
mi2 = pd.MultiIndex.from_tuples(mi, names=None)
```

Another way to create a multilevel DataFrame is to pass a list of arrays at construction:

```
arrays = [['G1','G1','G1','G2','G2'], [1,2,3,1,2]]
df2 = DataFrame(np.zeros((5,2)),index=arrays,columns=['X','Y'])
```

Renaming index levels

```
df.index.names
FrozenList(['A', 'B'])

df.index.names = ['Group','Num']
df
```

		X	Y
Group	**Num**		
G1	**1**	4	10
	2	5	11
	3	6	12
G2	**1**	7	13
	2	8	14
	3	9	15

Making selections on a multilevel DataFrame

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.loc.html

Select a specific subset:

```
df.loc['G1']
```

	X	Y
Num		
1	4	10
2	5	11
3	6	12

Note that `df['G1']` won't work., and `df.iloc[0]` only returns the first row

Select a specific row:

```
df.loc[('G1',2)]
X     5
Y    11
Name: (G1, 2), dtype: int32
```

similar to `df.loc['G1'].loc[2]`

Select a specific cell:

```
df.loc[('G1',2)]['Y']
11
```

Selecting a cross-section

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.xs.html

```
df.xs('G1')
```

	X	Y
Num		
1	4	10
2	5	11
3	6	12

Equivalent to `df.loc['G1']`

```
df.xs(('G1',2))
X     5
Y    11
Name: (G1, 2), dtype: int32
```

Equivalent to `df.loc[('G1',2)]`

This is where `.xs` becomes useful:

```
df.xs(3,level=1)
```

	X	Y
Group		
G1	6	12
G2	9	15

Equivalent to `df.xs(3,level='Num')`

This grabs all rows whose 2nd level index = 3. However, `.xs()` has its limitations, as we can't grab all rows whose 2nd level index = 2 *or* 3. That is, we can't pass more than one value into the `.xs()` key argument for each level searched: `df.xs(('G2',3),level=[0,1])` is valid, `df.xs((2,3),level=1)` is not. We'll see how to do this in the next section with slicers.
Also, we lose the "Num" column, so we don't see that these rows have a Num value of 3. It might help in this case to assign the cross-section to a variable like `df_Num3`. Finally, unlike `df.loc[]` and `df.iloc[]`, `df.xs()` cannot be used to set values.

Using slicers

See https://docs.python.org/3/library/functions.html#slice
and https://pandas.pydata.org/docs/user_guide/advanced.html#using-slicers

Slicers provide sophisticated indexing in both axes (row and column).

```
mi = pd.MultiIndex.from_product([['G1','G2','G3'],[1,2,3],['a','b']],
                                names=['L1','L2','L3'])
df = DataFrame(np.arange(1,55).reshape(3,18).T,index=mi)
df
```

			0	1	2
L1	**L2**	**L3**			
G1	**1**	**a**	1	19	37
		b	2	20	38
	2	**a**	3	21	39
		b	4	22	40
	3	**a**	5	23	41
		b	6	24	42
G2	**1**	**a**	7	25	43
		b	8	26	44
	2	**a**	9	27	45
		b	10	28	46
	3	**a**	11	29	47
		b	12	30	48
G3	**1**	**a**	13	31	49
		b	14	32	50
	2	**a**	15	33	51
		b	16	34	52
	3	**a**	17	35	53
		b	18	36	54

Grab all of L1, 1 *and* 3 from L2, and all of L3:

```
df.loc[(slice(None),[1,3]),:]
```

			0	1	2
L1	**L2**	**L3**			
G1	**1**	**a**	1	19	37
		b	2	20	38
	3	**a**	5	23	41
		b	6	24	42
G2	**1**	**a**	7	25	43
		b	8	26	44
	3	**a**	11	29	47
		b	12	30	48
G3	**1**	**a**	13	31	49
		b	14	32	50
	3	**a**	17	35	53
		b	18	36	54

because `df.loc[(:,[1,3]),:]` won't work

Grab a slice from G2 to G3 of L1, all of L2, and **a** from L3:

```
df.loc[(slice('G2','G3'),slice(None),['a']),:]
```

			0	1	2
L1	**L2**	**L3**			
G2	**1**	**a**	7	25	43
	2	**a**	9	27	45
	3	**a**	11	29	47
G3	**1**	**a**	13	31	49
	2	**a**	15	33	51
	3	**a**	17	35	53

Swapping index levels

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.swaplevel.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.reorder_levels.html

We can change the order of a hierarchical index by either swapping two levels or setting a new order level.
The following pairs of methods are equivalent, and do not change the DataFrame in place.
To change the order of column levels pass `axis=1`.

`df.swaplevel()`
`df.reorder_levels([0,2,1])`

Swaps the innermost levels,
essentially the same as `df.swaplevel(-2,-1)`

			0	1	2
L1	L3	L2			
G1	a	1	1	19	37
	b	1	2	20	38

`df.swaplevel(0)`
`df.reorder_levels([2,1,0])`

Swaps the first with the last level

			0	1	2
L3	L2	L1			
a	1	G1	1	19	37
b	1		2	20	38

`df.swaplevel(0,1)`
`df.reorder_levels([1,0,2])`

Swaps the outermost levels

			0	1	2
L2	L1	L3			
1	G1	a	1	19	37
		b	2	20	38

Sorting by index level

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sort_index.html

In the example above, the original order of rows was maintained. We can also sort rows by one or more index levels.

`df.sort_index(level='L3')`

			0	1	2
L1	L2	L3			
G1	1	a	1	19	37
	2	a	3	21	39
	3	a	5	23	41
G2	1	a	7	25	43
	2	a	9	27	45
	3	a	11	29	47

`df.sort_index(level=['L3','L2'])`

			0	1	2
L1	L2	L3			
G1	1	a	1	19	37
G2	1	a	7	25	43
G3	1	a	13	31	49
G1	2	a	3	21	39
G2	2	a	9	27	45
G3	2	a	15	33	51

Values sort ascending unless we pass in `ascending=False`
Null values sort last unless we pass in `na_position='first'`
To sort by column levels pass in `axis=1`

COLUMN HIERARCHY

The same technique can be applied to columns.

```
df = DataFrame(np.arange(12).reshape(3,4),
     columns=pd.MultiIndex.from_product([['A', 'B'],['X', 'Y']]))
df
```

	A		B	
	X	Y	X	Y
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

Note that the number of hierarchical columns (2x2) has to match the number of columns in the data (4).

Adding column level names

Column levels can be named just as index levels are.

```
df.columns.names = ['1st','2nd']
df
```

1st	A		B	
2nd	X	Y	X	Y
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

Swapping column levels

`df.swaplevel('2nd','1st', axis=1)` equivalent to `df.swaplevel(axis=1)`

or `df.reorder_levels([1,0], axis=1)`

2nd	X	Y	X	Y
1st	A	A	B	B
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

Selecting columns - avoid chained indexing

From https://pandas.pydata.org/docs/user_guide/indexing.html#returning-a-view-versus-a-copy

Using the multilevel df DataFrame above, consider these two methods for selecting the fourth column values:

```
df['B']['Y']          called "chained indexing" this takes two steps, and can lead to a performance hit
0     3
1     7
2    11
Name: Y, dtype: int32

df.loc[:,('B','Y')]   computationally more efficient, this method is preferred
0     3
1     7
2    11
Name: (B, Y), dtype: int32
```

Also, when setting values,

`df.loc[:,('B','Y')] = [1,2,3]` works, but `df['B']['Y'] = [1,2,3]` doesn't.

Operations on column levels

The old methods have changed.

To perform operations on specific column levels refer to the section on groupby in this document.

```
df.sum(level='1st',axis=1)
```

1st	A	B
0	1	5
1	9	13
2	17	21

FutureWarning: Using the level keyword in DataFrame and Series aggregations is deprecated and will be removed in a future version. Use groupby instead. df.sum(level=1) should use df.groupby(level=1).sum().

Swapping rows and columns

We can use .transpose() to set the multilevel index as the index and the range index as the columns:

```
df.T
```

		0	1	2
1st	**2nd**			
A	**X**	0	4	8
	Y	1	5	9
B	**X**	2	6	10
	Y	3	7	11

MISSING DATA

Finding, Dropping missing data in a Series

See https://pandas.pydata.org/docs/reference/api/pandas.Series.isna.html
and https://pandas.pydata.org/docs/reference/api/pandas.Series.notna.html
and https://pandas.pydata.org/docs/reference/api/pandas.Series.dropna.html

Use `isna` to identify, and `dropna` to remove null values

```
ser = Series(['one','two', np.nan, 'four'])
ser              ser.isna()       ser.notna()      ser.dropna()
0     one        0 False          0  True          0     one
1     two        1 False          1  True          1     two
2     NaN        2  True          2 False          3    four
3    four        3 False          3  True          dtype: object
dtype: object    dtype: bool      dtype: bool
```

Characters such as empty strings `''` or `numpy.inf` are not considered NA values.

`ser.isnull()` is an alias for `ser.isna()`, and
`ser.notnull()` is an alias for `ser.notna()`

Finding, Dropping missing data in a DataFrame (*Be Careful!*)

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.isna.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.notna.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.dropna.html

```
df = pd.DataFrame({'A':[1,2,3,np.nan],'B':[5,np.nan,7,np.nan],'C':[1,2,3,4]})
df
```

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	3.0	7.0	3
3	NaN	NaN	4

when an integer column contains a null value,
its data type reverts to a float.

`df.isna()`/`df.isnull()` and `df.notna()`/`df.notnull()` work as above.

```
df.dropna()                 will drop entire rows that contain at least one null value
df.dropna(how='all')        will drop only rows missing all data
df.dropna(axis=1)           will drop entire columns that contain at least one null value
df.dropna(thresh=2)         will drop rows that don't have at least 2 valid data points
df.dropna(subset=['A'])     will drop only rows missing data in column 'A'
```

With `inplace=False`, none of these methods change `df` in place.

Filling in missing data points

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html

Do this with caution! Generally, an empty cell is *not* the same as a zero-value cell, but cases can be made for interpolating missing values.

`df`

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	3.0	7.0	3
3	NaN	NaN	4

`df.fillna(1)` scalar: fills any missing data point with the same value

	A	B	C
0	1.0	5.0	1
1	2.0	1.0	2
2	3.0	7.0	3
3	1.0	1.0	4

`df.fillna({'A':7,'B':8})` dict: fills individual columns with the same value

	A	B	C
0	1.0	5.0	1
1	2.0	8.0	2
2	3.0	7.0	3
3	7.0	8.0	4

`df.fillna({'A':7,'B':8}, downcast='infer')` reverts floats back to ints

	A	B	C
0	1	5	1
1	2	8	2
2	3	7	3
3	7	8	4

`df.interpolate()` fills in equally spaced values by default. See docs for more options.

	A	B	C
0	1.0	5.0	1
1	2.0	6.0	2
2	3.0	7.0	3
3	3.0	7.0	4

For more info: https://pandas.pydata.org/docs/user_guide/missing_data.html

APPLYING FUNCTIONS TO DATA

As seen in the section on Summary Statistics, DataFrames offer a variety of built-in aggregate methods like .mean() In this section we'll apply methods to selected columns, apply multiple methods to the same column, and run user-defined functions on column data.

