41_NP_pandas

December 15, 2023

1 Pandas

1.1 What is panadas

- Pandas is a Python library for data analysis and manipulation.
- It is built on top of the NumPy library.
- It offers data structures and operations for manipulating numerical tables and time series.
- It provides data analysis tools for exploratory data analysis, data cleaning, and data transformation.
- It also provides data visualization features.
- It's free to use and open source.

1.2 Installation

To install pandas, you can use the following command:

```
conda install -c conda-forge pandas
```

1.3 Import pandas

To import pandas, you can use the following command:

[ ]:
```
import pandas as pd
```

Note : When importing pandas, it's by convention to use the alias `pd`.

1.4 Pandas - Data Structures

Pandas has two main data structures: `Series` and `DataFrames`.

- A DataFrame is a two-dimensional labeled data structure with columns of potentially different types.
- A Series is a one-dimensional labeled array.

1.5 Pandas - Data Structures

1.5.1 DataFrames

A DataFrame is a table. It contains an array of individual entries, each of which has a certain value. Each entry corresponds to a row (or record) and a column

[ ]:
```
pd.DataFrame({'Yes': [50, 21], 'No': [131, 2]})
```

In the previous example: - `Yes` and `No` are the column names - `50` and `21` are the values in the `Yes` column - `131` and `2` are the values in the `No` column - yes/0 is 50; yes/1 is 21; no/0 is 131; no/1 is 2

1.5.2 DataFrames

The `Dataframes` object is not limited to numbers values. It can contain any data type.

[ ]:
```
df = pd.DataFrame({'Bob': ['I liked it.', 'It was awful.'], 'Sue': ['Pretty␣
↪good.', 'Bland.']})
print(df)
df["Bob"][0]
```

We are using the `pd.DataFrame()` constructor to instaciate these DataFrame objects.

The syntax for declaring a new one is a dictionary whose keys are the column names (`Bob` and `Sue` in this example), and whose values are a list of entries. This is the standard way of constructing a new DataFrame, and the one you are most likely to encounter.

The dictionary-list constructor assigns values to the column labels, but just uses an *ascending count* from 0 (0, 1, 2, 3, ...) for the row labels. Sometimes this is OK, but oftentimes we will want to assign these labels ourselves.

The list of row labels used in a DataFrame is known as an `Index`. We can assign values to it by using an index parameter in our constructor:

[ ]:
```
df = pd.DataFrame({
    'Bob': ['I liked it.', 'It was awful.'],
    'Sue': ['Pretty good.', 'Bland.']
}, index=['Product A', 'Product B'])
df["Bob"]['Product A']
```

[ ]:
```
pd.DataFrame({'Bob': ['I liked it.', 'It was awful.', 2],
              'Sue': ['', 'Bland.', 1]},
             index=['Product A', 'Product B', "Product_count"])
```

1.6 Pandas - Data Structures

1.6.1 Series

A Series, by contrast, is a sequence of data values. If a DataFrame is a table, a Series is a list :

[ ]:
```
s = pd.Series([1, 2, 3, 4, 5])
s
```

Note : In fact you can create one with nothing more than a python list

1.6.2 Series

You can visualize a Series as a single column of a DataFrame. So you can assign row labels to the Series the same way as before, using an `index` parameter. However, a Series does not have a column name, it only has one overall name:

[ ]:
```
serie_label = pd.Series([30, 35, 40], index=['2015 Sales', '2016 Sales', '2017␣
↪Sales'], name='Product A')
print("serie_label", serie_label, sep="\n")
```

1.7 Pandas - Data Structures

1.7.1 Conclusion

The Series and the DataFrame are intimately related. It's helpful to think of a `DataFrame` as actually being just a bunch of Series "glued together".

1.8 Pandas - Reading Data Files

Being able to create a DataFrame or Series by hand is handy. But, most of the time, we won't actually be creating our own data by hand. Instead, we'll be working with data that already exists.

