

Section 1: Pandas Foundations

Accessing the main DataFrame components

When possible, Index objects are implemented using hash tables that allow for very fast selection and data alignment. They are similar to Python sets in that they support operations such as intersection and union, but are dissimilar because they are ordered with duplicates allowed.

Python dictionaries and sets are also implemented with hash tables that allow for membership checking to happen very fast in constant time, regardless of the size of the object.

Notice how the values DataFrame attribute returned a NumPy n-dimensional array, or ndarray. Most of pandas relies heavily on the ndarray. Beneath the index, columns, and data are NumPy ndarrays. They could be considered the base object for pandas that many other objects are built upon. To see this, we can look at the values of the index and columns:

```
>>> index.values
array([ 0, 1, 2, ..., 4913, 4914, 4915])
>>> columns.values
array(['color', 'director_name', 'num_critic_for_reviews',
...,
'imdb_score', 'aspect_ratio', 'movie_facebook_likes'],
dtype=object)
```

See also:

- Pandas official documentation of *Indexing and Selecting data* (<http://bit.ly/2vm8f12>)
- *A look inside pandas design and development* slide deck from pandas author, Wes McKinney (<http://bit.ly/2u4YVLi>)

Understanding data types

Almost all of pandas data types are built directly from NumPy. This tight integration makes it easier for users to integrate pandas and NumPy operations. As pandas grew larger and more popular, the object data type proved to be too generic for all columns with string values. Pandas created its own categorical data type to handle columns of strings (or numbers) with a fixed number of possible values.

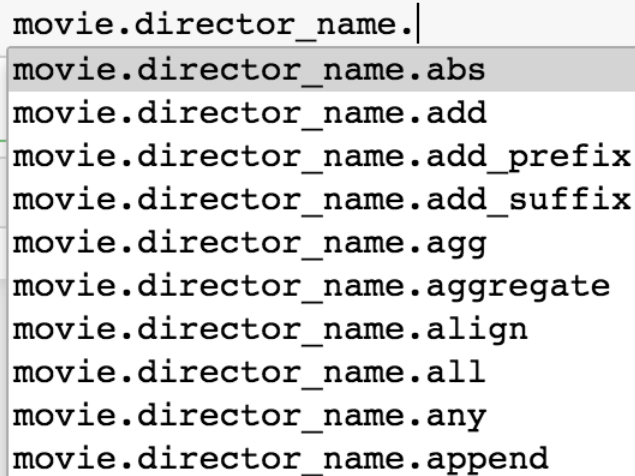
See also:

- Pandas official documentation for *dtypes* (<http://bit.ly/2vxe8Zl>)
- NumPy official documentation for *Data types* (<http://bit.ly/2wq0qEH>)

Selecting a single column of data as a Series

Why would anyone ever use the dot notation syntax if it causes trouble? Programmers are lazy, and there are fewer characters to type. But mainly, it is extremely handy when you want to have the autocomplete intelligence available. For this reason, column selection by dot notation will sometimes be used in this course. The autocomplete intelligence is fantastic for helping you become aware of all the possible attributes and methods available to an object.

The intelligence will fail to work when attempting to chain an operation after use of the indexing operator from step 1 but will continue to work with the dot notation from step 2. The following screenshot shows the pop-up window that appears after the selection of the `director_name` with the dot notation. All the possible attributes and methods will appear in a list after pressing **Tab** following the dot:



```
movie.director_name.|
movie.director_name.abs
movie.director_name.add
movie.director_name.add_prefix
movie.director_name.add_suffix
movie.director_name.agg
movie.director_name.aggregate
movie.director_name.align
movie.director_name.all
movie.director_name.any
movie.director_name.append
```

In a Jupyter notebook, when holding down *Shift + Tab + Tab* with the cursor placed somewhere in the object, a window of the docsstrings will pop out making the method far easier to use. This intelligence again disappears if you try to chain an operation after selecting a column with the indexing operator.

Yet another reason to be aware of the dot notation is the proliferation of its use online at the popular question and answer site Stack Overflow. Also, notice that the old column name is now the name of the Series and has actually become an attribute:

```
>>> director = movie['director_name']
>>> director.name
'director_name'
```

It is possible to turn this Series into a one-column DataFrame with the `to_frame` method. This method will use the Series name as the new column name:

```
>>> director.to_frame()
```

See also

- To understand how Python objects gain the capability to use the indexing operator, see the Python documentation on the `__getitem__` special method (<http://bit.ly/2u5ISN6>)
- Refer to the video, *Selecting multiple DataFrame columns* from Section 2, *Essential DataFrame operations*

Calling Series methods

The `value_counts` method is one of the most informative Series methods and heavily used during exploratory analysis, especially with categorical columns. It defaults to returning the counts, but by setting the `normalize` parameter to `True`, the relative frequencies are returned instead, which provides another view of the distribution:

```
>>> director.value_counts(normalize=True)
Steven Spielberg 0.005401
Woody Allen 0.004570
Martin Scorsese 0.004155
Clint Eastwood 0.004155
...
Fatih Akin 0.000208
Analeine Cal y Mayor 0.000208
Andrew Douglas 0.000208
Scott Speer 0.000208
Name: director_name, Length: 2397, dtype: float64
```

In this video, we determined that there were missing values in the Series by observing that the result from the count method did not match the size attribute. A more direct approach is to use the `hasnans` attribute:

```
>>> director.hasnans
True
```

There exists a complement of `isnull`: the `notnull` method, which returns `True` for all the non-missing values:

```
>>> director.notnull()
0 True
1 True
2 True
3 True
...
4912 False
4913 True
4914 True
4915 True
Name: director_name, Length: 4916, dtype: bool
```

See also

- To call many Series methods in succession, refer to the video, *Chaining Series methods together* in this section

Working with operators on a Series

All of the operators used in this video have method equivalents that produce the exact same result. For instance, in step 1, `imdb_score + 1` may be reproduced with the `add` method. Check the following code to see the method version of each step in the video:

```
>>> imdb_score.add(1) # imdb_score + 1
>>> imdb_score.mul(2.5) # imdb_score * 2.5
>>> imdb_score.floordiv(7) # imdb_score // 7
```

```
>>> imdb_score.gt(7) # imdb_score > 7
>>> director.eq('James Cameron') # director == 'James Cameron'
```

Why does pandas offer a method equivalent to these operators? By its nature, an operator only operates in exactly one manner. Methods, on the other hand, can have parameters that allow you to alter their default functionality:

Operator Group	Operator	Series method name
Arithmetic	+, -, *, /, //, %, **	add, sub, mul, div, floordiv, mod, pow
Comparison	<, >, <=, >=, ==, !=	lt, gt, le, ge, eq, ne

You may be curious as to how a Python Series object, or any object for that matter, knows what to do when it encounters an operator. For example, how does the expression `imdb_score * 2.5` know to multiply each element in the Series by 2.5? Python has a built-in, standardized way for objects to communicate with operators using **special methods**.

Special methods are what objects call internally whenever they encounter an operator. Special methods are defined in the Python data model, a very important part of the official documentation, and are the same for every object throughout the language. Special methods always begin and end with two underscores. For instance, the special method `__mul__` is called whenever the multiplication operator is used. Python interprets the `imdb_score * 2.5` expression as `imdb_score.__mul__(2.5)`. There is no difference between using the special method and using an operator as they are doing the exact same thing. The operator is just syntactic sugar for the special method.

See also

- Python official documentation on operators (<http://bit.ly/2wpOld8>)
- Python official documentation on the data model (<http://bit.ly/2v0LrDd>)

Chaining Series methods together

Instead of summing up the booleans in step 3 to find the total number of missing values, we can take the mean of the Series to get the percentage of values that are missing:

```
>>> actor_1_fb_likes.isnull().mean()
0.0014
```

As was mentioned at the beginning of the video, it is possible to use parentheses instead of the backslash for multi-line code. Step 4 may be rewritten this way:

```
>>> (actor_1_fb_likes.fillna(0)
     .astype(int)
     .head())
```

Not all programmers like the use of method chaining, as there are some downsides. One such downside is that debugging becomes difficult. None of the intermediate objects produced during the chain are stored in a variable, so if there is an unexpected result, it will be difficult to trace the exact location in the chain where it occurred. The example at the start of the video may be rewritten so that the result of each method gets preserved as/in a unique variable. This makes tracking bugs much easier, as you can inspect the object at each step:

```
>>> person1 = person.drive('store')
>>> person2 = person1.buy('food')
>>> person3 = person2.drive('home')
>>> person4 = person3.prepare('food')
>>> person5 = person4.cook('food')
>>> person6 = person5.serve('food')
>>> person7 = person6.eat('food')
>>> person8 = person7.cleanup('dishes')
```

Renaming row and column names

There are multiple ways to rename row and column labels. It is possible to reassign the index and column attributes directly to a Python list. This assignment works when the list has the same number of elements as the row and column labels. The following code uses the `tolist` method on each Index object to create a Python list of labels. It then modifies a couple values in the list and reassigns the list to the attributes `index` and `columns`:

```
>>> movie = pd.read_csv('data/movie.csv', index_col='movie_title')
>>> index = movie.index
>>> columns = movie.columns
>>> index_list = index.tolist()
>>> column_list = columns.tolist()
# rename the row and column labels with list assignments
>>> index_list[0] = 'Ratava'
>>> index_list[2] = 'Ertceps'
>>> column_list[1] = 'Director Name'
>>> column_list[2] = 'Critical Reviews'
>>> print(index_list)
['Ratava', 'Pirates of the Caribbean: At World's End', 'Ertceps', 'The Dark
Knight Rises', ... ]
>>> print(column_list)
['color', 'Director Name', 'Critical Reviews', 'duration', ...]
# finally reassign the index and columns
>>> movie.index = index_list
>>> movie.columns = column_list
```

Creating and deleting columns

It is possible to insert a new column into a specific place in a DataFrame besides the end with the `insert` method. The `insert` method takes the integer position of the new column as its first argument, the name of the new column as its second, and the values as its third. You will need to use the `get_loc` Index method to find the integer location of the column name.

The `insert` method modifies the calling DataFrame in-place, so there won't be an assignment statement. The profit of each movie may be calculated by subtracting budget from gross and inserting it directly after gross with the following:

```
>>> profit_index = movie.columns.get_loc('gross') + 1
>>> profit_index
9
>>> movie.insert(loc=profit_index,
column='profit',
value=movie['gross'] - movie['budget'])
```

An alternative to deleting columns with the drop method is to use the del statement:

```
>>> del movie['actor_director_facebook_likes']
```

See also

- Refer to the video, *Appending new rows to DataFrames* from Section 9, *Combining Pandas Objects* for adding and deleting rows, which is a less common operation
- Refer to the video, *Developing a data analysis routine* from Section 3, *Beginning Data Analysis*

Section 2: Essential DataFrame Operations

Selecting columns with methods

The filter method comes with another parameter, items, which takes a list of exact column names. This is nearly an exact duplication of the indexing operator, except that a KeyError will not be raised if one of the strings does not match a column name. For instance, movie.filter(items=['actor_1_name', 'asdf']) runs without error and returns a single column DataFrame.

One confusing aspect of select_dtypes is its flexibility to take both strings and Python objects. The following table should clarify all the possible ways to select the many different column data types. There is no standard or preferred method of referring to data types in pandas, so it's good to be aware of both ways:

Python object	String	Notes
np.number	number	Selects both integers and floats regardless of size
np.float64, np.float_, float	float64, float_, float	Selects only 64-bit floats
np.float16, np.float32, np.float128	float16, float32, float128	Respectively selects exactly 16, 32, and 128-bit floats
np.floating	floating	Selects all floats regardless of size
np.int0, np.int64, np.int_, int	int0, int64, int_, int	Selects only 64-bit integers
np.int8, np.int16, np.int32	int8, int16, int32	Respectively selects exactly 8, 16, and 32-bit integers
np.integer	integer	Selects all integers regardless of size
np.object	object, O	Select all object data types
np.datetime64	datetime64, datetime	All datetimes are 64 bits
np.timedelta64	timedelta64, timedelta	All timedeltas are 64 bits
pd.Categorical	category	Unique to pandas; no NumPy equivalent

Because all integers and floats default to 64 bits, you may select them by simply using the string, *int*, or *float* as you can see from the preceding table. If you want to select all integers and floats regardless of their specific size use the string *number*.