```
data = {'Name':['Amir','Beck','Cleo','Drew'],
        'Col1':[1,4,1,4],
        'Col2':[3,2,3,2],
        'Col3':[6,7,5,8]}
df = DataFrame(data)
df
```

	Name	Col1	Col2	Col3
0	Amir	1	3	6
1	Beck	4	2	7
2	Cleo	1	3	5
3	Drew	4	2	8

columns will appear in dictionary insertion order

Running aggregate methods on selected columns

```
df['Col3'].mean()
6.5

df[['Col3','Col2']].mean()
Col3    6.5
Col2    2.5
dtype: float64
```

requires two sets of brackets

Running user-defined functions on selected columns

See https://pandas.pydata.org/docs/reference/api/pandas.Series.apply.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html

Apply functions involving a single column

```
def times2(x):
    return x*2

df['Col3'].apply(times2)
0    12
1    14
2    10
3    16
Name: Col3, dtype: int64
def times2(x):
    return x*2
```

```
df[['Col3']].apply(times2)
```

	Col3
0	12
1	14
2	10
3	16

Use double-brackets to return a DataFrame

We can apply built-in Python functions:

```
df['Name'].apply(len)
0    4
1    4
2    4
3    4
Name: Name, dtype: int64
```

We can apply anonymous functions:

```
df['Col2'].apply(lambda x: x**2)
0    9
1    4
2    9
3    4
Name: Col2, dtype: int64
```

The previous methods applied functions element-wise. We can also define and apply aggregate functions.
For example, while DataFrames have .max() and .min() methods, they lack a statistical range method. Let's define one:

```
def stat_range(arr):                Don't use "range" as a function name!
    return arr.max()-arr.min()      It will overwrite Python's built-in range() generator function.

df[['Col3']].apply(stat_range)      requires double brackets
Col3    3
dtype: int64
```

Applying functions involving multiple columns
There are several ways to do this.

```
def some_math(x,y):
    return (10*x)+y
```

Using lambda (slow):

```
df[['Col2','Col3']].apply(lambda df: some_math(df['Col2'],df['Col3']),axis=1)
0    36
1    27
2    35
3    28
dtype: int64
```

Using numpy vectorize (computationally much faster):

```
np.vectorize(some_math)(df['Col2'],df['Col3'])
array([36, 27, 35, 28], dtype=int64)
```

For more info: https://numpy.org/doc/stable/reference/generated/numpy.vectorize.html

Running multiple functions on selected columns
See https://pandas.pydata.org/docs/reference/api/pandas.Series.agg.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.agg.html

The `.agg()` method lets us pass a list of aggregate functions, or a dictionary that maps functions to columns to a DataFrame object. Note: `.agg()` is an alias for `.aggregate()`. Use the alias.

```
df['Col1'].agg(['min','max',stat_range])      don't put quotes around user-defined functions
min           1                               rows appear in insertion order
max           4
stat_range    3
Name: Col1, dtype: int64
```

```
df[['Col2','Col3']].agg(['min','max',stat_range])
```

	Col2	Col3
min	2	5
max	3	8
stat_range	1	3

```
df.agg({'Col2':['sum'],'Col3':['mean','std']})
```

	Col2	Col3
sum	10.0	NaN
mean	NaN	6.500000
std	NaN	1.290994

DATAFRAME ARITHMETIC

See:

pandas.DataFrame.add()
pandas.DataFrame.sub()
pandas.DataFrame.mul()
pandas.DataFrame.pow()
pandas.DataFrame.div()
pandas.DataFrame.floordiv()
pandas.DataFrame.mod()
pandas.DataFrame.abs()

and their reverse counterparts:

pandas.DataFrame.radd()
pandas.DataFrame.rsub()
pandas.DataFrame.rmul()
pandas.DataFrame.rpow()
pandas.DataFrame.rdiv()
pandas.DataFrame.rfloordiv()
pandas.DataFrame.rmod()

We can use common arithmetic operators (`+`, `-`, `*`, etc.) to perform math with scalers, or to add, multiply, etc. two dataframes together. Method calls add the ability to substitute a fill_value for missing data, if appropriate. (Generally, an empty cell is *not* the same as a zero-value cell, but cases can be made for interpolating missing values).

```
df1 = DataFrame([[7,10,'a'],[8,11,'b'],[-9,np.nan,'c']],
                columns=['col1','col2','col3'])

df2 = DataFrame([[4,2,'d'],[8,4,'e'],[10,6,'f']],
                columns=['col1','col2','col3'])
```

df1

	col1	col2	col3
0	7	10.0	a
1	8	11.0	b
2	-9	NaN	c

df2

	col1	col2	col3
0	4	2	d
1	8	4	e
2	10	6	f

In the following examples, equivalent statements are stacked.

Addition

```
df1 + df2
df1.add(df2)
```

	col1	col2	col3
0	11	12.0	ad
1	16	15.0	be
2	1	NaN	cf

value + NaN = NaN

```
df1.add(df2, fill_value=0)
```

	col1	col2	col3
0	11	12.0	ad
1	16	15.0	be
2	1	6.0	cf

handles missing data in either input

```
df1[['col1','col2']] + [2,3]
df1[['col1','col2']].add([2,3])
```

	col1	col2
0	9	13.0
1	10	14.0
2	-7	NaN

All of pandas arithmetic methods have corresponding "reverse" methods that swap the order of the inputs:

```
df1.radd(df2)
```

	col1	col2	col3
0	11	12.0	da
1	16	15.0	eb
2	1	NaN	fc

Except for some scalar operations, addition is about the only thing we can do with strings.
For the remaining exercises we'll remove the string columns:

```
_1, _2 = df1.pop('col3'), df2.pop('col3')
```

Subtraction

```
df1 - df2
df1.sub(df2)
```

	col1	col2
0	3	8.0
1	0	7.0
2	-19	NaN

```
df1 - [2,3]
df1.sub([2,3])
```

	col1	col2
0	5	7.0
1	6	8.0
2	-11	NaN

Multiplication

```
df1 * df2
df1.mul(df2)
```

	col1	col2
0	28	20.0
1	64	44.0
2	-90	NaN

```
df1.mul(df2, fill_values=1)
```

	col1	col2
0	28	20.0
1	64	44.0
2	-90	6.0

Exponentiation

```
df1 ** 2
df1.pow(2)
```

	col1	col2
0	49	100.0
1	64	121.0
2	81	NaN

```
2 ** df2
df2.rpow(2)
```

this can't be done with negative values

	col1	col2
0	16	4
1	256	16
2	1024	64

Division

```
df1 / df2
df1.div(df2)
```

	col1	col2
0	1.75	5.00
1	1.00	2.75
2	-0.90	NaN

```
df1.div(df2, fill_values=1)
```

	col1	col2
0	1.75	5.000000
1	1.00	2.750000
2	-0.90	0.166667

Note that `fill_values` provides the same value to all inputs. To differentiate between numerator/denominator:

```
df1.fillna(0).div(df2.fillna(1))
```

	col1	col2
0	1.75	5.00
1	1.00	2.75
2	-0.90	0.00

Floor Division and Modulo

Where `.div()` performs "true division" (aka *float division*), floor division returns the integer quotient of division without the remainder. Modulo returns the remainder after floor division.

```
df1 // 3
df1.floordiv(3)
```

	col1	col2
0	2	3.0
1	2	3.0
2	-3	NaN

```
df1 % 3
df1.mod(3)
```

	col1	col2
0	2	3.0
1	2	3.0
2	-3	NaN

```
30 % df1
df1.rmod(30)
```

	col1	col2
0	2	0.0
1	6	8.0
2	-6	NaN

Absolute Value

```
df1.abs()
```

does not take any arguments

	col1	col2
0	7	10.0
1	8	11.0
2	9	NaN

GROUPBY ON DATAFRAMES

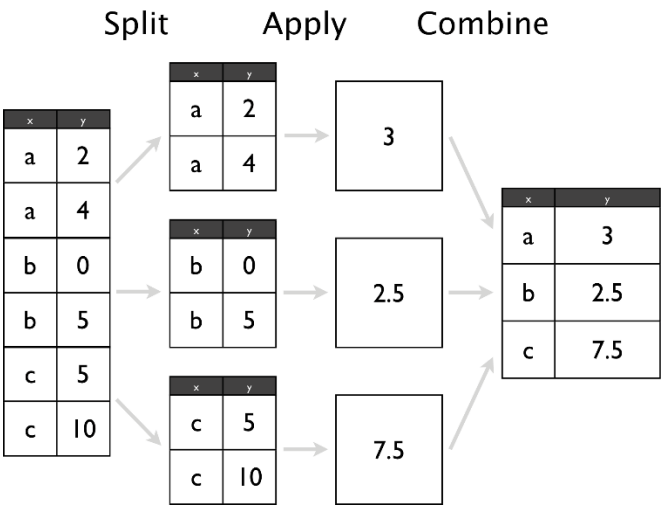
See https://pandas.pydata.org/docs/user_guide/groupby.html

By "group by" we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria.
- **Applying** a function to each group independently.
- **Combining** the results into a data structure.

Split, Apply, Combine - a visual explanation

Image source = https://github.com/ramnathv/rblocks/issues/8



Split here is accomplished by the groupby command. If the function you're applying requires that members of the group be sorted, sort the DataFrame first.

Apply can be a predefined function to be performed on each group in turn.

Combine is whatever gets returned once the apply finishes.