Pandas has built-in methods for reading data from various file formats including CSV, Excel, JSON, and SQL.

1.9 Pandas - Reading Data Files

Here is a table of the different types of files that pandas can read:

Data Format	Read	Write
CSV	pd.read_csv	pd.to_csv
Excel	pd.read_excel	pd.to_excel
JSON	pd.read_json	pd.to_json
SQL	pd.read_sql	pd.to_sql
HTML	pd.read_html	pd.to_html

Note : in this notebook we'll use the CSV file format

[ ]:
```
wine_reviews = pd.read_csv("./data/numpy_pandas/dataset/wine-reviews/
↪winemag-data-130k-v2.csv")
wine_reviews.shape
```

Note : We can see that this dataset has 129971 rows and 14 columns. This is like 1.8 Millions of entries.

1.10 Pandas - Reading Data Files

1.10.1 pd.read_csv()

The `pd.read_csv()` method contains many parameters :

Parameter	Description
filepath_or_buffer	The path to the file to read. Can be a URL.
sep	The separator for the columns. By default, it is a comma.

Parameter	Description
delimiter	The separator for the columns. By default, it is a comma.
header	The row to use as the column names. By default, it is the first row.
names	The column names. By default, it is None.
index_col	The column to use as the index. By default, it is None.
usecols	The columns to use. By default, it is None.
encoding	The encoding to use. By default, it is None.
engine	The engine to use. By default, it is None.
squeeze	If the data is only one column, return a Series. By default, it is False.
skiprows	The rows to skip. By default, it is None.
nrows	The number of rows to read. By default, it is None.
na_values	The values to use for missing values. By default, it is None.
parse_dates	If True, parse the dates. By default, it is False.
infer_datetime_format	If True, infer the datetime format. By default, it is False.

1.11 Pandas - Reading Data Files

1.11.1 Head and Tail

We can visulize the first and last rows of a DataFrame: - If I call directly the `DataFrame` object, I'll see the first 5 rows and the last 5 rows - If I call the `head()` method, I'll see the first 5 rows - If I call the `tail()` method, I'll see the last 5 rows

[ ]:
```
(wine_reviews)
```

[ ]:
```
(wine_reviews.head())
```

[ ]:
```
print(wine_reviews.tail(1))
```

If we look at the data more precisely, we can see that the dataset has a built-in index. If we want to use this column as index for the data we need to specify the `index_col` parameter.

[ ]:
```
wine_reviews = pd.read_csv("./data/numpy_pandas/dataset/wine-reviews/
↪winemag-data-130k-v2.csv", index_col=0)
wine_reviews.head()
```

1.12 Pandas - Indexing, Selecting, Assigning

1.12.1 Introduction

Selecting specific values of a pandas DataFrame or Series to work on is an implicit step in almost any data operation you'll run, so one of the first things you need to learn in working with data in Python is how to go about selecting the data points relevant to you quickly and effectively.

[ ]:
```
pd.set_option('display.max_rows', 10)
reviews = pd.read_csv("./data/numpy_pandas/dataset/wine-reviews/
  ↪winemag-data-130k-v2.csv", index_col=0)
```

1.13 Pandas - Indexing, Selecting, Assigning

1.13.1 Native accessors

Native Python objects provide good ways of indexing data. Pandas carries all of these over, which helps make it easy to start with.

Consider this DataFrame:

[ ]:
```
reviews
```

In Python, we can access the property of an object by accessing it as an attribute. A `book` object, for example, might have a `title` property, which we can access by calling `book.title`. Columns in a pandas DataFrame work in much the same way.

Hence to access the country property of reviews we can use:

[ ]:
```
reviews.country
```

If we have a Python dictionary, we can access its values using the indexing ([]) operator. We can do the same with columns in a DataFrame:

[ ]:
```
reviews['country']
```

Note : These are the two ways of selecting a specific Series out of a DataFrame. Neither of them is more or less syntactically valid than the other, but the indexing operator [] does have the advantage that it can handle column names with reserved characters in them (e.g. if we had a country providence column, reviews.country providence wouldn't work).