See also

- Refer to the video, *Understanding data types* from Section 1, *Pandas Foundations*
- The rarely used select method may also select columns based on their names (<http://bit.ly/2fchzhu>)

Ordering column names sensibly

There are alternative guidelines for ordering columns besides the simple suggestion mentioned earlier. Hadley Wickham's seminal paper on Tidy Data suggests placing the fixed variables first, followed by measured variables. As this data does not emanate from a controlled experiment, there is some flexibility in determining which variables are fixed and which ones are measured. Good candidates for measured variables are those that we would like to predict such as gross, the total revenue, or the imdb_score. For instance, in this ordering, we can mix discrete and continuous variables. It might make more sense to place the column for the number of Facebook likes directly after the name of that actor. You can, of course, come up with your own guidelines for column order as the computational parts are unaffected by it.

Quite often, you will be pulling data directly from a relational database. A very common practice for relational databases is to have the primary key (if it exists) as the first column and any foreign keys directly following it.

Primary keys uniquely identify rows in the current table. Foreign keys uniquely identify rows in other tables.

See also

- Hadley Wickham's paper on *Tidy Data* (<http://bit.ly/2v1hvH5>)

Chaining DataFrame methods together

Most of the columns in the movie dataset with object data type contain missing values. By default, the aggregation methods, min, max, and sum, do not return anything, as seen in the following code snippet, which selects three object columns and attempts to find the maximum value of each one:

```
>>> movie[['color', 'movie_title', 'color']].max()
Series([], dtype: float64)
```

To force pandas to return something for each column, we must fill in the missing values. Here, we choose an empty string:

```
>>> movie.select_dtypes(['object']).fillna("").min()
color           Color
director_name   Etienne Faure
actor_2_name     Zubaida Sahar
genres          Western
actor_1_name     Oscar Jaenada
movie_title      Æon Flux
actor_3_name     Oscar Jaenada
plot_keywords    zombie|zombie spoof
movie_imdb_link  http://www.imdb.com/title/tt5574490/?ref_=fn_t...
Language        Zulu
country         West Germany
```

```
content_rating    X
dtype: object
```

For purposes of readability, method chains are often written as one method call per line with the backslash character at the end to escape new lines. This makes it easier to read and insert comments on what is returned at each step of the chain:

```
>>> # rewrite the above chain on multiple lines
>>> movie.select_dtypes(['object']) \
.fillna("") \
.min()
```

It is atypical to aggregate a column of all strings, as the minimum and maximum values are not universally defined. Attempting to call methods that clearly have no string interpretation, such as finding the mean or variance, will not work.

See also

- Refer to the video, *Chaining Series methods together* in Section 1, *Pandas Foundations*

Working with operators on a DataFrame

Just as with Series, DataFrames have method equivalents of the operators. You may replace the operators with their method equivalents:

```
>>> college_ugds_op_round_methods = college_ugds_.add(.00501) \
.floordiv(.01) \
.div(100)
>>> college_ugds_op_round_methods.equals(college_ugds_round)
True
```

See also

- What every computer scientist should know about floating-point arithmetic (<http://bit.ly/2vmYZKi>)

Comparing missing values

All the comparison operators have method counterparts that allow for more functionality. Somewhat confusingly, the `eq` DataFrame method does element-by-element comparison, just like the `equals` operator. The `eq` method is not at all the same as the `equals` method. It merely does a similar task as the `equals` operator. The following code duplicates step 1:

```
>>> college_ugds_.eq(.0019) # same as college_ugds_ == .0019
```

Inside the `pandas.testing` sub-package, a function exists that developers must use when creating unit tests. The `assert_frame_equal` function raises an `AssertionError` if two DataFrames are not equal. It returns `None` if the two passed frames are equal:

```
>>> from pandas.testing import assert_frame_equal
>>> assert_frame_equal(college_ugds_, college_ugds_)
```

Unit tests are a very important part of software development and ensure that the code is running correctly. Pandas contains many thousands of unit tests that help ensure that it is running properly.

See also

- To read more on how pandas runs its unit tests, see the *Contributing to pandas* section in the documentation (<http://bit.ly/2vmCSU6>)

Transposing the direction of a DataFrame operation

The `cumsum` method with `axis=1` accumulates the race percentages across each row. It gives a slightly different view of the data. For example, it is very easy to see the exact percentage of white, black, and Hispanic together for each school:

```
>> college_ugds_cumsum = college_ugds_.cumsum(axis=1)
>>> college_ugds_cumsum.head()
```

	UGDS_WHITE	UGDS_BLACK	UGDS_HISP	UGDS_ASIAN	UGDS_AIAN	UGDS_NHPI	UGDS_2MOR	UGDS_NRA	UGDS_UNKN
INSTNM									
Alabama A & M University	0.0333	0.9686	0.9741	0.9760	0.9784	0.9803	0.9803	0.9862	1.0000
University of Alabama at Birmingham	0.5922	0.8522	0.8805	0.9323	0.9345	0.9352	0.9720	0.9899	0.9999
Amridge University	0.2990	0.7182	0.7251	0.7285	0.7285	0.7285	0.7285	0.7285	1.0000
University of Alabama in Huntsville	0.6988	0.8243	0.8625	0.9001	0.9144	0.9146	0.9318	0.9650	1.0000
Alabama State University	0.0158	0.9366	0.9487	0.9506	0.9516	0.9522	0.9620	0.9863	1.0000

See also

- Pandas official documentation for `cumsum` (<http://bit.ly/2v3B6EZ>)

Determining college campus diversity

Alternatively, we can find the schools that are least diverse by ordering them by their maximum race percentage:

```
>>> college_ugds_.max(axis=1).sort_values(ascending=False).head(10)
INSTNM
Dewey University-Manati                1.0
Yeshiva and Kollel Harbotzas Torah      1.0
Mr Leon's School of Hair Design-Lewiston 1.0
Dewey University-Bayamon                1.0
Shepherds Theological Seminary          1.0
Yeshiva Gedolah Kesser Torah            1.0
Monteclaro Escuela de Hoteleria y Artes Culinarias 1.0
Yeshiva Shaar Hatorah                   1.0
Bais Medrash Elyon                     1.0
Yeshiva of Nitra Rabbinical College     1.0
dtype: float64
```

We can also determine if any school has all nine race categories exceeding 1%:

```
>>> (college_ugds_ > .01).all(axis=1).any()
True
```

See also

- US News Campus Ethnic Diversity 2015-2016 (<http://bit.ly/2vmDhWC>)

Section 3: Beginning Data Analysis

Developing a data analysis routine

A crucial part of a data analysis involves creating and maintaining a data dictionary. A data dictionary is a table of metadata and notes on each column of data. One of the primary purposes of a data dictionary is to explain the meaning of the column names. The college dataset uses a lot of abbreviations that are likely to be unfamiliar to an analyst who is inspecting it for the first time. A data dictionary for the college dataset is provided in the following `college_data_dictionary.csv` file:

```
>>> pd.read_csv('data/college_data_dictionary.csv')
```

	column_name	description
0	INSTNM	Institution Name
1	CITY	City Location
2	STABBR	State Abbreviation
3	HBCU	Historically Black College or University
...
23	PCTFLOAN	Percent Students with federal loan
24	UG25ABV	Percent Students Older than 25
25	MD_EARN_WNE_P10	Median Earnings 10 years after enrollment
26	GRAD_DEBT_MDN_SUPP	Median debt of completers

As you can see, it is immensely helpful in deciphering the abbreviated column names. DataFrames are actually not the best place to store data dictionaries. A platform such as Excel or Google Sheets with easy ability to edit values and append columns is a better choice. Minimally, a column to keep track of notes on the data should be included in a data dictionary. A data dictionary is one of the first things that you can share as an analyst to collaborators.

It will often be the case that the dataset you are working with originated from a database whose administrators you will have to contact in order to get more information. Formal electronic databases generally have more formal representations of their data, called **schemas**. If possible, attempt to investigate your dataset with people who have expert knowledge on its design.

See also

- NumPy data hierarchy documentation (<http://bit.ly/2yqsg7p>)

Reducing memory by changing data types

To get a better idea of how object data type columns differ from integers and floats, a single value from each one of these columns can be modified and the resulting memory usage displayed. The `CURROPER` and `INSTNM` columns are of `int64` and object types, respectively:

```
>>> college.loc[0, 'CURROPER'] = 10000000
>>> college.loc[0, 'INSTNM'] = college.loc[0, 'INSTNM'] + 'a'
>>> college[['CURROPER', 'INSTNM']].memory_usage(deep=True)
Index 80
CURROPER 60280
INSTNM 660345
```

Memory usage for CURROPER remained the same since a 64-bit integer is more than enough space for the larger number. On the other hand, the memory usage for INSTNM increased by 105 bytes by just adding a single letter to one value.

Python 3 uses Unicode, a standardized character representation intended to encode all the world's writing systems. Unicode uses up to 4 bytes per character. It seems that pandas has some overhead (100 bytes) when making the first modification to a character value. Afterward, increments of 5 bytes per character are sustained.

Not all columns can be coerced to the desired type. Take a look at the MENONLY column, which from the data dictionary appears to contain only 0/1 values. The actual data type of this column upon import unexpectedly turns out to be float64. The reason for this is that there happen to be missing values, denoted by np.nan. There is no integer representation for missing values. Any numeric column with even a single missing value must be a float. Furthermore, any column of an integer data type will automatically be coerced to a float if one of the values becomes missing:

```
>>> college['MENONLY'].dtype
dtype('float64')
>>> college['MENONLY'].astype(np.int8)
ValueError: Cannot convert non-finite values (NA or inf) to integer
```

Additionally, it is possible to substitute string names in place of Python objects when referring to data types. For instance, when using the include parameter in the describe DataFrame method, it is possible to pass a list of either the formal object NumPy/pandas object or their equivalent string representation. These are available in the table at the beginning of the video, *Selecting columns with methods* in Section 2, *Essential DataFrame Operations*. For instance, each of the following produces the same result:

```
>>> college.describe(include=['int64', 'float64']).T
>>> college.describe(include=[np.int64, np.float64]).T
>>> college.describe(include=['int', 'float']).T
>>> college.describe(include=['number']).T
```

These strings can be similarly used when changing types:

```
>>> college['MENONLY'] = college['MENONLY'].astype('float16')
>>> college['RELAFFIL'] = college['RELAFFIL'].astype('int8')
```

The equivalence of a string and the outright pandas or NumPy object occurs elsewhere in the pandas library and can be a source of confusion as there are two different ways to access the same thing. Lastly, it is possible to see the enormous memory difference between the minimal RangeIndex and Int64Index, which stores every row index in memory:

```
>>> college.index = pd.Int64Index(college.index)
>>> college.index.memory_usage() # previously was just 80
60280
```

See also

- Pandas official documentation on data types (<http://bit.ly/2vxe8Zl>)

Selecting the largest of each group by sorting

It is possible to sort one column in ascending order while simultaneously sorting another column in descending order. To accomplish this, pass in a list of booleans to the ascending parameter that corresponds to how you would like each column sorted. The following sorts `title_year` and `content_rating` in descending order and `budget` in ascending order. It then finds the lowest budget film for each year and content rating group:

```
>>> movie4 = movie[['movie_title', 'title_year',  
'content_rating', 'budget']]  
>>> movie4_sorted = movie4.sort_values(['title_year',  
'content_rating', 'budget'],  
ascending=[False, False, True])  
>>> movie4_sorted.drop_duplicates(subset=['title_year',  
'content_rating']).head(10)
```

	movie_title	title_year	content_rating	budget
4108	Compadres	2016.0	R	3000000.0
4772	Fight to the Finish	2016.0	PG-13	150000.0
4775	Rodeo Girl	2016.0	PG	500000.0
3309	The Wailing	2016.0	Not Rated	NaN
4773	Alleluia! The Devil's Carnival	2016.0	NaN	500000.0
4848	Bizarre	2015.0	Unrated	500000.0
821	The Ridiculous 6	2015.0	TV-14	NaN
4956	The Gallows	2015.0	R	100000.0
4948	Romantic Schemer	2015.0	PG-13	125000.0
3868	R.L. Stine's Monsterville: The Cabinet of Souls	2015.0	PG	4400000.0

By default, `drop_duplicates` keeps the very first appearance, but this behavior may be modified by passing the `keep` parameter *last* to select the last row of each group or *False* to drop all duplicates entirely.