```
data = {'Day':[1,2,3,4],
        'Name':['Evan','Fran','Evan','Fran'],
        'Col3':[3,2,2,3],
        'Col4':[6,7,np.nan,8]}
df = DataFrame(data)
df
```

	Day	Name	Col3	Col4
0	1	Evan	3	6.0
1	2	Fran	2	7.0
2	3	Evan	2	NaN
3	4	Fran	3	8.0

columns will appear in dictionary insertion order

Create a GroupBy object

```
df_by_name = df.groupby('Name')
df_by_name
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000023E51864910>
```

divides the entire DataFrame into groups around entries in 'Name'

Note: `df.groupby(1)` won't work

A GroupBy object is just stored data, not a DataFrame. No splitting occurs until it's needed.
Creating the GroupBy object only verifies that we've passed a valid mapping.

In this case we've mapped the entire DataFrame into the GroupBy object – we'll see that this can make some methods more cumbersome. It helps to understand the data before constructing GroupBy objects.

GroupBy methods

```
df_by_name.sum()
```

	Day	Col3	Col4
Name			
Evan	4	5	6.0
Fran	6	5	15.0

equivalent to `df.groupby('Name').sum()`

most aggregate functions return a DataFrame
where the groupby column becomes a named index

Note that summing a "Day" column may not make sense. We can drop it when constructing the groupby object:

```
df_by_name = df.drop('Day',axis=1).groupby('Name')
```

```
df_by_name.sum()
```

	Col3	Col4
Name		
Evan	5	6.0
Fran	5	15.0

sums numerical columns, ignores null values
throws an error if there are categorical columns
(see next section on handling mixed data types)

```
df_by_name.mean()
```

	Col3	Col4
Name		
Evan	2.5	6.0
Fran	2.5	7.5

.mean() ignores null values (the mean of x and null is x)

```
df_by_name.size()
```

```
Name
Evan    2
Fran    2
dtype: int64
```

returns a Series with the number rows in each group

```
df_by_name.count()
```

	Col3	Col4
Name		
Evan	2	1
Fran	2	2

returns a count of not-null members that match up to each Name value

```
df_by_name.describe()
```

	Col3								Col4		
	count	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**	**count**	**mean**	**std**
Name											
Evan	2.0	2.5	0.707107	2.0	2.25	2.5	2.75	3.0	1.0	6.0	NaN
Fran	2.0	2.5	0.707107	2.0	2.25	2.5	2.75	3.0	2.0	7.5	0.707107

For numeric data, the result's index will include count, mean, std, min, max as well as lower, 50 and upper percentiles.
For object data (e.g. strings or timestamps), the result's index will include count, unique, top, and freq.
For mixed data types the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If include='all' is provided as an option, the result will include a union of attributes of each type.
For more info: https://pandas.pydata.org/docs/reference/api/pandas.core.groupby.DataFrameGroupBy.describe.html

Dealing with mixed data types

If we group by Col3 (dropping the Day column), that leaves Name and Col4 as our value columns where Name is categorical and Col4 is numeric. Consider:

```
df_by_3 = df.drop('Day',axis=1).groupby('Col3')
df_by_3.max()
```

	Name	Col4
Col3		
2	Fran	7.0
3	Fran	8.0

This is ok as `.max()` returns the last string alphabetically in the group. However:

```
df_by_3.sum()
FutureWarning: The default value of numeric_only in DataFrameGroupBy.sum is
deprecated.
```

We have two choices: either pass an argument into the method call, or change the way we construct the groupby.

Change the method call:

```
df_by_3.sum(numeric_only=True)
```

	Col4
Col3	
2	7.0
3	14.0

Change the groupby:

```
df_by_3 = df.select_dtypes(include=np.number).drop('Day',axis=1).groupby('Col3')
df_by_3.sum()
```

	Col4
Col3	
2	7.0
3	14.0

This forces us to group by a numerical column. Similarly we can pass `exclude=np.number` to retain only categorical columns – but this means we can only group by a categorical column.

In most cases it makes sense to use discrete column selections:

```
df_by_3 = df[['Name','Col3']].groupby('Col3')
```

or

```
df_by_3 = df.groupby('Col3')[['Name']]   use double brackets to return a DataFrameGroupBy object
```

```
df_by_3.nunique()
```

	Name
Col3	
2	2
3	2

For more info: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.select_dtypes.html

GroupBy sorting

`df`

	Day	Name	Col3	Col4
0	1	Evan	3	6.0
1	2	Fran	2	7.0
2	3	Evan	2	NaN
3	4	Fran	3	8.0

Group keys are sorted by default:

```
df.groupby('Col3').max()
```

	Day	Name	Col4
Col3			
2	3	Fran	7.0
3	4	Fran	8.0

To prevent this, so that groups appear in DataFrame order:

```
df.groupby('Col3', sort=False).max()
```

	Day	Name	Col4
Col3			
3	4	Fran	8.0
2	3	Fran	7.0

Running aggregate methods on selected columns

```
df.groupby('Col3')[['Day','Col4']].max()
```
or
```
df[['Day','Col4']].groupby(df['Col3']).max()
```

	Day	Col4
Col3		
2	3	7.0
3	4	8.0

Returns a DataFrame with maximums for Day and Col4, omitting the Name column
When specifying columns from df in the second example, we also have to call *df*['Col1'] because `df[['Day','Col4']].groupby('Col3')` raises a KeyError.

Running multiple functions on selected columns

See https://pandas.pydata.org/docs/reference/api/pandas.core.groupby.DataFrameGroupBy.aggregate.html

The `.agg()` method lets us pass a list of aggregate functions, or a dictionary that maps functions to columns to a GroupBy object. Note: `.agg()` is an alias for `.aggregate()`. Use the alias.

```
df.groupby('Name').agg({'Col3':['max','sum'],'Col4':['mean']})
```

	Col3		Col4
	max	**sum**	**mean**
Name			
Evan	3	5	6.0
Fran	3	5	7.5

Group by multiple column keys

```
df = DataFrame([list('ABABAB'),list('CCCDDD'),np.arange(1,7)],
                index=['1st','2nd','val']).T
```

df

	1st	2nd	val
0	A	C	1
1	B	C	2
2	A	C	3
3	B	D	4
4	A	D	5
5	B	D	6

```
df.groupby(['1st','2nd']).sum()
```

		val
1st	**2nd**	
A	**C**	4
	D	5
B	**C**	2
	D	10

returns a DataFrame with a hierarchical index
only useful if the same 2nd value appears multiple times for a given 1st

We can select columns to return:

```
df.groupby(['1st','2nd'])[['val']].sum()
```

or

```
df[['val']].groupby([df['1st'],df['2nd']]).sum()
```

returns same as above in this case

Note that `df[['val']].groupby(df[['1st','2nd']])` is not valid code.

Assign keys to a column and group by them instead

```
province = np.array(['AB','BC','AB','BC','AB','BC'])
timezone = np.array(['UTC','UTC','UTC','DST','DST','DST'])
df[['val']].groupby([province,timezone]).sum()
```

		val
AB	**DST**	5
	UTC	4
BC	**DST**	10
	UTC	2

Note that the output sorts by province then timezone alphabetically *after* assigning values to groups.
In practice we'd have to know that the passed in arrays match up row-by-row with column values.
As before, calling `df[['val']]` with double-brackets returns a DataFrame, while single-brackets would return a multilevel Series.

Iterate over groups

```
for name,group in df.groupby('1st'):
    print (f'This is the {name} group')
    print (group, '\n')
```

```
This is the A group
  1st 2nd val
0   A   C   1
2   A   C   3
4   A   D   5

This is the B group
  1st 2nd val
1   B   C   2
3   B   D   4
5   B   D   6
```

Iteration across multiple keys

```
for (k1,k2),group in df.groupby(['1st','2nd']):
    print (f'Key1 = {k1} Key2 = {k2}')
    print (group, '\n')
```

```
Key1 = A Key2 = C
  1st 2nd val
0   A   C   1
2   A   C   3

Key1 = A Key2 = D
  1st 2nd val
4   A   D   5

Key1 = B Key2 = C
  1st 2nd val
1   B   C   2

Key1 = B Key2 = D
  1st 2nd val
3   B   D   4
5   B   D   6
```

the return is sorted by k1 then k2

Create a dictionary from grouped data pieces

```
group_dict = dict(list(df.groupby('1st')))
group_dict
```

```
{'A':   1st 2nd val
 0   A   C   1
 2   A   C   3
 4   A   D   5,
 'B':   1st 2nd val
 1   B   C   2
 3   B   D   4
 5   B   D   6}
```

Here each unique member of 1st becomes a key, and its group becomes a value!

`group_dict['A']` returns a DataFrame:

	1st	2nd	val
0	A	C	1
2	A	C	3
4	A	D	5

Apply GroupBy to columns using a dictionary

```
rng = np.random.default_rng(3)
trips = DataFrame(rng.integers(0,20,size=16).reshape((4, 4)),
                  columns=['DC','LA','NY','SF'],
                  index=['Amir','Beck','Cleo','Drew'])
trips
```

	DC	LA	NY	SF
Amir	16	1	3	4
Beck	3	16	17	11
Cleo	0	1	6	8
Drew	12	9	5	3

Create a dictionary that maps cities to "region" values:

```
region_map = {'NY':'east','DC':'east','LA':'west','SF':'west','LV':'west'}
```

Order doesn't matter, and we can have cities that don't appear in the DataFrame.

Group the DataFrame using the dictionary:

```
regions = trips.groupby(region_map, axis=1)
regions.sum()
```

	east	west
Amir	19	5
Beck	20	27
Cleo	6	9
Drew	17	12

Depending on the dictionary, we may have to manipulate keys and values:

```
region_map = {'east':['NY','DC'],'west':['LA','SF','LV']}
```

we can't work directly from this

Using a for loop:

```
region_map_exploded = {}
for key in region_map.keys():
    for val in region_map[key]:
        region_map_exploded[val]=key
```

Using a Series:

```
ser = Series(region_map).explode()
region_map_exploded = Series(ser.index,ser.values).to_dict()

region_map_exploded
{'NY': 'east', 'DC': 'east', 'LA': 'west', 'SF': 'west', 'LV': 'west'}

regions = trips.groupby(region_map_exploded, axis=1)
regions.sum()
```

Returns the same as above

PIVOTING DATAFRAMES

Many of the following examples derive from https://pandas.pydata.org/docs/user_guide/reshaping.html

This section covers the following methods:

DataFrame.pivot	reshapes data based on column values, does not support data aggregation
DataFrame.pivot_table	creates a spreadsheet-style pivot table as a DataFrame, aggregates values (default=mean)
pandas.crosstab	returns a frequency table by default, although other aggfuncs are supported

DataFrame.pivot

See: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pivot.html

Reshapes data. Does not support aggregation, but it does handle non-numeric values.

```
df = DataFrame({'foo': ['one','one','one','two','two','two'],
                'bar': ['A','B','C','A','B','C'],
                'baz': [1, 2, 3, 4, 5, 6],
                'zoo': ['x','y','z','q','w','t']})
```

df

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

```
df.pivot(index='foo', columns='bar', values='baz')
```

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

The resulting DataFrame sets "foo" as a named index, unique values from "bar" become column headings, and intersecting values from "baz" fill the frame. For this to work, "foo"/"bar" combinations must be unique. If not, pandas raises a *ValueError: Index contains duplicate entries, cannot reshape*.