[ ]:
```
reviews.country[0]
```

1.14 Pandas - Indexing, Selecting, Assigning

1.14.1 Indexing in Pandas

The indexing operator and attribute selection are nice because they work just like they do in the rest of the Python ecosystem.

However, pandas has its own accessor operators, `loc` and `iloc`. For more advanced operations, these are the ones you're supposed to be using

- `loc` for label-based indexing
- `iloc` for Index-based indexing

Index-based selection `iloc` selects data based on its numerical position in the data. iloc follows this paradigm.

To select the first row of data in a DataFrame, we may use the following:

[ ]:
```
reviews.iloc[0]
```

Note : Both `loc` and `iloc` are row-first, column-second. This is the opposite of what we do in native Python, which is column-first, row-second.

This means that it's marginally easier to retrieve rows, and marginally harder to get retrieve columns.

To get a column with iloc, we can do the following:

[ ]:
```
reviews.iloc[:, 0]
```

On its own, the : operator, which also comes from native Python, means "everything". When combined with other selectors, however, it can be used to indicate a range of values. For example, to select the country column from just the first, second, and third row, we would do:

Or, to select just the second and third entries, we would do:

[ ]:
```
reviews.iloc[:3, 0]
```

It's also possible to pass a list:

[ ]:
```
reviews.iloc[[0, 10, 20], 0]
```

Finally, it's worth knowing that negative numbers can be used in selection.

[ ]:
```
reviews.iloc[-5:, 0]
```

Index-based selection The second paradigm for attribute selection is the one followed by the `loc` operator: **label-based selection**. In this paradigm, it's the data index value, not its position, which matters.

For example, to get the first entry in `reviews`, we would now do the following:

[ ]:
```
reviews.loc[0, 'country']
```

Note : iloc is conceptually simpler than loc because it ignores the dataset's indices. When we use iloc we treat the dataset like a big matrix (a list of lists), one that we have to index into by position. loc, by contrast, uses the information in the indices to do its work

Since your dataset usually has meaningful indices, it's usually easier to do things using loc instead. For example, here's one operation that's much easier using loc:

[ ]:
```
reviews.loc[:, ['taster_name', 'taster_twitter_handle', 'points']]
reviews.iloc[:, [8, 9, 3]]
```

Choosing between `loc` and `iloc` When choosing or transitioning between `loc` and `iloc`, there is one "gotcha" worth keeping in mind, which is that the two methods use slightly different indexing schemes.

`iloc` uses the Python stdlib indexing scheme, where the first element of the range is included and the last one excluded. So 0:10 will select entries 0,…,9. `loc`, meanwhile, indexes inclusively. So 0:10 will select entries 0,…,10.

Why the change? Remember that `loc` can index any stdlib type: strings, for example. If we have a DataFrame with index values Apples, ..., Potatoes, ..., and we want to select "all the alphabetical fruit choices between Apples and Potatoes", then it's a heck of a lot more convenient to index `df.loc['Apples':'Potatoes']` than it is to index something like `df.loc['Apples', 'Potatoed']` (`t` coming after `s` in the alphabet).

This is particularly confusing when the DataFrame index is a simple numerical list, e.g. 0,...,1000. In this case df.iloc[0:1000] will return 1000 entries, while df.loc[0:1000] return 1001 of them! To get 1000 elements using loc, you will need to go one lower and ask for df.loc[0:999].

Otherwise, the semantics of using loc are the same as those for iloc.

Manipulating the index Label-based selection derives its power from the labels in the index. Critically, the index we use is not immutable. We can manipulate the index in any way we see fit.

The `set_index()` method can be used to do the job. Here is what happens when we `set_index` to the `title` field:

[ ]:
```
reviews.set_index("title")
```

Conditional selection So far we've been indexing various strides of data, using structural properties of the DataFrame itself. To do *interesting* things with the data, however, we often need to ask questions based on conditions.