Replicating `nlargest` with `sort_values`

If you look at the `nlargest` documentation, you will see that the `keep` parameter has three possible values, *first*, *last*, and *False*. From my knowledge of other pandas methods, `keep=False` should allow all ties to remain part of the result. Unfortunately, pandas raises an error when attempting to do this. I created an issue with pandas development team on GitHub to make this enhancement (<http://bit.ly/2fGrCMa>).

Section 4: Selecting Subsets of Data

Selecting Series data

When passing a scalar value to the indexing operator, as with step 2 and step 5, a scalar value is returned. When passing a list or slice, as in the other steps, a Series is returned. This returned value might seem inconsistent, but if we think of a Series as a dictionary-like object that maps labels to values, then returning the value makes sense. To select a single item and retain the item in its Series, pass in as a single-item list rather than a scalar value:

```
>>> city.iloc[[3]]
INSTNM
University of Alabama in Huntsville Huntsville
Name: CITY, dtype: object
```

Care needs to be taken when using slice notation with `.loc`. If the start index appears after the stop index, then an empty Series is returned without an exception raised:

```
>>> city.loc['Reid State Technical College':
'Alabama State University':10]
Series([], Name: CITY, dtype: object)
```

See also

- Pandas official documentation on indexing (<http://bit.ly/2fdtZWu>)

Selecting DataFrame rows and columns simultaneously

When selecting a subset of rows, along with all the columns, it is not necessary to use a colon following a comma. The default behavior is to select all the columns if there is no comma present. The previous video selected rows in exactly this manner. You can, however, use a colon to represent a slice of all the columns. The following lines of code are equivalent:

```
>>> college.iloc[:10]
>>> college.iloc[:10, :]
```

Slicing lexicographically

With this video, it is easy to select colleges between two letters of the alphabet. For instance, to select all colleges that begin with the letter D through S, you would use `college.loc['D':'T']`. Slicing like this is still inclusive of the last index so this would technically return a college with the exact name T.

This type of slicing also works when the index is sorted in the opposite direction. You can determine which direction the index is sorted with the `index` attribute, `is_monotonic_increasing` or `is_monotonic_decreasing`. Either of these must be True in order for lexicographic slicing to work. For instance, the following code lexicographically sorts the index from Z to A:

```
>>> college = college.sort_index(ascending=False)
>>> college.index.is_monotonic_decreasing
True
>>> college.loc['E':'B']
```

	CITY	STABBR	HBCU	MENONLY	WOMENONLY	RELAFFIL	SATVRMID	SATMTMID	DISTANCEONLY	UGDS	...	UGDS_2MOR
INSTNM												
Dyersburg State Community College	Dyersburg	TN	0.0	0.0	0.0	0	NaN	NaN	0.0	2001.0	...	0.0185
Dutchess Community College	Poughkeepsie	NY	0.0	0.0	0.0	0	NaN	NaN	0.0	6885.0	...	0.0446
Dutchess BOCES-Practical Nursing Program	Poughkeepsie	NY	0.0	0.0	0.0	0	NaN	NaN	0.0	155.0	...	0.0581
...
BJ's Beauty & Barber College	Auburn	WA	0.0	0.0	0.0	0	NaN	NaN	0.0	28.0	...	0.0714
BIR Training Center	Chicago	IL	0.0	0.0	0.0	0	NaN	NaN	0.0	2132.0	...	0.0000
B M Spurr School of Practical Nursing	Glen Dale	WV	0.0	0.0	0.0	0	NaN	NaN	0.0	31.0	...	0.0000

1411 rows x 26 columns

Python sorts all capital letters before lowercase and all integers before capital letters.

Section 5: Boolean Indexing

Constructing multiple boolean conditions

A consequence of pandas using different syntax for the logical operators is that operator precedence is no longer the same. The comparison operators have a higher precedence than and, or, and not. However, the new operators for pandas (the bitwise operators &, |, and ~) have a higher precedence than the comparison operators, thus the need for parentheses. An example can help clear this up. Take the following expression:

```
>>> 5 < 10 and 3 > 4
False
```

In the preceding expression, 5 < 10 evaluates first, followed by 3 < 4, and finally, the and evaluates. Python progresses through the expression as follows:

```
>>> 5 < 10 and 3 > 4
>>> True and 3 > 4
>>> True and False
>>> False
```

Let's take a look at what would happen if the expression in criteria3 was written as follows:

```
>>> movie.title_year < 2000 | movie.title_year > 2009
TypeError: cannot compare a dtyped [float64] array with a scalar of type
[bool]
```

As the bitwise operators have higher precedence than the comparison operators, `2000 | movie.title_year` is evaluated first, which is nonsensical and raises an error. Therefore, parentheses are needed to have the operations evaluated in the correct order. Why can't pandas use `and`, `or`, and `not`? When these keywords are evaluated, Python attempts to find the **truthiness** of the objects as a whole. As it does not make sense for a Series as a whole to be either True or False--only each element--pandas raises an error.

Many objects in Python have boolean representation. For instance, all integers except 0 are considered True. All strings except the empty string are True. All non-empty sets, tuples, dictionaries, and lists are True. An empty DataFrame or Series does not evaluate as True or False and instead an error is raised. In general, to retrieve the truthiness of a Python object, pass it to the `bool` function.

See also

- Python operator precedence (<http://bit.ly/2vxuqSn>)

Filtering with boolean indexing

As was stated earlier, it is possible to use one long boolean expression in place of several other shorter ones. To replicate the `final_crit_a` variable from step 1 with one long line of code, we can do the following:

```
>>> final_crit_a2 = (movie.imdb_score > 8) & \
(movie.content_rating == 'PG-13') & \
((movie.title_year < 2000) |
(movie.title_year > 2009))
>>> final_crit_a2.equals(final_crit_a)
True
```

See also

- Pandas official documentation on *boolean indexing* (<http://bit.ly/2v1xK77>)
- Checking the truth of a Python object (<http://bit.ly/2vn8WXX>)

Selecting with unique and sorted indexes

Boolean selection gives much more flexibility than index selection as it is possible to condition on any number of columns. In this video, we used a single column as the index. It is possible to concatenate multiple columns together to form an index. For instance, in the following code, we set the index equal to the concatenation of the city and state columns:

```
>>> college.index = college['CITY'] + ', ' + college['STABBR']
>>> college = college.sort_index()
>>> college.head()
```

	INSTNM	CITY	STABBR	HBCU	MENONLY	WOMENONLY	RELAFFIL	SATVRMID	SATMTMID	DISTANCEONLY	...	UGDS_2MOR	I
	ARTESIA, CA	Angeles Institute	ARTESIA	CA	0.0	0.0	0.0	0	NaN	NaN	0.0 ...	0.0175	
	Aberdeen, SD	Presentation College	Aberdeen	SD	0.0	0.0	0.0	1	440.0	480.0	0.0 ...	0.0284	
	Aberdeen, SD	Northern State University	Aberdeen	SD	0.0	0.0	0.0	0	480.0	475.0	0.0 ...	0.0219	
	Aberdeen, WA	Grays Harbor College	Aberdeen	WA	0.0	0.0	0.0	0	NaN	NaN	0.0 ...	0.0937	
	Abilene, TX	Hardin-Simmons University	Abilene	TX	0.0	0.0	0.0	1	508.0	515.0	0.0 ...	0.0298	

From here, we can select all colleges from a particular city and state combination without boolean indexing. Let's select all colleges from Miami, FL:

```
>>> college.loc['Miami, FL'].head()
```

	INSTNM	CITY	STABBR	HBCU	MENONLY	WOMENONLY	RELAFFIL	SATVRMID	SATMTMID	DISTANCEONLY	...	UGDS_2MOR	
	Miami, FL	New Professions Technical Institute	Miami	FL	0.0	0.0	0.0	0	NaN	NaN	0.0 ...	0.0000	
	Miami, FL	Management Resources College	Miami	FL	0.0	0.0	0.0	0	NaN	NaN	0.0 ...	0.0000	
	Miami, FL	Strayer University-Doral	Miami	FL	NaN	NaN	NaN	1	NaN	NaN	NaN ...	NaN	
	Miami, FL	Keiser University-Miami	Miami	FL	NaN	NaN	NaN	1	NaN	NaN	NaN ...	NaN	
	Miami, FL	George T Baker Aviation Technical College	Miami	FL	0.0	0.0	0.0	0	NaN	NaN	0.0 ...	0.0046	

We can compare the speed of this compound index selection with boolean indexing. There is more than an order of magnitude difference:

```
>>> %%timeit
>>> crit1 = college['CITY'] == 'Miami'
>>> crit2 = college['STABBR'] == 'FL'
>>> college[crit1 & crit2]
2.43 ms ± 80.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> %%timeit college.loc['Miami, FL']
197 µs ± 8.69 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

See also

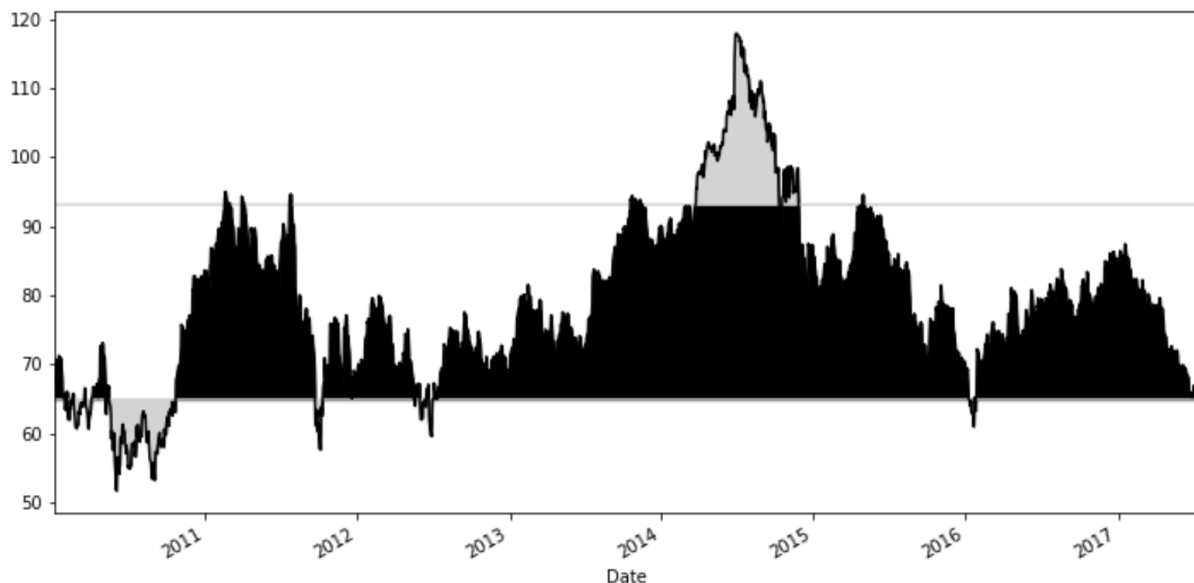
- The *Binary search algorithm* (<http://bit.ly/2wbMq20>)

Gaining perspective on stock prices

Instead of plotting red points (black points) over the closing prices to indicate the upper and lower tenth percentiles, we can use matplotlib's `fill_between` function. This function fills in all the areas

between two lines. It takes an optional where parameter that accepts a boolean Series, alerting it to exactly which locations to fill in:

```
>>> slb_close.plot(color='black', figsize=(12,6))
>>> plt.hlines(y=[lower_10, upper_10],
xmin=xmin, xmax=xmax,color='lightgray')
>>> plt.fill_between(x=criteria.index, y1=lower_10,
y2=slb_close.values, color='black')
>>> plt.fill_between(x=criteria.index,y1=lower_10,
y2=slb_close.values, where=slb_close < lower_10,
color='lightgray')
>>> plt.fill_between(x=criteria.index, y1=upper_10,
y2=slb_close.values, where=slb_close > upper_10,
color='lightgray')
```



Translating SQL WHERE clauses

For many operations, pandas has multiple ways to do the same thing. In this video, the criteria for salary uses two separate boolean expressions. Similarly to SQL, Series have a between method, with the salary criteria equivalently written as follows:

```
>>> criteria_sal = employee.BASE_SALARY.between(80000, 120000)
```

Another useful application of isin is to provide a sequence of values automatically generated by some other pandas statements. This would avoid any manual investigating to find the exact string names to store in a list. Conversely, let's try to exclude the rows from the top five most frequently occurring departments:

```
>>> top_5_depts = employee.DEPARTMENT.value_counts().index[:5]
>>> criteria = ~employee.DEPARTMENT.isin(top_5_depts)
>>> employee[criteria]
```

The SQL equivalent of this would be as follows:

```
SELECT
*
FROM
EMPLOYEE
WHERE
DEPARTMENT not in
(
SELECT
DEPARTMENT
FROM (
SELECT
DEPARTMENT,
COUNT(1) as CT
FROM
EMPLOYEE
GROUP BY
DEPARTMENT
ORDER BY
CT DESC
LIMIT 5
)
);
```

Notice the use of the pandas not operator, ~, which negates all boolean values of a Series.

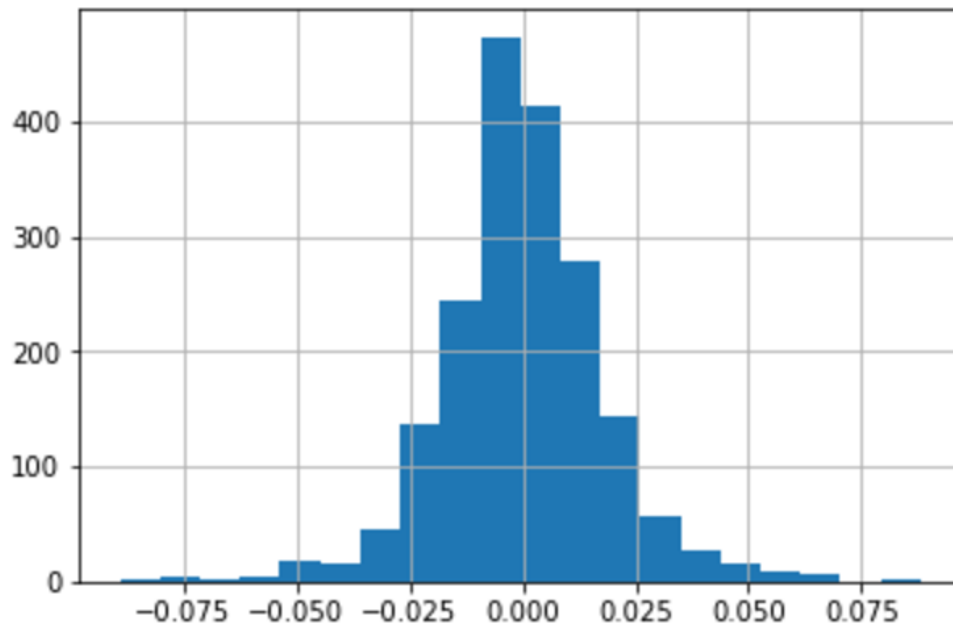
See also

- Pandas official documentation of the `isin` (<http://bit.ly/2v1GPfQ>) and `between` (<http://bit.ly/2wq9YPF>) Series methods
- Refer to the video, *Connecting to SQL databases* in Section 9, *Combining Pandas Objects*
- A basic introduction to SQL from W3 schools (<http://bit.ly/2hsq8Wp>)
- The SQL IN operator (<http://bit.ly/2v3H7Bq>)
- The SQL BETWEEN operator (<http://bit.ly/2vn5UTP>)

Determining the normality of stock market returns

To automate this process, we can write a function that accepts stock data in the and outputs the histogram of daily returns along with the percentages that fall within 1, 2, and 3 standard deviations from the mean. The following function does this and replaces the methods with their symbol counterparts:

```
>>> def test_return_normality(stock_data):
close = stock_data['Close']
daily_return = close.pct_change().dropna()
daily_return.hist(bins=20)
mean = daily_return.mean()
std = daily_return.std()
abs_z_score = abs(daily_return - mean) / std
pcts = [abs_z_score.lt(i).mean() for i in range(1,4)]
print('{:.3f} fall within 1 standard deviation. '
      '{:.3f} within 2 and {:.3f} within 3'.format(*pcts))
>>> slb = pd.read_csv('data/slb_stock.csv', index_col='Date',
parse_dates=['Date'])
>>> test_return_normality(slb)
0.742 fall within 1 standard deviation. 0.946 within 2 and 0.986 within 3
```



See also

- Pandas official documentation of the pct_change Series method (<http://bit.ly/2wcjmqT>)

Masking DataFrame rows

Let's compare the speed difference between masking and dropping missing rows and boolean indexing. Boolean indexing is about an order of magnitude faster in this case:

```
>>> %timeit movie.mask(criteria).dropna(how='all')
11.2 ms ± 144 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> %timeit movie[movie['title_year'] < 2010]
1.07 ms ± 34.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

See also

- Pandas official documentation on assert_frame_equal (<http://bit.ly/2u5H5YI>)
- Python official documentation of the assert statement (<http://bit.ly/2v1YKmY>)

Selecting with booleans, integer location, and labels

It is actually possible to pass arrays and lists of booleans to Series objects that are not the same length as the DataFrame you are doing the indexing on. Let's look at an example of this by selecting the first and third rows, and the first and fourth columns:

```
>>> movie.loc[[True, False, True], [True, False, False, True]]
```

	color	duration
movie_title		
Avatar	Color	178.0
Spectre	Color	148.0

Both of the boolean lists are not the same length as the axis they are indexing. The rest of the rows and columns not explicitly given a boolean value in the lists are dropped.

See also

- Refer to the video, *Selecting data with both integers and labels* from Section 4, *Selecting Subsets of Data*
- Refer to the video, *Selecting columns with methods* from Section 2, *Essential DataFrame Operations*

Section 6: Index Alignment

Examining the Index object

Indexes support the set operations, union, intersection, difference, and symmetric difference:

```
>>> c1 = columns[:4]
>>> c1
Index(['INSTNM', 'CITY', 'STABBR', 'HBCU'], dtype='object')
>>> c2 = columns[2:6]
>>> c2
Index(['STABBR', 'HBCU', 'MENONLY'], dtype='object')
>>> c1.union(c2) # or `c1 | c2`
Index(['CITY', 'HBCU', 'INSTNM', 'MENONLY', 'RELAFFIL', 'STABBR'],
      dtype='object')
>>> c1.symmetric_difference(c2) # or `c1 ^ c2`
Index(['CITY', 'INSTNM', 'MENONLY'], dtype='object')
```

Indexes share some of the same operations as Python sets. Indexes are similar to Python sets in another important way. They are (usually) implemented using hash tables, which make for extremely fast access when selecting rows or columns from a DataFrame. As they are implemented using hash tables, the values for the Index object need to be immutable such as a string, integer, or tuple just like the keys in a Python dictionary.

Indexes support duplicate values, and if there happens to be a duplicate in any Index, then a hash table can no longer be used for its implementation, and object access becomes much slower.

See also

- Pandas official documentation of Index (<http://bit.ly/2upfgtr>)

Exploding indexes

We can verify the number of values of salary_add by doing a little mathematics. As a Cartesian product takes place between all of the same index values, we can sum the square of their individual counts. Even missing values in the index produce Cartesian products with themselves:

```
>>> index_vc = salary1.index.value_counts(dropna=False)
>>> index_vc
Black or African American 700
White 665
Hispanic/Latino 480
Asian/Pacific Islander 107
NaN 35
American Indian or Alaskan Native 11
Others 2
Name: RACE, dtype: int64
>>> index_vc.pow(2).sum()
1175424
```

Filling values with unequal indexes

This video shows how to add Series with only a single index together. It is also entirely possible to add DataFrames together. Adding DataFrames together will align both the index and columns before computation and yield missing values for non-matching indexes. Let's start by selecting a few of the columns from the 2014 baseball dataset.

```
>>> df_14 = baseball_14[['G', 'AB', 'R', 'H']]
>>> df_14.head()
```

	G	AB	R	H
playerID				
altuvjo01	158	660	85	225
cartech02	145	507	68	115
castrja01	126	465	43	103
corpoca01	55	170	22	40
dominma01	157	564	51	121

Let's also select a few of the same and a few different columns from the 2015 baseball dataset:

```
>>> df_15 = baseball_15[['AB', 'R', 'H', 'HR']]
>>> df_15.head()
```

	AB	R	H	HR
playerID				
altuvjo01	638	86	200	15
cartech02	391	50	78	24
castrja01	337	38	71	11
congeha01	201	25	46	11
correca01	387	52	108	22

Adding the two DataFrames together create missing values wherever rows or column labels cannot align. Use the style attribute to access the highlight_null method to easily see where the missing values are:

```
>>> (df_14 + df_15).head(10).style.highlight_null('yellow')
```

	AB	G	H	HR	R
playerID					
altuvjo01	1298	nan	425	nan	171
cartech02	898	nan	193	nan	118
castrja01	802	nan	174	nan	81
congeha01	nan	nan	nan	nan	nan
corpoca01	nan	nan	nan	nan	nan
correca01	nan	nan	nan	nan	nan
dominma01	nan	nan	nan	nan	nan
fowlede01	nan	nan	nan	nan	nan
gattiev01	nan	nan	nan	nan	nan
gomezca01	nan	nan	nan	nan	nan

Only the rows with playerID appearing in both DataFrames will be non-missing. Similarly, the columns AB, H, and R are the only ones that appear in both DataFrames. Even if we use the add method with the fill_value parameter specified, we still have missing values. This is because some combinations of rows and columns never existed in our input data. For example, the intersection of playerID *congeha01* and column G. He only appeared in the 2015 dataset that did not have the G column. Therefore, no value was filled with it:

```
>>> df_14.add(df_15, fill_value=0).head(10) \
.style.highlight_null('yellow')
```

	AB	G	H	HR	R
playerID					
altuvjo01	1298	nan	425	nan	171
cartech02	898	nan	193	nan	118
castrja01	802	nan	174	nan	81
congeha01	nan	nan	nan	nan	nan
corpoca01	nan	nan	nan	nan	nan
correca01	nan	nan	nan	nan	nan
dominma01	nan	nan	nan	nan	nan
fowlede01	nan	nan	nan	nan	nan
gattiev01	nan	nan	nan	nan	nan
gomezca01	nan	nan	nan	nan	nan

Appending columns from different DataFrames

Not all indexes on the left-hand side of the equal sign need to have a match, but at most can have one. If there is nothing for the left DataFrame index to align to, the resulting value will be missing. Let's create an example where this happens. We will use only the first three rows of the `max_dept_sal` Series to create a new column:

```
>>> employee['MAX_SALARY2'] = max_dept_sal['BASE_SALARY'].head(3)
>>> employee.MAX_SALARY2.value_counts()
140416.0 29
100000.0 11
64251.0 5
Name: MAX_SALARY2, dtype: int64

>>> employee.MAX_SALARY2.isnull().mean()
.9775
```

The operation completed successfully but filled in salaries for only three of the departments. All the other departments that did not appear in the first three rows of the `max_dept_sal` Series resulted in a missing value.