```
df.pivot(index='foo', columns='bar', values=['baz','zoo'])
```

	baz			zoo		
bar	**A**	**B**	**C**	**A**	**B**	**C**
foo						
one	1	2	3	x	y	z
two	4	5	6	q	w	t

Equivalent in this case to `df.pivot(index='foo', columns='bar')`

DataFrame.pivot_table
See: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pivot_table.html

Creates a spreadsheet-style pivot table as a DataFrame. Values are aggregated; the default aggfunc is np.mean().

	A	B	C	D
0	foo	one	x	1
1	foo	one	x	3
2	foo	two	x	2
3	bar	two	y	5
4	bar	one	x	0
5	bar	one	y	1

```
d = {'A':['foo','foo','foo','bar','bar','bar'],
     'B':['one','one','two','two','one','one'],
     'C':['x','x','x','y','x','y'],
     'D':[1,3,2,5,0,1]}
df = DataFrame(d)
df
```

```
df.pivot_table(values='D',index='A',columns='B')
```

D column values are aggregated using np.mean() by default

B	one	two
A		
bar	0.5	5.0
foo	2.0	2.0

```
df.pivot_table('D','A','B',aggfunc=np.sum)
```

B	one	two
A		
bar	1	5
foo	4	2

```
df.pivot_table(values='D',index=['A', 'B'],columns=['C'])
```

	C	x	y
A	**B**		
bar	**one**	4.0	1.0
	two	NaN	5.0
foo	**one**	**2.0**	NaN
	two	2.0	NaN

In this example, `.pivot_table` maps the values in column D into columns formed from values in column C against a hierarchical index of A and B. Note that most column D values belong to unique combinations of A, B and C, with the exception of [**foo, one, x**]. Here, the D values of 1 and 3 were aggregated.

For more info: https://en.wikipedia.org/wiki/Pivot_table

Cross Tabulation
See https://pandas.pydata.org/docs/reference/api/pandas.crosstab.html

The pandas.crosstab general function returns a frequency table by default, although other aggfuncs are supported.

```
pd.crosstab(df['A'],df['B'],margins=True))
```

margins=True adds the "All" info

B	one	two	All
A			
bar	2	1	3
foo	2	1	3
All	4	2	6

STACKING

See: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.stack.html

Reshapes a DataFrame by pivoting a level of column labels into a multi-level index. If the columns have a single level, the output is a Series; if the columns have multiple levels, the output is a DataFrame. By default the innermost level is unstacked, but that can be controlled using an optional level argument.

```
df = pd.DataFrame(np.arange(1,13).reshape(3,4),
     columns=pd.MultiIndex.from_product([['a', 'b'],['x', 'y']]))
```

df

	a		b	
	x	y	x	y
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

`df.stack()`

		a	b
0	x	1	3
	y	2	4
1	x	5	7
	y	6	8
2	x	9	11
	y	10	12

Equivalent to `df.stack(-1)`

`df.stack(level=0)`

		x	y
0	a	1	2
	b	3	4
1	a	5	6
	b	7	8
2	a	9	10
	b	11	12

This is sometimes referred to as reshaping data from a 'wide' format where repeated measurements appear in separate columns of the same record to a 'long' format which places repeated measurements in separate records.

Stack filters out (removes) resulting rows where all values are missing. To avoid this use `.stack(dropna=False)`

UNSTACKING

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.unstack.html
and https://pandas.pydata.org/docs/reference/api/pandas.Series.unstack.html

Unstacking lets us pivot a level of hierarchical index labels. By default the innermost level is unstacked, but that can be controlled using an optional level argument. If the index is not a MultiIndex, the output will be a Series.

Unstacking a MultiIndex DataFrame

```
df = DataFrame(np.arange(1,13).reshape(3,4).T,
     index=pd.MultiIndex.from_product([['A', 'B'],['X', 'Y']]))
df
```

		0	1	2
A	**X**	1	5	9
	Y	2	6	10
B	**X**	3	7	11
	Y	4	8	12

```
df.unstack()
```

	0		1		2	
	X	**Y**	**X**	**Y**	**X**	**Y**
A	1	2	5	6	9	10
B	3	4	7	8	11	12

Equivalent to `df.unstack(-1)`

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

	0		1		2	
	A	**B**	**A**	**B**	**A**	**B**
X	1	3	5	7	9	11
Y	2	4	6	8	10	12

Unstacking a MultiIndex Series returns a DataFrame

```
ser = Series(np.arange(6),index=[[1,1,1,2,2,2],['a','b','c','a','b','c']])
ser
```

```
1  a    0
   b    1
   c    2
2  a    3
   b    4
   c    5
dtype: int32
```

```
ser.unstack()
```

	a	b	c
1	0	1	2
2	3	4	5

```
ser.unstack(level=0)
```

	1	2
a	0	3
b	1	4
c	2	5

RESHAPING BY MELT

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.melt.html

Melt offers more control over how we massage a DataFrame by setting selected columns as *identifier* variables and pivoting others into a pair of *measured* variable columns labelled "variable" and "value".

```
cars = DataFrame([['Honda','Accord','EX',30705,32,111.4],
                  ['Toyota','Camry','SE',28855,32,111.2]],
                  columns=['make','model','trim','msrp','mpg-gas',mpg-
hybrid,'wheelbase'])
cars
```

	make	model	trim	msrp	mpg	wheelbase
0	Honda	Accord	EX	30705	32	111.4
1	Toyota	Camry	SE	28855	32	111.2

```
cars.melt(id_vars=['make','model','trim'],value_vars=['mpg','wheelbase'])
```

	make	model	trim	variable	value
0	Honda	Accord	EX	mpg	32.0
1	Toyota	Camry	SE	mpg	32.0
2	Honda	Accord	EX	wheelbase	111.4
3	Toyota	Camry	SE	wheelbase	111.2

If we don't pass a `value_vars` parameter then all remaining columns will be added to variable.
Because the wheelbase column contained float values, the entire value column becomes a float.
Original index values are ignored unless we pass `ignore_index=False`, which will repeat values.
We can assign our own names to the variable and value columns by passing `var_name` and `value_name`.

For more info: https://pandas.pydata.org/docs/user_guide/reshaping.html#reshaping-melt

Similar functionality with pandas.wide_to_long()

See https://pandas.pydata.org/docs/reference/api/pandas.wide_to_long.html

This pandas general function handles frames with "stub data" – groups of columns whose names share the same prefix.

```
cars = DataFrame([['Honda','Accord','EX',32,48,111.4],
                  ['Toyota','Camry','SE',32,52,111.2]],
                  columns=['make','model','trim','mpg-gas','mpg-hybrid',
                           'wheelbase'])
cars
```

	make	model	trim	mpg-gas	mpg-hybrid	wheelbase
0	Honda	Accord	EX	32	48	111.4
1	Toyota	Camry	SE	32	52	111.2

```
pd.wide_to_long(cars, stubnames='mpg', sep='-', suffix='\D+',
                i=['make','model','trim'], j='power')
```

				wheelbase	mpg
make	**model**	**trim**	**power**		
Honda	**Accord**	**EX**	**gas**	111.4	32
			hybrid	111.4	48
Toyota	**Camry**	**SE**	**gas**	111.2	32
			hybrid	111.2	52

Passing `suffix='\D+'` is required for strings, passing `suffix='\d+'` is optional for numerical suffixes.
Note that, in the absence of missing data, the mpg column retains its integer dtype.
`['make', 'model', 'trim', 'power']` form a new MultiIndex on the returned DataFrame.

COMBINING DATAFRAMES
See https://pandas.pydata.org/docs/user_guide/merging.html

APPEND (deprecated)
The DataFrame.append() method has been deprecated; use pandas.concat() instead.
See https://pandas.pydata.org/docs/whatsnew/v1.4.0.html#whatsnew-140-deprecations-frame-series-append

CONCATENATE
See https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html
and https://pandas.pydata.org/docs/reference/api/pandas.concat.html

In numpy, to concatenate two or more arrays
If `arr1` is a 3x4 array:

`np.concatenate([arr1,arr1],axis=0)` creates a vertical, 6x4 array (default)
`np.concatenate([arr1,arr1],axis=1)` creates a horizontal, 3x8 array

In pandas, to concatenate two or more Series

`pd.concat([ser1,ser2])` returns one long vertical Series
`pd.concat([ser1,ser2], axis=1)` returns a DataFrame
`ser1`'s values fall in column 0, `ser2` in column 1

NOTE: if the two Series being concatenated share a common index value, then

- the index value will be repeated in a vertical concatenation (axis=0)
- the index value will appear once, and have values in both columns (axis=1)

Concatenate two or more DataFrames – columns match

```
df1 = DataFrame({'A': ['A0', 'A1'],
                 'B': ['B0', 'B1']})
```

df1

	A	B
0	A0	B0
1	A1	B1

```
df2 = DataFrame({'A': ['A2', 'A3'],
                 'B': ['B2', 'B3']})
```

df2

	A	B
0	A2	B2
1	A3	B3

`pd.concat([df1,df2])`

	A	B
0	A0	B0
1	A1	B1
0	A2	B2
1	A3	B3

`pd.concat([df1,df2],ignore_index=True)`

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

It is worth noting that `concat()` makes a full copy of the data, and that constantly reusing this function can create a signifcant performance hit. If you need to use the operation over several datasets, use a list comprehension.

Concatenate two or more DataFrames – indexes match

```
df3 = DataFrame({'C': ['C0', 'C1'],
                 'D': ['D0', 'D1']},
                 index=[1,2])
```

df3

	C	D
1	C0	D0
2	C1	D1

```
df1 = DataFrame({'A': ['A0', 'A1'],
                 'B': ['B0', 'B1']})
```

df1

	A	B
0	A0	B0
1	A1	B1

```
pd.concat([df3,df1], axis=1)
```

	C	D	A	B
1	C0	D0	A1	B1
2	C1	D1	NaN	NaN
0	NaN	NaN	A0	B0

```
pd.concat([df3,df1], axis=1, sort=True)
```

	C	D	A	B
0	NaN	NaN	A0	B0
1	C0	D0	A1	B1
2	C1	D1	NaN	NaN

Concatenate two or more DataFrames – inner join

Returns a DataFrame with only those columns (or rows) shared in common.

```
df4 = DataFrame({'B': ['B2', 'B3'],
                 'C': ['C0', 'C1']},
                 index=[1,2])
```

df4

	B	C
1	B2	C0
2	B3	C1

```
df1 = DataFrame({'A': ['A0', 'A1'],
                 'B': ['B0', 'B1']})
```

df1

	A	B
0	A0	B0
1	A1	B1

```
pd.concat([df1,df4], join='inner')
```

	B
0	B0
1	B1
1	B2
2	B3

```
pd.concat([df1,df4],axis=1,join='inner')
```

	A	B	B	C
1	A1	B1	B2	C0

Add a hierarchical index using "keys"

```
df_new = pd.concat([df1, df2, df3], keys=['df1','df2','df3'], axis=1)
df_new
```

	df1		df2		df3	
	A	B	A	B	C	D
0	A0	B0	A2	B2	NaN	NaN
1	A1	B1	A3	B3	C0	D0
2	NaN	NaN	NaN	NaN	C1	D1

Keys can be assigned any name; we used `'df1'` for illustration.
To form a multilevel index, omit the `axis=1` argument.
MultiIndex selection methods will work as expected. `df_new['df2']` will retrieve only those columns belonging to `'df2'`, including rows with missing data.

Append a new row of data

```
df = DataFrame([[1,2],[3,4]],
               index=[1,2]
               columns=['a','b'])
```

```
new_row = Series({'a':5, 'b':6})
new_row
a    5
b    6
dtype: int64
```

df

	a	b
1	1	2
2	3	4

```
pd.concat([df,new_row.to_frame().T])
```

	a	b
1	1	2
2	3	4
0	5	6

```
pd.concat([df,new_row.to_frame().T], ignore_index=True)
```

resets the entire index

	a	b
0	1	2
1	3	4
2	5	6

Append a Series as a new column of data

Can be done as simply as:

```
colors = Series(['Blue','Red'],index=[2,4])
df['Color']=colors
```

`df` now has a Color column with Blue matched to index 2, Red to 4, and NaN after everything else.
Series index labels that don't already exist in DataFrame are ignored.

`pd.concat([df,colors], axis=1)` also works, but colors will lack a column name unless the Series is named.

MERGE

See https://pandas.pydata.org/docs/reference/api/pandas.merge.html

From https://pandas.pydata.org/docs/user_guide/merging.html#database-style-dataframe-or-named-series-joining-merging:
pandas has full-featured, **high performance** in-memory join operations idiomatically very similar to relational databases like SQL. These methods perform significantly better (in some cases well over an order of magnitude better) than other open source implementations (like base::merge.data.frame in R). The reason for this is careful algorithmic design and internal layout of the data in DataFrame.