For example, suppose that we're interested specifically in better-than-average wines produced in Italy.

We can start by checking if each wine is Italian or not:

[ ]:
```
reviews.country == 'Italy'
```

This operation produced a Series of True/False booleans based on the country of each record. This result can then be used inside of loc to select the relevant data:

[ ]:
```
italian_wines = reviews.loc[reviews.country == 'Italy']
italian_wines
```

Note : This DataFrame has ~20,000 rows. The original had ~130,000. That means that around 15% of wines originate from Italy.

We also wanted to know which ones are better than average. Wines are reviewed on a 80-to-100 point scale, so this could mean wines that accrued at least 90 points.

We can use the ampersand (`&`) to bring the two questions together:

[ ]:
```
reviews.loc[(reviews.country == 'Italy') & (reviews.points >= 90)]
```

Suppose we'll buy any wine that's made in Italy or which is rated above average. For this we use a pipe (|):

[ ]:
```
reviews.loc[(reviews.country == 'Italy') | (reviews.points >= 90)]
```

1.15 Pandas Built-in conditinal selectors

Selector	Description
`isin()`	Selects rows where the column contains one or more values
`isnull()`	Selects rows where the column is null
`notnull()`	Selects rows where the column is not null
`between()`	Selects rows where the column is between two values
`any()`	Selects rows where the column contains at least one non-null value
`all()`	Selects rows where the column contains all non-null values

[ ]:
```
reviews.loc[reviews.country.isin(['Italy', 'France'])]
```

Note : The second is isnull (and its companion notnull). These methods let you highlight values which are (or are not) empty (NaN). For example, to filter out wines lacking a price tag in the dataset, here's what we would do:

[ ]:
```
reviews.loc[reviews.designation.notnull()]
```

Assagning data Going the other way, assigning data to a DataFrame is easy. You can assign either a constant value:

[ ]:
```
reviews['critic'] = 'everyone'
reviews
```

[ ]:
```
# Or with iterable values
reviews['index_backwards'] = range(len(reviews)-1, -1, -1)
reviews['index_backwards']
```

1.16 Summary functions and Maps

1.16.1 Introduction

In the last tutorial, we learned how to select relevant data out of a DataFrame or Series. Plucking the right data out of our data representation is critical to getting work done, as we demonstrated in the exercises.

However, the data does not always come out of memory in the format we want it in right out of the bat. Sometimes we have to do some more work ourselves to reformat it for the task at hand. This tutorial will cover different operations we can apply to our data to get the input "just right".

[ ]:
```
import numpy as np
```

1.16.2 Summary functions

Pandas provides many simple "summary functions" (not an official name) which restructure the data in some useful way.

For example, consider the `describe()` method:

[ ]:
```
pd.set_option('display.max_rows', 10)
reviews = pd.read_csv("./data/numpy_pandas/dataset/wine-reviews/
  ↪winemag-data-130k-v2.csv", index_col=0)
reviews.points.describe()
```

This method generates a high-level summary of the attributes of the given column. It is type-aware, meaning that its output changes based on the data type of the input. The output above only makes sense for numerical data; for string data here's what we get:

[ ]:
```
reviews.taster_name.describe()
```

If you want to get some particular simple summary statistic about a column in a DataFrame or a Series, there is usually a helpful pandas function that makes it happen.

For example, to see the mean of the points allotted (e.g. how well an averagely rated wine does), we can use the mean() function:

[ ]:
```
reviews.points.mean()
```

To see a list of unique values we can use the unique() function:

[ ]:
```
reviews.taster_name.unique()
```

To see a list of unique values and how often they occur in the dataset, we can use the value_counts() method:

[ ]:
```
reviews.taster_name.value_counts()
```

1.16.3 Maps

A map is a term, borrowed from mathematics, for a function that takes one set of values and "maps" them to another set of values. In data science we often have a need for creating new representations from existing data, or for transforming data from the format it is in now to the format that we want it to be in later. Maps are what handle this work, making them extremely important for getting your work done!