See also

- *Selecting the largest from the smallest* from Section 3, *Beginning Data Analysis*

Highlighting the maximum value from each column

By default, the `highlight_max` method highlights the maximum value of each column. We can use the `axis` parameter to highlight the maximum value of each row instead. Here, we select just the race percentage columns of the college dataset and highlight the race with the highest percentage for each school:

```
>>> college = pd.read_csv('data/college.csv', index_col='INSTNM')
>>> college_ugds = college.filter(like='UGDS_').head()
>>> college_ugds.style.highlight_max(axis='columns')
```

Attempting to apply a style on a large DataFrame can cause Jupyter to crash, which is why the style was only applied to the head of the DataFrame.

See also

- Pandas official documentation on Dataframe *Styling* (<http://bit.ly/2hsZkVK>)

Replicating idxmax with method chaining

It is possible to complete this video in one long line of code chaining the indexing operator with an anonymous function. This little trick removes the need for step 10. We can time the difference between the direct `idxmax` method and our manual effort in this video:

```
>>> %timeit college_n.idxmax().values
1.12 ms ± 28.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
>>> %timeit college_n.eq(college_n.max()) \
```

```
.cumsum() \
.cumsum() \
.eq(1) \
.any(axis='columns') \
[lamba x: x].index
5.35 ms ± 55.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Our effort is, unfortunately, five times as slow as the built-in `idxmax` pandas method but regardless of its performance regression, many creative and practical solutions use the accumulation methods like `cumsum` with boolean Series to find streaks or specific patterns along an axis.

Section 7: Grouping for Aggregation, Filtration, and Transformation

Grouping and aggregating with multiple columns and functions

There are a few main flavors of syntax that you will encounter when performing an aggregation. The following four blocks of pseudocode summarize the main ways you can perform an aggregation with the `groupby` method:

1. Using `agg` with a dictionary is the most flexible and allows you to specify the aggregating function for each column:

```
>>> df.groupby(['grouping', 'columns']) \
.agg({'agg_cols1':['list', 'of', 'functions'],
'agg_cols2':['other', 'functions']})
```

2. Using `agg` with a list of aggregating functions applies each of the functions to each of the aggregating columns:

```
>>> df.groupby(['grouping', 'columns'])['aggregating', 'columns'] \
.agg([aggregating, functions])
```

3. Directly using a method following the aggregating columns instead of `agg`, applies just that method to each aggregating column. This way does not allow for multiple aggregating functions:

```
>>> df.groupby(['grouping', 'columns'])['aggregating', 'columns'] \
.aggregating_method()
```

4. If you do not specify the aggregating columns, then the aggregating method will be applied to all the non-grouping columns:

```
>>> df.groupby(['grouping', 'columns']).aggregating_method()
```

In the preceding four code blocks it is possible to substitute a string for any of the lists when grouping or aggregating by a single column.

Removing the MultiIndex after grouping

By default, at the end of a groupby operation, pandas puts all of the grouping columns in the index. The `as_index` parameter in the groupby method can be set to `False` to avoid this behavior. You can chain the `reset_index` method after grouping to get the same effect as done in step 3. Let's see an example of this by finding the average distance traveled per flight from each airline:

```
>>> flights.groupby(['AIRLINE'], as_index=False)['DIST'].agg('mean') \
.round(0)
```

	AIRLINE	DIST
0	AA	1114.0
1	AS	1066.0
2	B6	1772.0
3	DL	866.0
4	EV	460.0
5	F9	970.0
6	HA	2615.0
7	MQ	404.0
8	NK	1047.0
9	OO	511.0
10	UA	1231.0
11	US	1181.0
12	VX	1240.0
13	WN	810.0

Take a look at the order of the airlines in the previous result. By default, pandas sorts the grouping columns. The `sort` parameter exists within the groupby method and is defaulted to `True`. You may set it to `False` to keep the order of the grouping columns the same as how they are encountered in the dataset. You also get a small performance improvement by not sorting your data.

Customizing an aggregation function

It is possible to apply our customized function to multiple aggregating columns. We simply add more column names to the indexing operator. The `max_deviation` function only works with numeric columns:

```
>>> college.groupby('STABBR')['UGDS', 'SATVRMID', 'SATMTMID'] \
.agg(max_deviation).round(1).head()
```

	UGDS	SATVRMID	SATMTMID
STABBR			
AK	2.6	NaN	NaN
AL	5.8	1.6	1.8
AR	6.3	2.2	2.3
AS	NaN	NaN	NaN
AZ	9.9	1.9	1.4

You can also use your customized aggregation function along with the prebuilt functions. The following does this and groups by state and religious affiliation:

```
>>> college.groupby(['STABBR', 'RELAFFIL']) \
['UGDS', 'SATVRMID', 'SATMTMID'] \
.agg([max_deviation, 'mean', 'std']).round(1).head()
```

		UGDS			SATVRMID			SATMTMID		
		max_deviation	mean	std	max_deviation	mean	std	max_deviation	mean	std
STABBR	RELAFFIL									
AK	0	2.1	3508.9	4539.5	NaN	NaN	NaN	NaN	NaN	NaN
	1	1.1	123.3	132.9	NaN	555.0	NaN	NaN	503.0	NaN
AL	0	5.2	3248.8	5102.4	1.6	514.9	56.5	1.7	515.8	56.7
	1	2.4	979.7	870.8	1.5	498.0	53.0	1.4	485.6	61.4
AR	0	5.8	1793.7	3401.6	1.9	481.1	37.9	2.0	503.6	39.0

Notice that pandas uses the name of the function as the name for the returned column. You can change the column name directly with the rename method or you can modify the special function attribute `__name__`:

```
>>> max_deviation.__name__
'max_deviation'
>>> max_deviation.__name__ = 'Max Deviation'
>>> college.groupby(['STABBR', 'RELAFFIL']) \
['UGDS', 'SATVRMID', 'SATMTMID'] \
.agg([max_deviation, 'mean', 'std']).round(1).head()
```

		UGDS			SATVRMID			SATMTMID		
		Max Deviation	mean	std	Max Deviation	mean	std	Max Deviation	mean	std
STABBR	RELAFFIL									
AK	0	2.1	3508.9	4539.5	NaN	NaN	NaN	NaN	NaN	NaN
	1	1.1	123.3	132.9	NaN	555.0	NaN	NaN	503.0	NaN
AL	0	5.2	3248.8	5102.4	1.6	514.9	56.5	1.7	515.8	56.7
	1	2.4	979.7	870.8	1.5	498.0	53.0	1.4	485.6	61.4
AR	0	5.8	1793.7	3401.6	1.9	481.1	37.9	2.0	503.6	39.0

Customizing aggregating functions with *args and **kwargs

Unfortunately, pandas does not have a direct way to use these additional arguments when using multiple aggregation functions together. For example, if you wish to aggregate using the `pct_between` and `mean` functions, you will get the following exception:

```
>>> college.groupby(['STABBR', 'RELAFFIL'])['UGDS'] \
.agg(['mean', pct_between], low=100, high=1000)
TypeError: pct_between() missing 2 required positional arguments: 'low' and 'high'
```

Pandas is incapable of understanding that the extra arguments need to be passed to `pct_between`. In order to use our custom function with other built-in functions and even other custom functions, we can define a special type of nested function called a **closure**. We can use a generic closure to build all of our customized functions:

```
>>> def make_agg_func(func, name, *args, **kwargs):
def wrapper(x):
return func(x, *args, **kwargs)
wrapper.__name__ = name
return wrapper
>>> my_agg1 = make_agg_func(pct_between, 'pct_1_3k', low=1000, high=3000)
>>> my_agg2 = make_agg_func(pct_between, 'pct_10_30k', 10000, 30000)
>>> college.groupby(['STABBR', 'RELAFFIL'])['UGDS'] \
.agg(['mean', my_agg1, my_agg2]).head()
```

		mean	pct_1_3k	pct_10_30k
STABBR	RELAFFIL			
AK	0	3508.857143	0.142857	0.142857
	1	123.333333	0.000000	0.000000
AL	0	3248.774648	0.236111	0.083333
	1	979.722222	0.333333	0.000000
AR	0	1793.691176	0.279412	0.014706

The `make_agg_func` function acts as a factory to create customized aggregation functions. It accepts the customized aggregation function that you already built (`pct_between` in this case), a name argument, and an arbitrary number of extra arguments. It returns a function with the extra arguments already set. For instance, `my_agg1` is a specific customized aggregating function that finds the percentage of schools with an undergraduate population between one and three thousand. The extra arguments (`*args` and `**kwargs`) specify an exact set of parameters for your customized function (`pct_between` in this case). The name parameter is very important and must be unique each time `make_agg_func` is called. It will eventually be used to rename the aggregated column.

A closure is a function that contains a function inside of it (a nested function) and returns this nested function. This nested function must refer to variables in the scope of the outer function in order to be a closure. In this example, `make_agg_func` is the outer function and returns the nested function `wrapper`, which accesses the variables `func`, `args`, and `kwargs` from the outer function.

See also

- *Arbitrary Argument Lists* from the official Python documentation (<http://bit.ly/2vumbTE>)
- A tutorial on *Python Closures* (<http://bit.ly/2xFdYga>)

Examining the groupby object

There are several useful methods that were not explored from the list in step 2. Take for instance the `nth` method, which, when given a list of integers, selects those specific rows from each group. For example, the following operation selects the first and last rows from each group:

```
>>> grouped.nth([1, -1]).head(8)
```

		CITY	CURROPER	DISTANCEONLY	GRAD_DEBT_MDN_SUPP	HBCU	INSTNM	MD_EARN_WNE_P10	MENONLY	PCTFLOAN	F
STABBR	RELAFFIL										
AK	0	Fairbanks	1	0.0	19355	0.0	University of Alaska Fairbanks	36200	0.0	0.2550	
	0	Barrow	1	0.0	PrivacySuppressed	0.0	Ilisagvik College	24900	0.0	0.0000	
	1	Anchorage	1	0.0	23250	0.0	Alaska Pacific University	47000	0.0	0.5297	
	1	Soldotna	1	0.0	PrivacySuppressed	0.0	Alaska Christian College	NaN	0.0	0.6792	
AL	0	Birmingham	1	0.0	21941.5	0.0	University of Alabama at Birmingham	39700	0.0	0.5214	
	0	Dothan	1	0.0	PrivacySuppressed	0.0	Alabama College of Osteopathic Medicine	NaN	0.0	NaN	
	1	Birmingham	1	0.0	27000	0.0	Birmingham Southern College	44200	0.0	0.4809	
	1	Huntsville	1	NaN	36173.5	NaN	Strayer University-Huntsville Campus	49200	NaN	NaN	

See also

- Official documentation of the display function from IPython (<http://bit.ly/2iAlogC>)

Filtering for states with a minority majority

Our function, `check_minority`, is flexible and accepts a parameter to lower or raise the percentage of minority threshold. Let's check the shape and number of unique states for a couple of other thresholds:

```
>>> college_filtered_20 = grouped.filter(check_minority, threshold=.2)
>>> college_filtered_20.shape
(7461, 26)
>>> college_filtered_20['STABBR'].nunique()
57
>>> college_filtered_70 = grouped.filter(check_minority, threshold=.7)
>>> college_filtered_70.shape
(957, 26)
>>> college_filtered_70['STABBR'].nunique()
10
```

See also

- Pandas official documentation on *Filtration* (<http://bit.ly/2xGUoA7>)

Transforming through a weight loss bet

Take a look at the DataFrame output from step 7. Did you notice that the months are in alphabetical and not chronological order? Pandas unfortunately, in this case at least, orders the months for us alphabetically. We can solve this issue by changing the data type of Month to a categorical variable. Categorical variables map all the values of each column to an integer. We can choose this mapping

to be the normal chronological order for the months. Pandas uses this underlying integer mapping during the pivot method to order the months chronologically:

```
>>> week4a = week4.copy()
>>> month_chron = week4a['Month'].unique() # or use drop_duplicates
>>> month_chron
array(['Jan', 'Feb', 'Mar', 'Apr'], dtype=object)
>>> week4a['Month'] = pd.Categorical(week4a['Month'],
categories=month_chron,
ordered=True)
>>> week4a.pivot(index='Month', columns='Name',
values='Perc Weight Loss')
```

Name	Amy	Bob
Month		
Jan	-0.036	-0.027
Feb	-0.089	-0.053
Mar	-0.017	-0.026
Apr	-0.053	-0.042

To convert the Month column, use the Categorical constructor. Pass it the original column as a Series and a unique sequence of all the categories in the desired order to the categories parameter. As the Month column is already in chronological order, we can simply use the unique method, which preserves order to get the array that we desire. In general, to sort columns of object data type by something other than alphabetical, convert them to categorical.

See also

- Pandas official documentation on *groupby Transformation* (<http://bit.ly/2vBkpA7>)
- NumPy official documentation on the where function (<http://bit.ly/2weT21l>)

Calculating weighted mean SAT scores per state with apply

In this video, we returned a single row as a Series for each group. It's possible to return any number of rows and columns for each group by returning a DataFrame. In addition to finding just the arithmetic and weighted means, let's also find the geometric and harmonic means of both SAT columns and return the results as a DataFrame with rows as the name of the type of mean and columns as the SAT type. To ease the burden on us, we use the NumPy function average to compute the weighted average and the SciPy functions gmean and hmean for geometric and harmonic means:

```
>>> from scipy.stats import gmean, hmean
>>> def calculate_means(df):
df_means = pd.DataFrame(index=['Arithmetic', 'Weighted',
'Geometric', 'Harmonic'])
cols = ['SATMTMID', 'SATVRMID']
for col in cols:
arithmetic = df[col].mean()
weighted = np.average(df[col], weights=df['UGDS'])
geometric = gmean(df[col])
harmonic = hmean(df[col])
df_means[col] = [arithmetic, weighted,
geometric, harmonic]
```

```
df_means['count'] = len(df)
return df_means.astype(int)
>>> college2.groupby('STABBR').apply(calculate_means).head(12)
```

		SATMTMID	SATVRMID	count
STABBR				
AK	Arithmetic	503	555	1
	Weighted	503	555	1
	Geometric	503	555	1
	Harmonic	503	555	1
AL	Arithmetic	504	508	21
	Weighted	536	533	21
	Geometric	500	505	21
	Harmonic	497	502	21
AR	Arithmetic	515	491	16
	Weighted	529	504	16
	Geometric	514	489	16
	Harmonic	513	487	16

See also

- Pandas official documentation of the apply groupby method (<http://bit.ly/2wmG9ki>)
- Python official documentation of the OrderedDict class (<http://bit.ly/2xwtUCa>)
- SciPy official documentation of its stats module (<http://bit.ly/2wHtQ4L>)

Grouping by continuous variables

We can find more results when grouping by the cuts variable. For instance, we can find the 25th, 50th, and 75th percentile airtime for each distance grouping. As airtime is in minutes, we can divide by 60 to get hours:

```
>>> flights.groupby(cuts)['AIR_TIME'].quantile(q=[.25, .5, .75]) \
.div(60).round(2)
DIST
(-inf, 200.0] 0.25 0.43
              0.50 0.50
              0.75 0.57
(200.0, 500.0] 0.25 0.77
              0.50 0.92
              0.75 1.05
(500.0, 1000.0] 0.25 1.43
               0.50 1.65
               0.75 1.92
(1000.0, 2000.0] 0.25 2.50
                 0.50 2.93
                 0.75 3.40
(2000.0, inf] 0.25 4.30
              0.50 4.70
              0.75 5.03
Name: AIR_TIME, dtype: float64
```

We can use this information to create informative string labels when using the cut function. These labels replace the interval notation. We can also chain the unstack method which transposes the inner index level to column names:

```
>>> labels=['Under an Hour', '1 Hour', '1-2 Hours',
'2-4 Hours', '4+ Hours']
>>> cuts2 = pd.cut(flights['DIST'], bins=bins, labels=labels)
>>> flights.groupby(cuts2)['AIRLINE'].value_counts(normalize=True) \
.round(3) \
.unstack() \
.style.highlight_max(axis=1)
```

	AIRLINE	AA	AS	B6	DL	EV	F9	HA	MQ	NK	OO	UA	US	VX	WN
DIST															
Under an Hour		0.052	nan	nan	0.086	0.289	nan	nan	0.211	nan	0.326	0.027	nan	nan	0.009
1 Hour		0.071	0.001	0.007	0.189	0.156	0.005	nan	0.1	0.012	0.159	0.062	0.016	0.028	0.194
1-2 Hours		0.144	0.023	0.003	0.206	0.101	0.038	nan	0.051	0.03	0.106	0.131	0.025	0.004	0.138
2-4 Hours		0.264	0.016	0.003	0.165	0.016	0.031	nan	0.003	0.045	0.046	0.199	0.04	0.012	0.16
4+ Hours		0.212	0.012	0.08	0.171	nan	0.004	0.028	nan	0.019	nan	0.289	0.065	0.074	0.046

See also

- Pandas official documentation on the cut function (<http://bit.ly/2whcUkJ>)
- Refer to Section 8, *Restructuring Data into Tidy Form*, for many more videos with unstack

Counting the total number of flights between cities

You might be wondering why we can't use the simpler sort_values Series method. This method does not sort independently and instead, preserves the row or column as a single record as one would expect while doing a data analysis. Step 3 is a very expensive operation and takes several seconds to complete. There are only about 60,000 rows so this solution would not scale well to larger data. Calling the Step 3 is a very expensive operation and takes several seconds to complete. There are only about 60,000 rows so this solution would not scale well to larger data. Calling the apply method with axis=1 is one of the least performant operations in all of pandas. Internally, pandas loops over each row and does not provide any speed boosts from NumPy. If possible, avoid using apply with axis=1.

We can get a massive speed increase with the NumPy sort function. Let's go ahead and use this function and analyze its output. By default, it sorts each row independently:

```
>>> data_sorted = np.sort(flights[['ORG_AIR', 'DEST_AIR']])
>>> data_sorted[:10]
array([[ 'LAX', 'SLC'],
[ 'DEN', 'IAD'],
[ 'DFW', 'VPS'],
[ 'DCA', 'DFW'],
[ 'LAX', 'MCI'],
[ 'IAH', 'SAN'],
[ 'DFW', 'MSY'],
[ 'PHX', 'SFO'],
[ 'ORD', 'STL'],
[ 'IAH', 'SJC']], dtype=object)
```

A two-dimensional NumPy array is returned. NumPy does not easily do grouping operations so let's use the DataFrame constructor to create a new DataFrame and check whether it equals the flights_sorted DataFrame from step 3:

```
>>> flights_sort2 = pd.DataFrame(data_sorted, columns=['AIR1', 'AIR2'])
>>> fs_orig = flights_sort.rename(columns={'ORG_AIR':'AIR1',
'DEST_AIR':'AIR2'})
>>> flights_sort2.equals(fs_orig)
True
```

As the DataFrames are the same, you can replace step 3 with the previous faster sorting routine. Let's time the difference between each of the different sorting methods:

```
>>> %%timeit
>>> flights_sort = flights[['ORG_AIR', 'DEST_AIR']] \
.apply(sorted, axis=1)
7.41 s ± 189 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
>>> %%timeit
>>> data_sorted = np.sort(flights[['ORG_AIR', 'DEST_AIR']])
>>> flights_sort2 = pd.DataFrame(data_sorted,
columns=['AIR1', 'AIR2'])
10.6 ms ± 453 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The NumPy solution is an astounding 700 times faster than using apply with pandas.

See also

- NumPy official documentation on the sort function (<http://bit.ly/2vtRt0M>)

Finding the longest streak of on-time flights

Now that we have found the longest streaks of on-time arrivals, we can easily find the opposite--the longest streak of delayed arrivals. The following function returns two rows for each group passed to it. The first row is the start of the streak, and the last row is the end of the streak. Each row contains the month and day that the streak started/ended, along with the total streak length:

```
>>> def max_delay_streak(df):
df = df.reset_index(drop=True)
s = 1 - df['ON_TIME']
s1 = s.cumsum()
streak = s.mul(s1).diff().where(lambda x: x < 0) \
.fillna().add(s1, fill_value=0)
last_idx = streak.idxmax()
first_idx = last_idx - streak.max() + 1
df_return = df.loc[[first_idx, last_idx], ['MONTH', 'DAY']]
df_return['streak'] = streak.max()
df_return.index = ['first', 'last']
df_return.index.name = 'type'
return df_return
>>> flights.sort_values(['MONTH', 'DAY', 'SCHED_DEP']) \
.groupby(['AIRLINE', 'ORG_AIR']) \
.apply(max_delay_streak) \
.sort_values('streak', ascending=False).head(10)
```


AIRLINE	ORG_AIR	streak_row	MONTH DAY		streak
AA	DFW	first	2.0	26.0	38.0
		last	3.0	1.0	38.0
MQ	ORD	first	1.0	6.0	28.0
		last	1.0	12.0	28.0
	DFW	first	2.0	21.0	25.0
		last	2.0	26.0	25.0
NK	ORD	first	6.0	7.0	15.0
		last	6.0	18.0	15.0
DL	ATL	first	12.0	23.0	14.0
		last	12.0	24.0	14.0

As we are using the apply groupby method, a DataFrame of each group is passed to the max_delay_streak function. Inside this function, the index of the DataFrame is dropped and replaced by a RangeIndex in order for us to easily find the first and last row of the streak. The ON_TIME column is inverted and then the same logic is used to find streaks of delayed flights. The index of the first and last rows of the streak are stored as variables. These indexes are then used to select the month and day when the streaks ended. We use a DataFrame to return our results. We label and name the index to make the final result clearer.

Our final results show the longest delayed streaks accompanied by the first and last date. Let's investigate to see if we can find out why these delays happened. Inclement weather is a common reason for delayed or canceled flights. Looking at the first row, American Airlines (AA) started a streak of 38 delayed flights in a row from the Dallas Fort-Worth (DFW) airport beginning February 26 until March 1 of 2015. Looking at historical weather data from February 27, 2015, two inches of snow fell, which was a record for that day (<http://bit.ly/2iLGsCg>). This was a major weather event for DFW and caused massive problems for the entire city (<http://bit.ly/2wmsHPi>). Notice that DFW makes another appearance as the third longest streak but this time a few days earlier and for a different airline.

See also

- Refer to the video, *Working with operators on a Series* from Section 1, *Pandas Foundations*
- Pandas official documentation of ffill (<http://bit.ly/2gn5zGU>)

Section 8: Restructuring Data into a Tidy Form

Tidying variable values as column names with stack

One of the keys to using stack is to place all of the columns that you do not wish to transform in the index. The dataset in this video was initially read with the states in the index. Let's take a look at what would have happened if we did not read the states into the index:

```
>>> state_fruit2 = pd.read_csv('data/state_fruit2.csv')
>>> state_fruit2
```

	State	Apple	Orange	Banana
0	Texas	12	10	40
1	Arizona	9	7	12
2	Florida	0	14	190

As the state names are not in the index, using stack on this DataFrame reshapes all values into one long Series of values:

```
>>> state_fruit2.stack()
0 State Texas
  Apple 12
  Orange 10
  Banana 40
1 State Arizona
  Apple 9
  Orange 7
  Banana 12
2 State Florida
  Apple 0
  Orange 14
  Banana 190
dtype: object
```

This command reshapes all the columns, this time including the states, and is not at all what we need. In order to reshape this data correctly, you will need to put all the non-reshaped columns into the index first with the set_index method, and then use stack. The following code gives a similar result to step 1:

```
>>> state_fruit2.set_index('State').stack()
```

See also

- Pandas official documentation on *Reshaping and Pivot Tables* (<http://bit.ly/2xbnNms>)
- Pandas official documentation on the stack method (<http://bit.ly/2vWZhH1>)

Tidying variable values as column names with melt

Like most large Python libraries, pandas has many different ways to accomplish the same task--the differences usually being readability and performance. Pandas contains a DataFrame method named

melt that works similarly to the stack method described in the previous video but gives a bit more flexibility.

Before pandas version 0.20, melt was only provided as a function that had to be accessed with `pd.melt`. Pandas is still an evolving library and you need to expect changes with each new version. Pandas has been making a push to move all functions that only operate on DataFrames to methods, such as they did with melt. This is the preferred way to use melt and the way this video uses it. Check the *What's New* part of the pandas documentation to stay up to date with all the changes (<http://bit.ly/2xzXlhG>).