```
left = DataFrame({'key': ['K1', 'K2', 'K3'],
                  'A': ['A1', 'A2', 'A3'],
                  'B': ['B1', 'B2', 'B3']})

right = DataFrame({'key': ['K1', 'K2', 'K3','K4'],
                   'C': ['C1', 'C2', 'C3','C4'],
                   'D': ['D1', 'D2', 'D3','D4']})
```

`left`

	key	A	B
0	K1	A1	B1
1	K2	A2	B2
2	K3	A3	B3

`right`

	key	C	D
0	K1	C1	D1
1	K2	C2	D2
2	K3	C3	D3
3	K4	C4	D4

`pd.merge(left,right)`

	key	A	B	C	D
0	K1	A1	B1	C1	D1
1	K2	A2	B2	C2	D2
2	K3	A3	B3	C3	D3

`merge` automatically chooses overlapping columns to merge on (here it's 'key')
`merge` performs an "inner join" by default, omitting any unmatched keys (here it's K4)

```
left2 = DataFrame({'code':['X','Y','X'],'A':[1,2,3]})
right2 = DataFrame({'code':['X','X','Y'],'B':[44,55,66]})
```

`left2`

	code	A
0	X	1
1	Y	2
2	X	3

`right2`

	code	B
0	X	44
1	X	55
2	Y	66

`pd.merge(left2,right2)`

	code	A	B
0	X	1	44
1	X	1	55
2	X	3	44
3	X	3	55
4	Y	2	66

Where shared key values appear more than once, `merge` provides every possible combination - in this case a Cartesian Product of `[1,3]` and `[44,55]`.

Merging on multiple keys

`left3`

	key1	key2	A	B
0	K1	K3	A1	B1
1	K2	K4	A2	B2
2	K1	K5	A3	B3

`right3`

	key1	key2	C	D
0	K1	K3	C1	D1
1	K2	K4	C2	D2
2	K2	K6	C3	D3

```
pd.merge(left3, right3, on=['key1','key2'], how='inner')
```
'inner' is the default

	key1	key2	A	B	C	D
0	K1	K3	A1	B1	C1	D1
1	K2	K4	A2	B2	C2	D2

```
pd.merge(left3, right3, on=['key1','key2'], how='outer')
```

	key1	key2	A	B	C	D
0	K1	K3	A1	B1	C1	D1
1	K2	K4	A2	B2	C2	D2
2	K1	K5	A3	B3	NaN	NaN
3	K2	K6	NaN	NaN	C3	D3

An outer join takes the union of the two DataFrames. Similarly, `how=left` uses only keys from the left frame (SQL: left outer join) and `how=right` uses only keys from the right frame (SQL: right outer join)

Merge key indicator (new in pandas 0.17.0):

```
pd.merge(left3, right3, on=['key1','key2'], how='outer', indicator=True)
```

	key1	key2	A	B	C	D	_merge
0	K1	K3	A1	B1	C1	D1	both
1	K2	K4	A2	B2	C2	D2	both
2	K1	K5	A3	B3	NaN	NaN	left_only
3	K2	K6	NaN	NaN	C3	D3	right_only

Handle duplicate key names with suffixes

If we had merged `left3` and `right3` on key1 only, there would be two columns named key2.
By default, pandas sets them up as **`key2_x`** for left data, and **`key2_y`** for right data.

However, we can assign our own suffixes :

```
pd.merge(df_left,df_right,on='key1',suffixes=('_lefty','_righty'))
```

this returns columns named **`key2_lefty`** and **`key2_righty`**

JOIN

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.join.html

Join is similar to merge, except that DataFrames are joined on their index instead of a column.

`left_df`

	A	B
K1	A1	B1
K2	A2	B2
K3	A3	B3

`right_df`

	C	D
K2	C1	D1
K3	C2	D2
K4	C3	D3

`left_df.join(right_df)` left-join by default

	A	B	C	D
K1	A1	B1	NaN	NaN
K2	A2	B2	C1	D1
K3	A3	B3	C2	D2

`left_df.join(right_df, how='inner')`

	A	B	C	D
K2	A2	B2	C1	D1
K3	A3	B3	C2	D2

Column names must be unique. If not:

`left_df.join(left_df, rsuffix='_R')` will accept `lsuffix`, `rsuffix`, or both
only duplicated column names receive the suffix

	A	B	A_R	B_R
K1	A1	B1	A1	B1
K2	A2	B2	A2	B2
K3	A3	B3	A3	B3

HANDLING OVERLAPPING DATA

See https://pandas.pydata.org/docs/reference/api/pandas.Series.combine_first.html

Concatenate, Merge and Join bring two DataFrames together with tools for mapping values from each DataFrame. However, they lack the ability to *choose* between two corresponding values. Can we choose a value over an empty cell?

```
ser1 = Series([1,2,np.nan,4],
              list('abcd'))
ser1
a    1.0
b    2.0
c    NaN
d    4.0
dtype: float64
```

```
ser2 = Series([5,6,7], list('abc'))

ser2
a    5
b    6
c    7
dtype: int64
```

```
ser1.combine_first(ser2)
a    1.0
b    2.0
c    7.0
d    4.0
dtype: float64
```

Series don't have to be the same size, and only null values in `ser1` are replaced by values in `ser2`.

DATA INPUT/OUTPUT - READING & WRITING FILES

Determine the current working directory in Jupyter

```
pwd
'C:\\Users\\Mike\\Documents\\Jupyter Notebooks'
```

We can obtain a directory listing with `ls`

Set path names

Set commonly used directories as raw data strings in the code:

```
path = r'C:\\Users\\Mike\\Documents\\Finance\\'
```

Supports use of relative paths:

```
path = r'..\\Finance\\'

url = r'https://www.statlearning.com/s/Advertising.csv'
```

CSV (Comma Separated Value) FILES

See https://pandas.pydata.org/docs/user_guide/io.html#csv-text-files

Reading .csv

See https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html

```
df = pd.read_csv('file.csv')              file.csv sits in same directory as current notebook
df = pd.read_csv(path+'file.csv')         adds a path variable (defined above) to the filename
df = pd.read_csv(url)                     accepts a url (defined above)

df = pd.read_csv('file.csv',header=None)  assigns an integer column index
df = pd.read_csv('file.csv',index_col=0)  sets the first column as the index (1 for 2nd col, etc.)
df = pd.read_csv('file.csv',nrows=2)      takes only the first two rows
```

Writing to .csv

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html

```
df.to_csv('mydataout.csv')                writes to a .csv file in the current directory
df.to_csv(path+'mydataout.csv')           writes to a .csv file in a target directory
df.to_csv('mydataout.csv', index=False)   strips the unnamed numerical index, if desired
df.to_csv('mydataout.csv', mode='x')      fails to write if the file already exists
```

Be careful! Without a `mode` argument this will truncate existing files without warning.

```
import sys
df.to_csv(sys.stdout,sep='_')                displays the output without saving it
df.to_csv(sys.stdout,columns=['c1','c2'])    sends a specific set of columns
```

Pandas provides options for handling quoted text, datetime data, and more. Refer to the docs for more info.

For info on Python's csv reader/writer visit https://docs.python.org/3/library/csv.html

EXCEL FILES Requires that **openpyxl** (for .xlsx) and **xlrd** (for .xls) be installed
See https://pandas.pydata.org/docs/user_guide/io.html#excel-files

Reading .xlsx
See https://pandas.pydata.org/docs/reference/api/pandas.read_excel.html

```
df = pd.read_excel('file.xlsx', 'Sheet1')
```

Sheets can be selected by name or position. By default, `pd.read_excel(filename)` selects the first sheet.
When passing a list of sheets, .read_excel() returns a dictionary of DataFrames. `sheet_name=None` returns all sheets.
Note that pandas takes in cell *values*, not formulas or macros.

Unnamed columns are assigned names based on position. The tenth column would be "Unnamed: 9"
Pass `header=None` to set numerical column names
Pass `index_col=0` to set the first column as the index
Pass `usecols="A,D:F,J"` to select only columns A,D,E,F,J. Ranges are inclusive.
Note: this will omit column A if `index_col=0` was passed.

OPTIONAL: Open an excel file as an object:
See https://pandas.pydata.org/docs/reference/api/pandas.ExcelFile.html

```
xlsfile = pd.ExcelFile('file.xlsx')
```

This wraps the source file into a special "ExcelFile" class object, which can then be passed to `.read_excel` either sheet by sheet or all at once. Provides a performance benefit of reading the original file only once.