There are two mapping methods that you will use often. - `map()` - `apply()`

`map()` is the first, and slightly simpler one. For example, suppose that we wanted to remean the scores the wines received to 0. We can do this as follows:

[ ]:
```
review_points_mean = reviews.points.mean()
reviews.points.map(lambda p: p - review_points_mean)
```

Note : The function you pass to map() should expect a single value from the Series (a point value, in the above example), and return a transformed version of that value.

map() returns a new Series where all the values have been transformed by your function

apply() is the equivalent method if we want to transform a whole DataFrame by calling a custom method on each row.

[ ]:
```
def remean_points(row):
    row.points = row.points - review_points_mean
    return row

reviews.apply(remean_points, axis='columns')
```

If we had called reviews.apply() with axis='index', then instead of passing a function to transform each row, we would need to give a function to transform each column.

Note : that map() and apply() return new, transformed Series and DataFrames, respectively. They don't modify the original data they're called on. If we look at the first row of reviews, we can see that it still has its original points value.

[ ]:
```
reviews.head(1)
```

Pandas provides many common mapping operations as built-ins. For example, here's a faster way of remeaning our points column:

[ ]:
```
review_points_mean = reviews.points.mean()
reviews.points - review_points_mean
```

In this code we are performing an operation between a lot of values on the left-hand side (everything in the Series) and a single value on the right-hand side (the mean value). Pandas looks at this expression and figures out that we must mean to subtract that mean value from every value in the dataset.

Pandas will also understand what to do if we perform these operations between Series of equal length. For example, an easy way of combining country and region information in the dataset would be to do the following:

[ ]:
```
reviews.country + " - " + reviews.region_1
```

These operators are faster than map() or apply() because they use speed ups built into pandas. All of the standard Python operators (>, <, ==, and so on) work in this manner.

However, they are not as flexible as map() or apply(), which can do more advanced things, like applying conditional logic, which cannot be done with addition and subtraction alone.

1.17 Pandas - Grouping and sorting

1.17.1 Introduction

Maps allow us to transform data in a DataFrame or Series one value at a time for an entire column. However, often we want to group our data, and then do something specific to the group the data is in.

As you'll learn, we do this with the groupby() operation. We'll also cover some additional topics, such as more complex ways to index your DataFrames, along with how to sort your data.

1.17.2 Groupwise analysis

One function we've been using heavily thus far is the value_counts() function. We can replicate what value_counts() does by doing the following:

[ ]:
```
reviews.groupby('points').points.count()
```

groupby() created a group of reviews which allotted the same point values to the given wines. Then, for each of these groups, we grabbed the points() column and counted how many times it appeared. value_counts() is just a shortcut to this groupby() operation.

We can use any of the summary functions we've used before with this data. For example, to get the cheapest wine in each point value category, we can do the following:

[ ]:
```
reviews.groupby('points').price.mean()
```

You can think of each group we generate as being a slice of our DataFrame containing only data with values that match. This DataFrame is accessible to us directly using the apply() method, and we can then manipulate the data in any way we see fit.

For example, here's one way of selecting the name of the first wine reviewed from each winery in the dataset:

[ ]:
```
reviews.groupby('winery').apply(lambda df: df.title.iloc[0])
```

For even more fine-grained control, you can also group by more than one column.

For an example, here's how we would pick out the best wine by country and province:

[ ]:
```
reviews.groupby(['country', 'province']).apply(lambda df: df.loc[df.points.
  ↪idxmax()])
```

Another groupby() method worth mentioning is agg(), which lets you run a bunch of different functions on your DataFrame simultaneously.

For example, we can generate a simple statistical summary of the dataset as follows:

[ ]:
```
reviews.groupby(['country']).price.agg([len, min, max])
```

Note : Effective use of groupby() will allow you to do lots of really powerful things with your dataset

1.17.3 Multi-indexes

In all of the examples we've seen thus far we've been working with DataFrame or Series objects with a single-label index. groupby() is slightly different in the fact that, depending on the operation we run, it will sometimes result in what is called a multi-index.