See also

- Pandas official documentation on the melt method (<http://bit.ly/2vcuZNJ>)
- Pandas developers discussion of melt and other similar functions being converted to methods (<http://bit.ly/2iqIQhl>)

Stacking multiple groups of variables simultaneously

To become a powerful user of the str methods, you will need to be familiar with regular expressions, which are a sequence of characters that match a particular pattern within some text. They consist of **metacharacters**, which have a special meaning, and **literal** characters. To make yourself useful with regular expressions check this short tutorial from *Regular-Expressions.info* (<http://bit.ly/2wiWPbz>).

The function `wide_to_long` works when all groupings of variables have the same numeric ending like they did in this video. When your variables do not have the same ending or don't end in a digit, you can still use `wide_to_long` to do simultaneous column stacking. For instance, let's take a look at the following dataset:

```
>>> df = pd.read_csv('data/stackme.csv')
>>> df
```

	State	Country	a1	b2	Test	d	e
0	TX	US	0.45	0.3	Test1	2	6
1	MA	US	0.03	1.2	Test2	9	7
2	ON	CAN	0.70	4.2	Test3	4	2

Let's say we wanted columns a1 and b1 stacked together, as well as columns d and e. Additionally, we wanted to use a1 and b1 as labels for the rows. To accomplish this task, we would need to rename the columns so that they ended in the label we desired:

```
>>> df2 = df.rename(columns = {'a1':'group1_a1', 'b2':'group1_b2',
'd':'group2_a1', 'e':'group2_b2'})
>>> df2
```

	State	Country	group1_a1	group1_b2	Test	group2_a1	group2_b2
0	TX	US	0.45	0.3	Test1	2	6
1	MA	US	0.03	1.2	Test2	9	7
2	ON	CAN	0.70	4.2	Test3	4	2

We would then need to modify the suffix parameter, which normally defaults to a regular expression that selects digits. Here, we simply tell it to find any number of characters:

```
>>> pd.wide_to_long(df2,
stubnames=['group1', 'group2'],
i=['State', 'Country', 'Test'],
j='Label',
suffix='.+',
sep='_')
```

			group1	group2
State	Country	Test	Label	
TX	US	Test1	a1	0.45
			b2	0.30
MA	US	Test2	a1	0.03
			b2	1.20
ON	CAN	Test3	a1	0.70
			b2	4.20

See also

- Pandas official documentation for `wide_to_long` (<http://bit.ly/2xb8NVP>)

Inverting stacked data

To help further understand `stack/unstack`, let's use them to **transpose** the college DataFrame. In this context, we are using the precise mathematical definition of the transposing of a matrix, where the new rows are the old columns of the original data matrix.

If you take a look at the output from step 2, you'll notice that there are two index levels. By default, the `unstack` method uses the innermost index level as the new column values. Index levels are numbered beginning from zero from the outside. Pandas defaults the `level` parameter of the `unstack` method to -1, which refers to the innermost index. We can instead unstack the outermost column using `level=0`:

```
>>> college.stack().unstack(0)
```

INSTNM	Alabama A & M University	University of Alabama at Birmingham	Amridge University	University of Alabama in Huntsville	Alabama State University	The University of Alabama	Central Alabama Community College	Athens State University	Auburn University at Montgomery	Auburn University	...
UGDS_WHITE	0.0333	0.5922	0.2990	0.6988	0.0158	0.7825	0.7255	0.7823	0.5328	0.8507	...
UGDS_BLACK	0.9353	0.2600	0.4192	0.1255	0.9208	0.1119	0.2613	0.1200	0.3376	0.0704	...
UGDS_HISP	0.0055	0.0283	0.0069	0.0382	0.0121	0.0348	0.0044	0.0191	0.0074	0.0248	...
UGDS_ASIAN	0.0019	0.0518	0.0034	0.0376	0.0019	0.0106	0.0025	0.0053	0.0221	0.0227	...
UGDS_AIAN	0.0024	0.0022	0.0000	0.0143	0.0010	0.0038	0.0044	0.0157	0.0044	0.0074	...
UGDS_NHPI	0.0019	0.0007	0.0000	0.0002	0.0006	0.0009	0.0000	0.0010	0.0016	0.0000	...
UGDS_2MOR	0.0000	0.0368	0.0000	0.0172	0.0098	0.0261	0.0000	0.0174	0.0297	0.0000	...
UGDS_NRA	0.0059	0.0179	0.0000	0.0332	0.0243	0.0268	0.0000	0.0057	0.0397	0.0100	...
UGDS_UNKN	0.0138	0.0100	0.2715	0.0350	0.0137	0.0026	0.0019	0.0334	0.0246	0.0140	...

There is actually a very simple way to transpose a DataFrame that don't require stack or unstack by using the transpose method or the T attribute like this:

```
>>> college.T
>>> college.transpose()
```

See also

- Refer to the video, *Selecting DataFrame rows and columns simultaneously* from Section 4, *Selecting Subsets of Data*
- Pandas official documentation of the unstack (<http://bit.ly/2xlyFvr>) and pivot (<http://bit.ly/2f3qAWP>) Methods

Unstacking after a groupby aggregation

If there are multiple grouping and aggregating columns, then the immediate result will be a DataFrame and not a Series. For instance, let's calculate more aggregations than just the mean, as was done in step 2:

```
>>> agg2 = employee.groupby(['RACE', 'GENDER'])['BASE_SALARY'] \
.agg(['mean', 'max', 'min']).astype(int)
>>> agg2
```

		mean	max	min
RACE	GENDER			
American Indian or Alaskan Native	Female	60238	98536	26125
	Male	60305	81239	26125
Asian/Pacific Islander	Female	63226	130416	26125
	Male	61033	163228	27914
Black or African American	Female	48915	150416	24960
	Male	51082	275000	26125
Hispanic/Latino	Female	46503	126115	26125
	Male	54782	165216	26104
Others	Female	63785	63785	63785
	Male	38771	38771	38771
White	Female	66793	178331	27955
	Male	63940	210588	26125

Unstacking the **Gender** column will result in MultiIndex columns. From here, you can keep swapping row and column levels with both the unstack and stack methods until you achieve the structure of data you desire:

```
>>> agg2.unstack('GENDER')
```

GENDER	mean		max		min	
	Female	Male	Female	Male	Female	Male
RACE						
American Indian or Alaskan Native	60238	60305	98536	81239	26125	26125
Asian/Pacific Islander	63226	61033	130416	163228	26125	27914
Black or African American	48915	51082	150416	275000	24960	26125
Hispanic/Latino	46503	54782	126115	165216	26125	26104
Others	63785	38771	63785	38771	63785	38771
White	66793	63940	178331	210588	27955	26125

See also

- Refer to the video *Grouping and aggregating with multiple columns* and functions from Section 7, *Grouping for Aggregation, Filtration, and Transformation*

Replicating pivot_table with a groupby aggregation

It is possible to replicate much more complex pivot tables with groupby aggregations. For instance, take the following result from pivot_table:

```
>>> flights.pivot_table(index=['AIRLINE', 'MONTH'],
columns=['ORG_AIR', 'CANCELLED'],
values=['DEP_DELAY', 'DIST'],
aggfunc=[np.sum, np.mean],
fill_value=0)
```

mean										... sum											
DEP_DELAY										... DIST											
ORG_AIR	ATL		DEN		DFW		IAH		LAS		LAX		MSP		ORD		PHX		SFO		
CANCELLED	0		1	0		1	0		1	0		1	...	0		1	0		1	0	
AIRLINE	MONTH																				
AA	1	-3.250000	0	7.062500	0	11.977591	-3.0	9.750000	0	32.375000	0	...	135921	2475	7281	0	129334	0	21018	0	33483
	2	-3.000000	0	5.461538	0	8.756579	0.0	1.000000	0	-3.055556	0	...	113483	5454	5040	0	120572	5398	17049	868	32110
	3	-0.166667	0	7.666667	0	15.383784	0.0	10.900000	0	12.074074	0	...	131836	1744	14471	0	127072	802	25770	0	43580
	4	0.071429	0	20.266667	0	10.501493	0.0	6.933333	0	27.241379	0	...	170285	0	4541	0	152154	4718	17727	0	51054
	5	5.777778	0	23.466667	0	16.798780	0.0	3.055556	0	2.818182	0	...	167484	0	6298	0	110864	1999	11164	0	40233

To replicate this with a groupby aggregation, simply follow the same pattern from the video and place all the columns from the index and columns parameters into the groupby method and then unstack the columns:

```
>>> flights.groupby(['AIRLINE', 'MONTH', 'ORG_AIR', 'CANCELLED']) \
['DEP_DELAY', 'DIST'] \
.agg(['mean', 'sum']) \
```

```
.unstack(['ORG_AIR', 'CANCELLED'], fill_value=0) \
.swaplevel(0, 1, axis='columns')
```

There are a few differences. The `pivot_table` method does not accept aggregation functions as strings when passed as a list like the `agg` groupby method. Instead, you must use NumPy functions. The order of the column levels also differs, with `pivot_table` putting the aggregation functions at a level preceding the columns in the `values` parameter. This is equalized with the `swaplevel` method that, in this instance, switches the order of the top two levels.

As of the time of developing this course, there is a bug when unstacking more than one column. The `fill_value` parameter is ignored (<http://bit.ly/2jCPnWZ>). To work around this bug, chain `.fillna(0)` to the end of the code.

Renaming axis levels for easy reshaping

If you wish to dispose of the level values altogether, you may set them to `None`. A case for this can be made when there is a need to reduce clutter in the visual output of a DataFrame or when it is obvious what the column levels represent and no further processing will take place:

```
>>> cg.rename_axis([None, None], axis='index') \
.rename_axis([None, None], axis='columns')
```

		UGDS			SATMTMID		
		count	min	max	count	min	max
AK	0	7	109.0	12865.0	0	NaN	NaN
	1	3	27.0	275.0	1	503.0	503.0
AL	0	71	12.0	29851.0	13	420.0	590.0
	1	18	13.0	3033.0	8	400.0	560.0
AR	0	68	18.0	21405.0	9	427.0	565.0
	1	14	20.0	4485.0	7	495.0	600.0

Tidying when multiple variables are stored as column names

Another way to complete this video, beginning after step 2, is by directly assigning new columns from the `sex_age` column without using the `split` method. The `assign` method may be used to add these new columns dynamically:

```
>>> age_group = wl_melt.sex_age.str.extract('(\d{2})[-+](?:\d{2})?')
expand=False)
```

```
>>> sex = wl_melt.sex_age.str[0]
>>> new_cols = {'Sex':sex,
'Age Group': age_group}
>>> wl_tidy2 = wl_melt.assign(**new_cols) \
.drop('sex_age',axis='columns')
>>> wl_tidy2.sort_index(axis=1).equals(wl_tidy.sort_index(axis=1))
True
```

The Sex column is found in the exact same manner as done in step 5. Because we are not using split, the Age Group column must be extracted in a different manner. The extract method uses a complex regular expression to extract very specific portions of the string. To use extract correctly, your pattern must contain capture groups. A capture group is formed by enclosing parentheses around a portion of the pattern. In this example, the entire expression is one large capture group. It begins with `\d{2}`, which searches for exactly two digits, followed by either a literal plus or minus, optionally followed by two more digits. Although the last part of the expression, `(?:\d{2})?`, is surrounded by parentheses, the `?:` denotes that it is not actually a capture group. It is technically a non-capturing group used to express two digits together as optional. The sex_age column is no longer needed and is dropped. Finally, the two tidy DataFrames are compared against one another and are found to be equivalent.

See also

- Refer to the site *Regular-Expressions.info* for more on non-capturing groups (<http://bit.ly/2f60KSd>)

Tidying when multiple variables are stored as column values

It is actually possible to use the `pivot_table` method, which has no restrictions on how many non-pivoted columns are allowed. The `pivot_table` method differs from `pivot` by performing an aggregation for all the values that correspond to the intersection between the columns in the index and columns parameters. Because it is possible that there are multiple values in this intersection, `pivot_table` requires the user to pass it an aggregating function, in order to output a single value. We use the first aggregating function, which takes the first of the values of the group. In this particular example, there is exactly one value for each intersection, so there is nothing to be aggregated. The default aggregation function is the mean, which will produce an error here, since some of the values are strings:

```
>>> inspections.pivot_table(index=['Name', 'Date'],
columns='Info',
values='Value',
aggfunc='first') \
.reset_index() \
.rename_axis(None, axis='columns')
```

See also

- Pandas official documentation of the `droplevel` (<http://bit.ly/2yo5BXf>) and `squeeze` (<http://bit.ly/2yo5TgN>) methods

Tidying when two or more values are stored in the same cell

The split method worked exceptionally well in this example with a simple regular expression. For other examples, some columns might require you to create splits on several different patterns. To search for multiple regular expressions, use the pipe character |. For instance, if we wanted to split only the degree symbol and comma, each followed by a space, we would do the following:

```
>>> cities.Geolocation.str.split(pat='° |, ', expand=True)
```

This returns the same DataFrame from step 2. Any number of additional split patterns may be appended to the preceding string pattern with the pipe character. The extract method is another excellent method which allows you to extract specific groups within each cell. These capture groups must be enclosed in parentheses. Anything that matches outside the parentheses is not present in the result. The following line produces the same output as step 2:

```
>>> cities.Geolocation.str.extract('([0-9.]+). (N|S), ([0-9.]+). (E|W)', expand=True)
```

This regular expression has four capture groups. The first and third groups search for at least one or more consecutive digits with decimals. The second and fourth groups search for a single character (the direction). The first and third capture groups are separated by any character followed by a space. The second capture group is separated by a comma and then a space.

Tidying when variables are stored in column names and values

Whenever a solution involves melt, pivot_table, or pivot, you can be sure that there is an alternative method using stack and unstack. The trick is first to move the columns that are not currently being pivoted into the index:

```
>>> sensors.set_index(['Group', 'Property']) \
.stack() \
.unstack('Property') \
.rename_axis(['Group', 'Year'], axis='index') \
.rename_axis(None, axis='columns') \
.reset_index()
```

Tidying when multiple observational units are stored in the same table

It is possible to recreate the original movie table by joining all the tables back together. First, join the associative tables to the actor/director tables. Then pivot the num column, and add the column prefixes back:

```
>>> actors = actor_associative.merge(actor_unique, on='actor_id') \
.drop('actor_id', 1) \
.pivot_table(index='id',
columns='num',
aggfunc='first')
```

```
>>> actors.columns = actors.columns.get_level_values(0) + '_' + \
actors.columns.get_level_values(1).astype(str)
>>> directors = director_associative.merge(director_unique,
on='director_id') \
.drop('director_id', 1) \
.pivot_table(index='id',
columns='num',
aggfunc='first')
>>> directors.columns = directors.columns.get_level_values(0) + '_' + \
directors.columns.get_level_values(1) \
.astype(str)
```

	actor_1	actor_2	actor_3	actor_fb_likes_1	actor_fb_likes_2	actor_fb_likes_3
id						
0	CCH Pounder	Joel David Moore	Wes Studi	1000.0	936.0	855.0
1	Johnny Depp	Orlando Bloom	Jack Davenport	40000.0	5000.0	1000.0
2	Christoph Waltz	Rory Kinnear	Stephanie Sigman	11000.0	393.0	161.0
3	Tom Hardy	Christian Bale	Joseph Gordon-Levitt	27000.0	23000.0	23000.0
4	Doug Walker	Rob Walker	None	131.0	12.0	NaN

	director_1	director_fb_likes_1
id		
0	James Cameron	0.0
1	Gore Verbinski	563.0
2	Sam Mendes	0.0
3	Christopher Nolan	22000.0
4	Doug Walker	131.0

These tables can now be joined together with movie_table:

```
>>> movie2 = movie_table.merge(directors.reset_index(),
on='id', how='left') \
.merge(actors.reset_index(),
on='id', how='left')
>>> movie.equals(movie2[movie.columns])
True
```

See also

- More on database normalization (<http://bit.ly/2w8wahQ>), associative tables (<http://bit.ly/2yqE4oh>), and primary and foreign keys (<http://bit.ly/2xglvEb>)
- Refer to the video, *Stacking multiple groups of variables simultaneously* in this section for more information on the wide_to_long function

Section 9: Combining Pandas Objects

Appending new rows to DataFrames

Appending a single row to a DataFrame is a fairly expensive operation and if you find yourself writing a loop to append single rows of data to a DataFrame, then you are doing it wrong. Let's first create 1,000 rows of new data as a list of Series:

```
>>> random_data = []
>>> for i in range(1000):
>>>     d = dict()
>>>     for k, v in data_dict.items():
>>>         if isinstance(v, str):
>>>             d[k] = np.random.choice(list('abcde'))
>>>         else:
>>>             d[k] = np.random.randint(10)
>>>     random_data.append(pd.Series(d, name=i + len(bball_16)))
>>> random_data[0].head()
2B 3
3B 9
AB 3
BB 9
CS 4
Name: 16, dtype: object
```

Let's time how long it takes to loop through each item making one append at a time:

```
>>> %%timeit
>>> bball_16_copy = bball_16.copy()
>>> for row in random_data:
>>>     bball_16_copy = bball_16_copy.append(row)
4.88 s ± 190 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

That took nearly five seconds for only 1,000 rows. If we instead pass in the entire list of Series, we get an enormous speed increase:

```
>>> %%timeit
>>> bball_16_copy = bball_16.copy()
>>> bball_16_copy = bball_16_copy.append(random_data)
78.4 ms ± 6.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

By passing in the list of Series, the time has been reduced to under one-tenth of a second. Internally, pandas converts the list of Series to a single DataFrame and then makes the append.

Concatenating multiple DataFrames together

The append method is a heavily watered down version of concat that can only append new rows to a DataFrame. Internally, append just calls the concat function. For instance, step 2 from this video, may be duplicated with the following:

```
>>> stocks_2016.append(stocks_2017)
```

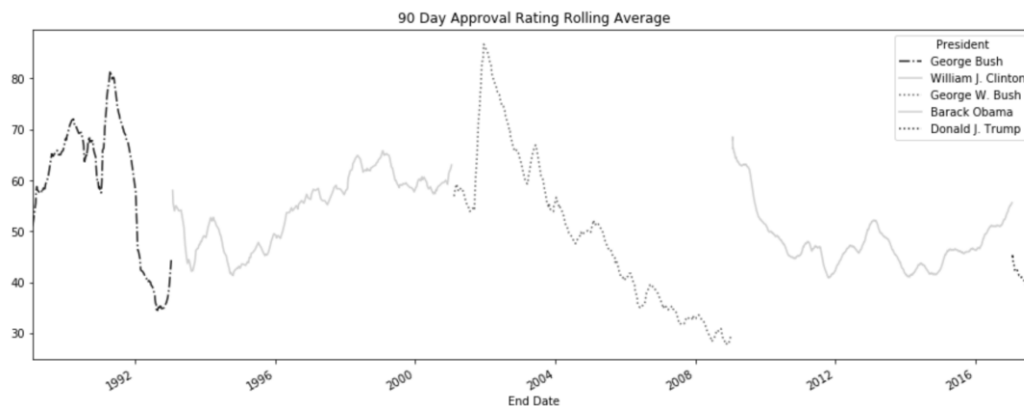
Comparing President Trump's and Obama's approval ratings

The plot from step 19 shows quite a lot of noise and the data might be easier to interpret if it were smoothed. One common smoothing method is called the **rolling average**. Pandas offers the rolling method for DataFrames and groupby objects. It works analogously to the groupby method by returning an object waiting for an additional action to be performed on it. When creating it, you must pass the size of the window as the first argument, which can either be an integer or a date offset string. In this example, we take a 90-day moving average with the date offset string *90D*. The *on* parameter specifies the column from which the rolling window is calculated:

```
>>> pres_rm = pres_41_45.groupby('President', sort=False) \
.rolling('90D', on='End Date')['Approving'] \
.mean()
>>> pres_rm.head()
President End Date
George Bush 1989-01-26 51.000000
1989-02-27 55.500000
1989-03-02 57.666667
1989-03-10 58.750000
1989-03-13 58.200000
Name: Approving, dtype: float64
```

From here, we can restructure the data so that it looks similar to the output from step 23 with the `unstack` method, and then make our plot:

```
>>> styles = ['-.', '-', ':', '-', ':']
>>> colors = [.9, .3, .7, .3, .9]
>>> color = cm.Greys(colors)
>>> title='90 Day Approval Rating Rolling Average'
>>> plot_kwargs = dict(figsize=(16,6), style=styles,
color = color, title=title)
>>> correct_col_order = pres_41_45.President.unique()
>>> pres_rm.unstack('President')[correct_col_order].plot(**plot_kwargs)
```



See also

- Colormap references for matplotlib (<http://bit.ly/2yJZOvt>)
- A list of all the date offsets and their aliases (<http://bit.ly/2xO5Yg0>)

Understanding the differences between concat, join, and merge

It is possible to read all files from a particular directory into DataFrames without knowing their names. Python provides a few ways to iterate through directories, with the glob module being a popular choice. The gas prices directory contains five different CSV files, each having weekly prices of a particular grade of gas beginning from 2007. Each file has just two columns--the date for the week and the price. This is a perfect situation to iterate through all the files, read them into DataFrames, and combine them all together with the concat function. The glob module has the glob function, which takes a single parameter-- the location of the directory you would like to iterate through as a string. To get all the files in the directory, use the string *. In this example, *.csv returns only files that end in .csv. The result from the glob function is a list of string filenames, which can be directly passed to the read_csv function:

```
>>> import glob
>>> df_list = []
>>> for filename in glob.glob('data/gas prices/*.csv'):
df_list.append(pd.read_csv(filename, index_col='Week',
parse_dates=["Week"]))
>>> gas = pd.concat(df_list, axis='columns')
>>> gas.head()
```

	All Grades	Diesel	Midgrade	Premium	Regular
Week					
2017-09-25	2.701	2.788	2.859	3.105	2.583
2017-09-18	2.750	2.791	2.906	3.151	2.634
2017-09-11	2.800	2.802	2.953	3.197	2.685
2017-09-04	2.794	2.758	2.946	3.191	2.679
2017-08-28	2.513	2.605	2.668	2.901	2.399

See also

- IPython official documentation of the read_html function (<http://bit.ly/2fzFRzd>)
- Refer to the video, *Exploding indexes* from Section 6, *Index Alignment*

Connecting to SQL databases

If you are adept with SQL, you can write a SQL query as a string and pass it to the read_sql_query function. For example, the following will reproduce the output from step 4:

```
>>> sql_string1 = ""
select
Name,
time(avg(Milliseconds) / 1000, 'unixepoch') as avg_time
from (
select
g.Name,
```

```

t.Milliseconds
from
genres as g
join
tracks as t
on
g.genreid == t.genreid
)
group by
Name
order by
avg_time
"""
>>> pd.read_sql_query(sql_string1, engine)

```

	Name	avg_time
0	Rock And Roll	00:02:14
1	Opera	00:02:54
2	Hip Hop/Rap	00:02:58

To reproduce the answer from step 6, use the following SQL query:

```

>>> sql_string2 = """
select
c.customerid,
c.FirstName,
c.LastName,
sum(ii.quantity * ii.unitprice) as Total
from
customers as c
join
invoices as i
on c.customerid = i.customerid
join
invoice_items as ii
on i.invoiceid = ii.invoiceid
group by
c.customerid, c.FirstName, c.LastName
order by
Total desc
"""
>>> pd.read_sql_query(sql_string2, engine)

```

	CustomerId	FirstName	LastName	Total
0	6	Helena	Holý	49.62
1	26	Richard	Cunningham	47.62
2	57	Luis	Rojas	46.62

See also

- All engine configurations for *SQLAlchemy* (<http://bit.ly/2kb07vV>)
- Pandas official documentation on *SQL Queries* (<http://bit.ly/2fFsOQ8>)