```
xlsfile.sheet_names              provides a list of sheet names contained in the xlsfile object
xlsfile.parse(1)                 provides a quick view of the second sheet
df = pd.read_excel(xlsfile, 'Sheet1')
```

Using the ExcelFile class as a context manager:

```
with pd.ExcelFile('file.xlsx') as xls:
    df1 = pd.read_excel(xls, 'Sheet1')
    df2 = pd.read_excel(xls, 'Sheet2')
```

Writing to .xlsx
See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_excel.html

```
df.to_excel('output.xlsx','Sheet1')
```

Writing multiple sheets to the same Excel file
See https://pandas.pydata.org/docs/reference/api/pandas.ExcelWriter.html

This won't work:

```
df1.to_excel('output.xlsx','Sheet1')
df2.to_excel('output.xlsx','Sheet2')
```

All we will see is Sheet2.

But this will:

```
with pd.ExcelWriter('output.xlsx') as writer:
    df1.to_excel(writer,'Sheet1')
    df2.to_excel(writer,'Sheet2')
```

To append sheets to an existing file:

```
with pd.ExcelWriter('output.xlsx', mode='a') as writer:
    df3.to_excel(writer,'Sheet3')
    df4.to_excel(writer,'Sheet4')
```

JSON (JavaScript Object Notation) FILES

See https://pandas.pydata.org/docs/user_guide/io.html#json

and https://pandas.pydata.org/docs/reference/api/pandas.read_json.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_json.html

```
df = pd.read_json(obj)                  where "obj" is a valid JSON string, file, or URL
df = pd.read_json(obj,orient='split')   optional argument to correctly parse the incoming string
df.to_json(orient='index')              writes to the designated JSON string format
```

Examples:

```
df = DataFrame([['a', 'b'], ['c', 'd']],
               index=['row 1', 'row 2'],
               columns=['col 1', 'col 2'])

df.to_json(orient='split')
'{"columns":["col 1","col 2"],"index":["row 1","row 2"],"data":[["a","b"],["c","d"]]}'

pd.read_json(_, orient='split')
```

	col 1	col 2
row 1	a	b
row 2	c	d

```
df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'

pd.read_json(_, orient='index')
```

	col 1	col 2
row 1	a	b
row 2	c	d

For info on normalizing semi-structured JSON data into a flat table see
https://pandas.pydata.org/docs/reference/api/pandas.json_normalize.html

HTML FILES Requires **lxml**, **html5lib and BeautifulSoup4** be installed
See https://pandas.pydata.org/docs/user_guide/io.html#html

Reading html
See https://pandas.pydata.org/docs/reference/api/pandas.read_html.html

```
url = 'https://www.fdic.gov/resources/resolutions/bank-failures/failed-bank-list/'
df_list = pd.read_html(url)        creates a list of DataFrame objects taken from the page
df = df_list[0]                    in this example, there is no df_list[1]
df.head(3)
```

	Bank NameBank	CityCity	StateSt	CertCert	Acquiring InstitutionAI	Closing DateClosing	FundFund
0	Signature Bank	New York	NY	57053	Flagstar Bank, N.A.	March 12, 2023	10540
1	Silicon Valley Bank	Santa Clara	CA	24735	First–Citizens Bank & Trust Company	March 10, 2023	10539
2	Almena State Bank	Almena	KS	15426	Equity Bank	October 23, 2020	10538

Note: this data changes regularly, so results may vary.

The strange appearance of the column names above is due to the website's use of separate labels for full-size screens and reduced-size/mobile screens. For a workaround involving the *requests* and *BeautifulSoup* libraries visit the deep dive appendix section on Webscraping.

```
df_list = pd.read_html(url, match='Bank')     grabs every table that contains the word "Bank"
```

The following examples use this html table:

```
html_table = """
<table>
  <tr>
    <th>GitHub</th>
    <th>Some number</th>
  </tr>
  <tr>
    <td><a href="https://github.com/pandas-dev/pandas">pandas</a></td>
    <td>000456</td>
  </tr>
</table>
"""
```

```
pd.read_html(html_table)[0]
```

	GitHub	Some number
0	pandas	456

```
pd.read_html(html_table,converters={'Some number':str})[0]
```

	GitHub	Some number
0	pandas	000456

converts this column to a string to retain any leading zeroes

```
pd.read_html(html_table, extract_links="body")[0])    cells will contain a tuple of (contents, href)
```

	GitHub	Some number
0	(pandas, https://github.com/pandas-dev/pandas)	(000456, None)

numbers become strings

Be sure to read https://pandas.pydata.org/docs/user_guide/io.html#html-table-parsing-gotchas

Writing to html

See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_html.html

```
df = DataFrame({'City':['NY','LA'],'Region':['East','West']},['Amir','Beck'])
df
```

	City	Region
Amir	NY	East
Beck	LA	West

Pandas will write a DataFrame to a raw HTML text string:

```
html = df.to_html()
print(html)
```

```
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>City</th>
      <th>Region</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>Amir</th>
      <td>NY</td>
      <td>East</td>
    </tr>
    <tr>
      <th>Beck</th>
      <td>LA</td>
      <td>West</td>
    </tr>
  </tbody>
</table>
```

In Jupyter we can render the text as HTML using IPython.display:

```
from IPython.display import display, HTML
display(HTML(html))
```

	City	Region
Amir	NY	East
Beck	LA	West

Refer to the docs for an extensive list of customizable parameters.

THE CLIPBOARD

See https://pandas.pydata.org/docs/reference/api/pandas.read_clipboard.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_clipboard.html

Let's grab NFL Win-Loss data from Wikipedia:

```
import webbrowser
url = 'https://en.wikipedia.org/wiki/List_of_all-time_NFL_win-loss_records'
webbrowser.open(url)
```

Select the first five rows including the header and copy it to the clipboard:

Regular season [edit]

The following is a listing of all 32 current National Football League (NFL) teams ranked by their regular season win–loss record percentage, accurate as of the end of week 18 of the 2021 NFL season.

Best win–loss record in division

Rank	Team	GP	Won	Lost	Tied	Pct.	First NFL season	Division
1	Dallas Cowboys	964	550	408	6	.574	1960	NFC East
2	Green Bay Packers	1,418	790	590	38	.571	1921	NFC North
3	Baltimore Ravens	434	243	190	1	.561	1996	AFC North
4	New England Patriots	966	537	420	9	.561	1960	AFC East
5	Chicago Bears	1,452	786	624	42	.556	1920	NFC North
6	Miami Dolphins	882	485	393	4	.552	1966	AFC East
7	Minnesota Vikings	952	516	425	11	.548	1961	NFC North
8	Kansas City Chiefs	966	521	433	12	.546	1960	AFC West

```
df_nfl = pd.read_clipboard(sep='\t+')
df_nfl
```

	Rank	Team	GP	Won	Lost	Tied	Pct.	First NFL season	Division
0	1	Dallas Cowboys	964	550	408	6	0.574	1960	NFC East
1	2	Green Bay Packers	1,418	790	590	38	0.571	1921	NFC North
2	3	Baltimore Ravens	434	243	190	1	0.561	1996	AFC North
3	4	New England Patriots	966	537	420	9	0.561	1960	AFC East

Items of note:

- Without the `sep` argument this technique is hit-or-miss, depending on how the table is copied.
- Another option is to copy the table to Excel, save as .csv, and read the .csv file instead.
- Pandas automatically adds an index in the left-most column.

Similarly we can copy a DataFrame, or a slice of one, to the clipboard:

```
df_nfl.iloc[:3].sort_values('Team').to_clipboard()
```

We can then paste into our destination:

```
 → Rank → Team → GP → Won → Lost → Tied → Pct. → First NFL season → Division¶
2 → 3 → Baltimore Ravens → 434 → 243 → 190 → 1 → 0.561 → 1996 → AFC North¶
0 → 1 → Dallas Cowboys → 964 → 550 → 408 → 6 → 0.574 → 1960 → NFC East¶
1 → 2 → Green Bay Packers → 1,418 → 790 → 590 → 38 → 0.571 → 1921 → NFC North¶
```

Tabs and paragraph marks are shown for illustration. We can pass a `sep` argument to change the delimiter. Pasting into Excel or Google Sheets works as expected.

ADDITIONAL PANDAS OPERATIONS

REPLACE

See https://pandas.pydata.org/docs/reference/api/pandas.Series.replace.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.replace.html

We can update elements based on position using .loc or .iloc. To update elements based on value we use replace.

```
df = DataFrame(np.array(['a','b','a','A','B','A','Aa','Bb','Aa']).reshape(3,3),
               index=['a','b','c'], columns=['a','b','c'])
df
```

	a	b	c
a	a	b	a
b	A	B	A
c	Aa	Bb	Aa

```
df.replace('a','x')
```

	a	b	c
a	x	b	x
b	A	B	A
c	Aa	Bb	Aa

replaces 'a' with 'x' when 'a' is the entire entry
affects values only and ignores index and column labels

```
df.replace(['a','Aa'],['x','Xx'])
```

	a	b	c
a	x	b	x
b	A	B	A
c	Xx	Bb	Xx

replaces a collection of values with a new collection

```
df.replace(['a','Aa'],'XX')
```

	a	b	c
a	XX	b	XX
b	A	B	A
c	XX	Bb	XX

replaces a collection of values with the same value

Refer to the online docs for more information on regular expressions and padding options.

MAP

See https://pandas.pydata.org/docs/reference/api/pandas.Series.map.html

Map is used for substituting each value in a Series with another value, based on either a dictionary, Series or function. Because it is a Series method, map can be applied to DataFrame columns.

```
ser = Series(['crimson','navy','teal'])
ser
0    crimson
1       navy
2       teal
dtype: object

d = {'crimson':'red','navy':'blue'}
ser.map(d)              passing a dictionary
0     red               values that are not found in the dict are converted to NaN unless the dict
1    blue               has a default value (e.g. defaultdict)
2     NaN
dtype: object

ser.map(str.title)      passing a function
0    Crimson
1       Navy
2       Teal
dtype: object
```

Map requires that the input mapping have individual index entries / dictionary keys for each value to be replaced. Consider the following:

```
offices = DataFrame({'name':['Amir','Beck','Cleo','Drew'],
                     'city':['DC','LA','NY','SF']})
offices
```

	manager	city
0	Amir	DC
1	Beck	LA
2	Cleo	NY
3	Drew	SF

```
region_map = {'east':['NY','DC'],'west':['LA','SF','LV']}     we can't work directly from this
ser = Series(region_map).explode()
ser
east    NY
east    DC
west    LA
west    SF
west    LV
dtype: object
```

To map regions to city entries we need to pass in a Series with the index and value columns swapped:

```
offices['region'] = offices['city'].map(Series(ser.index,index=ser.values))
offices
```

	manager	city	region
0	Amir	DC	east
1	Beck	LA	west
2	Cleo	NY	east
3	Drew	SF	west

To apply a function elementwise to an entire DataFrame use `df.applymap(func)`
See: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.applymap.html

`offices.applymap(lambda x: len(str(x)))` `applymap(len)` won't work on numbers

	manager	city	region
0	4	2	4
1	4	2	4
2	4	2	4
3	4	2	4

RENAME index and column labels
See https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.rename.html

```
df=DataFrame(np.arange(2,6).reshape(2,2))
df
```

	0	1
0	2	3
1	4	5

Dictionary/Series method

```
df = df.rename(index={0:'ABC',1:'DEF'}, columns={0:'GHI',1:'JKL'})
df
```

	GHI	JKL
ABC	2	3
DEF	4	5

We can map values using either a dictionary or Series. If the mapping doesn't include a column/index label, it isn't renamed, and extra labels in the mapping don't throw an error. Labels are case sensitive.

`rename.(index={0:'a'})` is equivalent to `rename.({0:'a'}, axis=0)`.
We recommend using keyword arguments for clarity and readability.
To rename a specific MultiIndex level or levels pass in `level=`
To make permanent use `df = df.rename()` or pass `inplace=True`.

Function method

```
df.rename(index=str.title, columns=str.lower)
df
```

	ghi	jkl
Abc	2	3
Def	4	5

If we pass a function, it must return a value when called with any of the labels, and must produce a set of unique values.

For more examples see https://pandas.pydata.org/docs/user_guide/basics.html#renaming-mapping-labels

RENAME index and column names
See https://pandas.pydata.org/docs/reference/api/pandas.Index.rename.html

We recommend calling index and column separately to avoid confusion (we prefer it over df.rename_axis())

```
df.index.rename()
df.columns.rename()
```

These take a label or, in the case of a MultiIndex, a list of labels.
To make permanent use `df.index = df.index.rename()` or pass `inplace=True`.

REINDEX

See https://pandas.pydata.org/docs/reference/api/pandas.Series.reindex.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.reindex.html

Reindex lets us reorder, insert and remove rows and columns from a Series or DataFrame object.