A multi-index differs from a regular index in that it has multiple levels.

For example:

[ ]:
```
countries_reviewed = reviews.groupby(['country', 'province']).description.
  ↪agg([len])
countries_reviewed
```

[ ]:
```
mi = countries_reviewed.index
type(mi)
```

```
mi
```

Multi-indices have several methods for dealing with their tiered structure which are absent for single-level indices. They also require two levels of labels to retrieve a value. Dealing with multi-index output is a common "gotcha" for users new to pandas.

The use cases for a multi-index are detailed alongside instructions on using them in the MultiIndex / Advanced Selection section of the pandas documentation.

However, in general the multi-index method you will use most often is the one for converting back to a regular index, the reset_index() method:

[ ]:
```
countries_reviewed.reset_index()
```

1.17.4 Sorting

Looking again at countries_reviewed we can see that grouping returns data in index order, not in value order. That is to say, when outputting the result of a groupby, the order of the rows is dependent on the values in the index, not in the data.

To get data in the order want it in we can sort it ourselves. The sort_values() method is handy for this.

[ ]:
```
countries_reviewed = countries_reviewed.reset_index()
countries_reviewed.sort_values(by='len')
```

sort_values() defaults to an ascending sort, where the lowest values go first. However, most of the time we want a descending sort, where the higher numbers go first. That goes thusly:

[ ]:
```
countries_reviewed.sort_values(by='len', ascending=False)
```

To sort by index values, use the companion method sort_index(). This method has the same arguments and default order:

[ ]:
```
countries_reviewed.sort_index()
```

Finally, know that you can sort by more than one column at a time:

[ ]:
```
countries_reviewed.sort_values(by=['country', 'len'])
```

[ ]:
```
countries_reviewed.sort_values(by=['country', 'len'], ascending=[False, True])
```

[ ]:
```
reviews.sort_values(by=["price"], ascending=False).iloc[0,:].name
```

1.18 Pandas - Data types and missing values

1.18.1 Introduction

Pandas can works with differents data types and it will create new data types. We'll also see how we can replace and deal with missing values.

1.18.2 Dtypes

The data type for a column in a DataFrame or a Series is known as the dtype.

You can use the dtype property to grab the type of a specific column. For instance, we can get the dtype of the price column in the reviews DataFrame:

[ ]:
```
reviews.price.dtype
```

Alternatively, the dtypes property returns the dtype of every column in the DataFrame:

[ ]:
```
reviews.dtypes
```

Data types tell us something about how pandas is storing the data internally. float64 means that it's using a 64-bit floating point number; int64 means a similarly sized integer instead, and so on.

One peculiarity to keep in mind (and on display very clearly here) is that columns consisting entirely of strings do not get their own type; they are instead given the object type.

It's possible to convert a column of one type into another wherever such a conversion makes sense by using the astype() function.

For example, we may transform the points column from its existing int64 data type into a float64 data type:

[ ]:
```
print(reviews.points.dtypes)
reviews.points.astype('float64').astype('int64')
```

A DataFrame or Series index has its own dtype, too:

[ ]:
```
reviews.index.dtype
```

1.18.3 Missing data

Entries missing values are given the value NaN, short for "Not a Number". For technical reasons these NaN values are always of the float64 dtype.

Pandas provides some methods specific to missing data. To select NaN entries you can use pd.isnull() (or its companion pd.notnull()). This is meant to be used thusly:

[ ]:
```
reviews[pd.isnull(reviews.country)]
```

Replacing missing values is a common operation. Pandas provides a really handy method for this problem: fillna(). fillna() provides a few different strategies for mitigating such data. For example, we can simply replace each NaN with an "Unknown":

[ ]:
```
reviews.region_2.fillna("Unknown")
```

Or we could fill each missing value with the first non-null value that appears sometime after the given record in the database. This is known as the backfill strategy.

Alternatively, we may have a non-null value that we would like to replace. For example, suppose that since this dataset was published, reviewer Kerin O'Keefe has changed her Twitter handle from @kerinokeefe to @kerino. One way to reflect this in the dataset is using the replace() method:

[ ]:
```
reviews.taster_twitter_handle.replace("@kerinokeefe", "@kerino")
```

The replace() method is worth mentioning here because it's handy for replacing missing data which is given some kind of sentinel value in the dataset: things like "Unknown", "Undisclosed", "Invalid", and so on.

We can also choose to not use a row that has missing data by using the dropna() method:

[ ]:
```
reviews.dropna(inplace=True)
```

the dropna() method will remove any row that has missing data. the inplace parameter will modify the original DataFrame rather than returning a new one.

1.19 Pandas - renamming and combining

1.19.1 Introduction

Oftentimes data will come to us with column names, index names, or other naming conventions that we are not satisfied with. In that case, you'll learn how to use pandas functions to change the names of the offending entries to something better.

You'll also explore how to combine data from multiple DataFrames and/or Series.

1.19.2 Renaming

The first function we'll introduce here is rename(), which lets you change index names and/or column names.

For example, to change the points column in our dataset to score, we would do:

[ ]:
```
reviews.rename(columns={'points': 'score'}, inplace=True)
```

rename() lets you rename index or column values by specifying a index or column keyword parameter, respectively. It supports a variety of input formats, but usually a Python dictionary is the most convenient. Here is an example using it to rename some elements of the index.

[ ]:
```
reviews.rename(index={4: 'firstEntry', 10: 'secondEntry'})
```

You'll probably rename columns very often, but rename index values very rarely. For that, set_index() is usually more convenient.

Both the row index and the column index can have their own name attribute. The complimentary rename_axis() method may be used to change these names. For example:

[ ]:
```
reviews.rename_axis("wines", axis='rows').rename_axis("fields", axis='columns')
```

1.19.3 Combining

When performing operations on a dataset, we will sometimes need to combine different DataFrames and/or Series in non-trivial ways. Pandas has three core methods for doing this. In order of increasing complexity, these are concat(), join(), and merge(). Most of what merge() can do can also be done more simply with join(), so we will omit it and focus on the first two functions here.

The simplest combining method is concat(). Given a list of elements, this function will smush those elements together along an axis.

This is useful when we have data in different DataFrame or Series objects but having the same fields (columns). One example: the YouTube Videos dataset, which splits the data up based on country of origin (e.g. Canada and the UK, in this example). If we want to study multiple countries simultaneously, we can use concat() to smush them together:

[ ]:
```
canadian_youtube = pd.read_csv("./data/numpy_pandas/dataset/youtube/CAvideos.
↪csv")
print(canadian_youtube.shape)
british_youtube = pd.read_csv("./data/numpy_pandas/dataset/youtube/GBvideos.
↪csv")
print(british_youtube.shape)

pd.concat([canadian_youtube, british_youtube])
```

The middlemost combiner in terms of complexity is join(). join() lets you combine different DataFrame objects which have an index in common. For example, to pull down videos that happened to be trending on the same day in both Canada and the UK, we could do the following:

[ ]:
```
left = canadian_youtube.set_index(['title', 'trending_date'])
right = british_youtube.set_index(['title', 'trending_date'])

left.join(right, lsuffix='_CAN', rsuffix='_UK')
```

Here we do a join on two DataFrames based on the index of the two DataFrames. The lsuffix and rsuffix parameters are used to indicate the suffixes of the column names in the left and right DataFrame, respectively.

The lsuffix and rsuffix parameters are necessary here because the data has the same column names in both British and Canadian datasets. If this wasn't true (because, say, we'd renamed them beforehand) we wouldn't need them.

[ ]:
```
np.random.seed()
ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000",␣
↪periods=1000))
ts = ts.cumsum()
ts.plot();
```

[ ]:
```

```