```
ser = Series([1,2,3,4],index=['A','B','C','D'])
ser
A    1
B    2
C    3
D    4
dtype: int64
```

To remove row A, reorder B,C,D, and add new rows E,F:

```
ser1 = ser.reindex(['C','D','B','E','F'])
ser1
C    3.0
D    4.0
B    2.0
E    NaN
F    NaN
dtype: float64
```

null values convert the dtype from *int64* to *float64*

```
ser2 = ser1.reindex(['C','B','D','E','F','G'],fill_value=0)
ser2
C    3.0
D    4.0
B    2.0
E    NaN
F    NaN
G    0.0
dtype: float64
```

items 'E' and 'F' are still null values. we could use .fillna() to address this.

Inserting rows by reindexing on a DataFrame

```
df = DataFrame([[0,1],[2,3]], index=['a','c'], columns=['D','E'])
df
```

	D	E
a	0	1
c	2	3

```
df = df.reindex(['a','b','c'])
df
```

	D	E
a	0.0	1.0
b	NaN	NaN
c	2.0	3.0

Inserting columns by reindexing on a DataFrame

```
df = df.reindex(columns=['D','E','F'], fill_value=5)
df
```

	D	E	F
a	0.0	1.0	5
b	NaN	NaN	5
c	2.0	3.0	5

Propagating values between indices

```
ser = Series(['upper','middle','lower'],index=[2,5,8])
ser
2     upper
5    middle
8     lower
dtype: object
```

```
ser.reindex(range(11),
       method='ffill')
0       NaN
1       NaN
2     upper
3     upper
4     upper
5    middle
6    middle
7    middle
8     lower
9     lower
10    lower
dtype: object
```

```
ser.reindex(range(11),
       method='bfill')
0     upper
1     upper
2     upper
3    middle
4    middle
5    middle
6     lower
7     lower
8     lower
9       NaN
10      NaN
dtype: object
```

```
ser.reindex(range(11),
     method='nearest')
0     upper
1     upper
2     upper
3     upper
4    middle
5    middle
6    middle
7     lower
8     lower
9     lower
10    lower
dtype: object
```

Reordering columns by position

Reindex does not accept positional arguments. That is, `df.reindex([2,1,0])` won't work unless the index is already numeric. To reorder columns this way use `.iloc[:,[order]]`:

```
df = DataFrame([[3,5,7,2,4]],columns=list('abcde'))
df
```

	a	b	c	d	e
0	3	5	7	2	4

```
df.iloc[:,[2,4,0,1,3]]
```

	c	e	a	b	d
0	7	4	3	5	2

```
df.iloc[:,[-1,0,1,2,3]]
```

	e	a	b	c	d
0	4	3	5	7	2

To bring a handful of columns from the end to the beginning:

```
df.iloc[:,np.arange(df.shape[1])-2]
```

	d	e	a	b	c
0	2	4	3	5	7

`.iloc[]` lets us duplicate and remove columns, but we can't introduce new columns.

REINDEX_LIKE

See https://pandas.pydata.org/docs/reference/api/pandas.Series.reindex_like.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.reindex_like.html

This conforms one object to another by taking the indexes along all axes (index and column at the same time).
Similar to running `df.reindex(index=other.index, columns=other.columns,...)`
but may be clearer in some cases. Values from `other` are ignored, existing values will not change.
If index or column labels are monotonically increasing ,then we can pass a `method` to fill in holes.

```
df1 = DataFrame([[0,1],[np.nan,3]], index=['a','c'], columns=['D','E'])
df1
```

	D	E
a	0.0	1
c	NaN	3

light-blue fill is for illustration

```
df2 = DataFrame([[4,5,6],[7,8,9],[10,11,12]],['a','b','c'],['E','D','F'])
df2
```

	E	D	F
a	4	5	6
b	7	8	9
c	10	11	12

```
df1.reindex_like(df2)
```

	E	D	F
a	1.0	0.0	NaN
b	NaN	NaN	NaN
c	3.0	NaN	NaN

```
df1.reindex_like(df2, method='ffill')
```

forward-fills columm F from E, row b from a
does not affect the original NaN in row c col D

	E	D	F
a	1	0.0	1
b	1	0.0	1
c	3	NaN	3

```
df1.reindex_like(df2).interpolate()
```

pass `axis=1` to interpolate across columns

	E	D	F
a	1.0	0.0	NaN
b	2.0	0.0	NaN
c	3.0	0.0	NaN

BINNING with pandas.cut

See https://pandas.pydata.org/docs/reference/api/pandas.cut.html

Binning creates a categorical feature from a continuous variable by dividing a range of values into discrete intervals. This is the mechanism behind histograms.

Consider a Series containing years from 1990 to 2019 inclusive:

```
rng = np.random.default_rng(238)        use this seed to recreate the following example
ser = Series(rng.integers(1990,2019,size=8,endpoint=True))
ser
0    2009
1    2002
2    1998
3    2015
4    1990
5    2018
6    2008
7    2007
dtype: int64
```

If we want to divide them into three decades we might pass a scalar:

```
decades = pd.cut(ser, 3)
decades
0      (2008.667, 2018.0]
1    (1999.333, 2008.667]
2    (1989.972, 1999.333]
3      (2008.667, 2018.0]
4    (1989.972, 1999.333]
5      (2008.667, 2018.0]
6    (1999.333, 2008.667]
7    (1999.333, 2008.667]
dtype: category
Categories (3, interval[float64, right]):
           [(1989.972, 1999.333] < (1999.333, 2008.667] < (2008.667, 2018.0]]
```

Each row is assigned to a bin. The notation `()` means "open/exculsive" while `[]` means "closed/inclusive".

Due to the range of the actual data in this example, the cuts land in odd spots, and 2009 bins with 2015.
Alternatively we can pass a sequence of scalars:

```
bins = [1990, 2000, 2010, 2020]         order matters!
decades = pd.cut(ser, bins)             …otherwise use sorted(bins)
decades
0    (2000.0, 2010.0]
1    (2000.0, 2010.0]
2    (1990.0, 2000.0]
3    (2010.0, 2020.0]
4                 NaN
5    (2010.0, 2020.0]
6    (2000.0, 2010.0]
7    (2000.0, 2010.0]
dtype: category
Categories (3, interval[int64, right]):
           [(1990, 2000] < (2000, 2010] < (2010, 2020]]
```

Values that fall outside the defined bins will be excluded.
By default the left-hand edges are exclusive, so 1990 was excluded. We have two options:

```
decades = pd.cut(ser, bins, include_lowest=True)
decades
0       (2000.0, 2010.0]
1       (2000.0, 2010.0]
2    (1989.999, 2000.0]
3       (2010.0, 2020.0]
4    (1989.999, 2000.0]
5       (2010.0, 2020.0]
6       (2000.0, 2010.0]
7       (2000.0, 2010.0]
dtype: category
Categories (3, interval[float64, right]):
           [(1989.999, 2000.0] < (2000.0, 2010.0] < (2010.0, 2020.0]]

decades = pd.cut(ser, bins, right=False)
decades
0    [2000, 2010)
1    [2000, 2010)
2    [1990, 2000)
3    [2010, 2020)
4    [1990, 2000)
5    [2010, 2020)
6    [2000, 2010)
7    [2000, 2010)
dtype: category
Categories (3, interval[int64, left]):
           [[1990, 2000) < [2000, 2010) < [2010, 2020)]
```

This is the best option for our example, as we would want 2000 to bin with 2001, not 1999.

To see how many rows are assigned to each bin:

```
pd.value_counts(decades).sort_index()
[1990, 2000)    2
[2000, 2010)    4
[2010, 2020)    2
dtype: int64
```

To retrieve the specified bins:

```
decades.cat.categories
IntervalIndex([[1990, 2000), [2000, 2010), [2010, 2020)],
              dtype='interval[int64, left]')
```

To control the precision of a scalar cut:

```
halves = pd.cut(ser, 2, precision=0)
pd.value_counts(halves).sort_index()
(1990.0, 2004.0]    3
(2004.0, 2018.0]    5
dtype: int64
```

Creates 2 bins, evenly cut between the min/max values of "ser". Bin precision is set to return whole numbers (the default is 3 decimal places). Even so, 1990 is picked up whenever an integer bin selector is used.

```
age = pd.cut(ser, 2, labels=['old','new'])
pd.value_counts(age).sort_index()
old    3
new    5
dtype: int64
```

OUTLIERS

Consider a 4-column data set with 1000 rows of random values over the *standard normal* distribution [σ = 1]:

```
rng = np.random.default_rng(20)                    use this seed to recreate the following example
df = DataFrame(rng.standard_normal((1000,4)))
```

Identify rows with outliers in a specific column

We want to grab rows where the first column has an absolute value greater than 3.
In this set that's approximately equal to 3 standard deviations from the mean.

```
df[np.abs(df[0])>3]
```

	0	1	2	3
327	-3.255589	0.916954	0.902060	2.416978
719	3.112768	-0.659428	0.004861	-1.175767

Identify rows with outliers in any column

```
df[(np.abs(df)>3).any(axis=1)]
```
Outliers: grab rows where any column > 3

	0	1	2	3
218	-1.651904	0.062495	-0.279872	-3.058939
327	-3.255589	0.916954	0.902060	2.416978
472	-0.055992	0.214129	-3.590483	-1.001295
719	3.112768	-0.659428	0.004861	-1.175767
930	-0.582495	-0.648933	-0.053027	3.006934

light-blue fill is for illustration

```
df[(np.abs(df)<3).all(axis=1)]
```
Remaining data: grab rows where all columns < 3
Returns the other 995 rows

To cap data at a given threshold

Anywhere the absolute value > 3, replace with a value of 3 multiplied by the sign of the value (in place)

```
df2 = df.copy()                                    to preserve the original data
df2[np.abs(df2)>3] = np.sign(df2)*3
```

```
df2.max()                  df2.min()
0    3.000000              0   -3.000000
1    2.781240              1   -2.962198
2    2.872653              2   -3.000000
3    3.000000              3   -3.000000
dtype: float64             dtype: float64
```

This accounts for our five outliers.

Use scipy.stats to solve for the general case

Since we drew values from a standard normal distribution we're fairly confident that a value of 3 sits three standard deviations from the mean. When we're unsure of the shape of the data, we can apply statistical tools.

```
from scipy import stats
df[(np.abs(stats.zscore(df[0])) > 3)]
```

	0	1	2	3
327	-3.255589	0.916954	0.902060	2.416978
719	3.112768	-0.659428	0.004861	-1.175767

ROUNDING

See https://pandas.pydata.org/docs/reference/api/pandas.Series.round.html
and https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.round.html

For values exactly halfway between rounded decimal values, pandas rounds to the nearest even value.
Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc.

```
arr = np.array([1.445,2.555,3.999]*3)
df = DataFrame(arr.reshape([3,3]).T, ['a','b','c'], ['A','B','C'])
df
```

	A	B	C
a	1.445	1.445	1.445
b	2.555	2.555	2.555
c	3.999	3.999	3.999

```
df.round(2)
```

	A	B	C
a	1.44	1.44	1.44
b	2.56	2.56	2.56
c	4.00	4.00	4.00

We can pass a dictionary or Series of parameters mapped to column labels:

```
df.round({'A': 1, 'C': 2})
```

	A	B	C
a	1.4	1.445	1.44
b	2.6	2.555	2.56
c	4.0	3.999	4.00

```
ser = Series([1, 0, 2], index=['A','B','C'])
df.round(ser)
```

	A	B	C
a	1.4	1.0	1.44
b	2.6	3.0	2.56
c	4.0	4.0	4.00

APPENDIX I – DEEP DIVE
These tools and techniques are somewhat off the beaten path

NUMPY

Array Cartesian Product

To generate possible permutations between two 1d arrays we can use a list comprehension, since NumPy lacks a built-in Cartesian product tool like itertools.product().

```
arr1, arr2 = np.array([1,2,3]), np.array([7,8,9])
np.array([(x, y) for x in arr1 for y in arr2])
array([[1, 7],
       [1, 8],
       [1, 9],
       [2, 7],
       [2, 8],
       [2, 9],
       [3, 7],
       [3, 8],
       [3, 9]])
```

PANDAS

Series & DataFrames can hold any object, even functions (although that's rare)

```
Series([sum,print,len])
0      <built-in function sum>
1    <built-in function print>
2      <built-in function len>
dtype: object
```

Using len to group by length of index name

```
df = DataFrame(np.arange(6), ['apple','bear','cat','dog','eagle','frog'])
df
```

	0
apple	0
bear	1
cat	2
dog	3
eagle	4
frog	5

```
df.groupby(len).sum()
```

	0
3	5
4	6
5	4

Non-traditional sorting using key

pandas.DataFrame.sort_values() sorts "ascii-betically", meaning capital letters sort before lowercase letters.
We can change this using the `key` parameter (new in pandas 1.1.0):

```
df = DataFrame({'col1':list('SeRiEs')})
df
```

	col1
0	S
1	e
2	R
3	i
4	E
5	s

```
df.sort_values('col1')
```

	col1
4	E
2	R
0	S
1	e
3	i
5	s

```
df.sort_values('col1', key=lambda col: col.str.lower())
```

	col1
1	e
4	E
3	i
2	R
0	S
5	s

Copy-on-Write (CoW)

See https://pandas.pydata.org/docs/user_guide/copy_on_write.html

Introduced in pandas v1.5.0 and broadly implemented in v2.0.0, Copy-on-Write provides data integrity & optimization. CoW means that any DataFrame or Series derived from another in any way always behaves as a copy. As a consequence, we can only change the values of an object through modifying the object itself. CoW disallows updating a DataFrame or a Series that shares data with another DataFrame or Series object inplace.

To implement globally run:

```
pd.options.mode.copy_on_write = True
```

String Methods
See https://pandas.pydata.org/docs/user_guide/text.html#string-methods

object dtypes have a `.str` attribute that acts as a pathway to several string processing methods.
Method calls on non-string elements (like integers) return a NaN value.

```
ser1 = pd.Series(['Amir','Beck','Cleo'])
ser1.str.upper()        returns a modified copy of ser1, but does not change ser1 in place
0    AMIR
1    BECK
2    CLEO
dtype: object

ser2 = pd.Series(['Amir@aol.com','Beck@bbc.com','Cleo@cnn.com'])
ser2.str.split('@')
0    [Amir, aol.com]
1    [Beck, bbc.com]
2    [Cleo, cnn.com]
dtype: object

ser2.str.split('@',expand=True).rename(columns={0:'local part',1:'domain name'})
```

	local part	domain name
0	Amir	aol.com
1	Beck	bbc.com
2	Cleo	cnn.com

Stacking String Methods

```
ser3 = pd.Series(['aMIR','beck  ','  cleo'])
ser3
0      aMIR
1    beck
2      cleo
dtype: object

ser3.str.title().str.strip()      each method call must be preceded by .str
0    Amir
1    Beck
2    Cleo
dtype: object
```

The `.str` attribute and related methods can be applied to Index objects as well.

NOTE: pandas version 1.0.0 introduced an experimental **StringDtype** to use in place of **object**.
For more info visit https://pandas.pydata.org/docs/user_guide/text.html

Webscraping

pandas.read_html() failed to parse the FDIC failed bank column names well. This is my workaround.

```
url = 'https://www.fdic.gov/resources/resolutions/bank-failures/failed-bank-list/'
df_list = pd.read_html(url)
df = df_list[0]
df.head(3)
```

	Bank NameBank	CityCity	StateSt	CertCert	Acquiring InstitutionAI	Closing DateClosing	FundFund
0	Signature Bank	New York	NY	57053	Flagstar Bank, N.A.	March 12, 2023	10540
1	Silicon Valley Bank	Santa Clara	CA	24735	First–Citizens Bank & Trust Company	March 10, 2023	10539
2	Almena State Bank	Almena	KS	15426	Equity Bank	October 23, 2020	10538

Note: this data changes regularly, so results may vary.

The strange appearance of the column names above is due to the website's use of separate labels for full-size screens and reduced-size/mobile screens. Once we have a DataFrame we can use the *requests* and *BeautifulSoup* libraries to scrape the names we need:

```
import requests
from bs4 import BeautifulSoup

r = requests.get(url)
soup = BeautifulSoup(r.text, 'html.parser')
```

Since there's only one table on this website we don't need to sift through multiple `<thead>` (table header) tags. Within each table header cell, however, this site has two span classes, `"dtfullname"` and `"dtmobilename"`. We chose to collect the former.

```
table_headings = soup.find_all('th')          collect all html table header cells
columns = []
for cell in table_headings:
    columns.append(cell.find('span', class_ = "dtfullname").text)

print(columns)
['Bank Name', 'City', 'State', 'Cert', 'Acquiring Institution', 'Closing Date',
'Fund']

df.columns = columns
df.head(3)
```

	Bank Name	City	State	Cert	Acquiring Institution	Closing Date	Fund
0	Signature Bank	New York	NY	57053	Flagstar Bank, N.A.	March 12, 2023	10540
1	Silicon Valley Bank	Santa Clara	CA	24735	First–Citizens Bank & Trust Company	March 10, 2023	10539
2	Almena State Bank	Almena	KS	15426	Equity Bank	October 23, 2020	10538

APPENDIX II – POTENTIAL PITFALLS

Indexing past lexsort depth may impact performance
From https://pandas.pydata.org/docs/user_guide/advanced.html#sorting-a-multiindex:

For MultiIndex-ed objects to be indexed and sliced effectively, they need to be sorted.
Indexing will work even if the data are not sorted, but will be rather inefficient (and show a PerformanceWarning).
It will also return a copy of the data rather than a view:

```
dfm = DataFrame({'L1':[0,0,1,1], 'L2':['x','x','z','y'], 'data':[1,1,1,1]})
dfm = dfm.set_index(['L1', 'L2'])
dfm
```

		data
L1	**L2**	
0	x	1
	x	1
1	z	1
	y	1

```
dfm.loc[(1, 'z')]
PerformanceWarning: indexing past lexsort depth may impact performance.
```

		data
L1	**L2**	
1	z	1

Index level names should be unique from column names
From https://pandas.pydata.org/docs/user_guide/groupby.html#splitting-an-object-into-groups:

A string passed to groupby may refer to either a column or an index level. If a string matches both a column name and an index level name, a ValueError will be raised.

APPENDIX III – THINGS WE CHOSE NOT TO INCLUDE

We feel that some tools have limited utility. This section explains why they were not included in the main text.

pandas.DataFrame.assign()

Lets you assign new columns to a DataFrame. The following two statements are equivalent:

```
df['C'] = df['A'] + df['B']
```

```
df = df.assign(C = df['A'] + df['B'])
```

The only benefit .assign() has over traditional assignment is we can chain multiple assignments into one statement, and we can reuse columns just created:

```
df = df.assign(C = df['A'] + df['B'],
               D = df['C'] * 1.5)
```

pandas.DataFrame.cummax()
pandas.DataFrame.cummin()

Returns a DataFrame or Series of the same size with the cumulative maximum/minimum values over the specified axis. We're having trouble coming up with a use case.

pandas.DataFrame.rename_axis()
pandas.Index.set_names()

As far as we can tell, offers the same functionality as `df.index.rename()` and `df.columns.rename()` with no added benefit. Using a dictionary mapper on just one or two names doesn't seem worth it.

numpy.ndarray.swapaxes()
pandas.DataFrame.swapaxes()

As far as we can tell, offers the same functionality as `.transpose()` with no added benefit, and has the added hassle of requiring axes arguments.

pandas.DataFrame.take()

As far as we can tell, offers the same functionality as `.iloc[]` with no added benefit. From this StackOverflow answer: `.iloc` can index into rows AND columns at the same time. `.take` can only select from one or the other. `.take` always returns a DataFrame with the same number of levels in both axes. In contrast, `.iloc` can easily return fewer levels, to the point of totally returning a Series instead of a DataFrame. Also, `.take` always returns a copy. This means that you can also use `.iloc` for assignment, but this is not the case for `.take`.

pandas.read_table()

The difference between .read_table() and .read_csv() is semantics.
Where .read_csv() defaults to sep=',' (comma-delimited), .read_table() defaults to sep='\t' (tab-delimited).

numpy.chararray

Not recommended. From the docs:
The chararray class exists for backwards compatibility with Numarray; it is not recommended for new development. Starting from numpy 1.4, if one needs arrays of strings, it is recommended to use arrays of dtype object_, string_ or unicode_, and use the free functions in the numpy.char module for fast vectorized string operations.

We omitted strings from the NumPy section on array types. We feel pandas handles them better.

APPENDIX IV – ADDITIONAL RESOURCES

Good source of NumPy practice exercises: https://github.com/rougier/numpy-100

Pandas cookbook - *a repository for short and sweet examples and links for useful pandas recipes*
https://pandas.pydata.org/docs/user_guide/cookbook.html

Pandas Cheat Sheet: https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf