universidad de león

PANDAS

Pandas, the most popular Python library for data analysis.

```
import pandas as pd
```

There are two core objects in pandas: the **DataFrame** and the **Series**.

⇒ A DataFrame is a table. It contains an array of individual entries, each of which has a certain value.

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

DataFrame entries are not limited to integers. For instance, here's a DataFrame whose values are strings:

```
pd.DataFrame({'Bob': ['I liked it.', 'It was awful.'], 'Sue': ['Pretty good.', 'Bland.']})
```

	Bob	Sue
0	I liked it.	Pretty good.
1	It was awful.	Bland.

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

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:

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

	Bob	Sue
Product A	I liked it.	Pretty good.
Product B	It was awful.	Bland.

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

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

A Series is, in essence, a single column of a DataFrame. So you can assign column values 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:

```
pd.Series([30, 35, 40], index=['2015 Sales', '2016 Sales', '2017 Sales'],
name='Product A')
```

```
2015 Sales    30
2016 Sales    35
2017 Sales    40
Name: Product A, dtype: int64
```

Reading data files

Data can be stored in any of a number of different forms and formats. By far the most basic of these is the humble CSV file.

So a CSV file is a table of values separated by commas.

We'll use the pd.read_csv() function to read the data into a DataFrame.

```
wine_reviews = pd.read_csv("../input/wine-reviews/winemag-data-130k-
v2.csv")
```

We can use the shape attribute to check how large the resulting DataFrame is:

```
wine_reviews.shape
```

```
(129971, 14)
```

So our new DataFrame has 130,000 records split across 14 different columns. That's almost 2 million entries!

We can examine the contents of the resultant DataFrame using the head() command, which grabs the first five rows:

```
wine_reviews.head()
```

Use to_csv to save a DataFrame to a CSV file

```
animals.to_csv("cows_and_goats.csv")
```

Native accessors¶

Native Python objects provide good ways of indexing data.

Selecting a specific Series out of a DataFrame.

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']
```

To drill down to a single specific value, we need only use the indexing operator [] once more:

```
reviews['country'][0]
```

Indexing in pandas¶

Pandas has its own accessor operators, loc and iloc.

⇒ Selecting data based on its numerical position in the data. iloc

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

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:

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

: 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.

Finally, it's worth knowing that negative numbers can be used in selection. This will start counting forwards from the end of the values. So, for example here are the last five elements of the dataset.

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

⇒ 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.

```
reviews.loc[0, ‘country']
```

iloc is conceptually simpler than loc because it ignores the dataset's indices.

Manipulating the index

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.

Conditional selection¶

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:

```
reviews.loc[reviews.country == ‘Italy']
```

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.

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

We'll buy any wine that's made in Italy or which is rated above average.

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

Pandas comes with a few built-in conditional selectors.

The first is isin, lets you select data whose value "is in" a list of values. For example, here's how we can use it to select wines only from Italy or France:

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

The second is isnull. 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.price.notnull()]
```

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.

Summary functions¶

Pandas provides many simple "summary functions" which restructure the data in some useful way. For example, consider the **describe()** method:

```
reviews.points.describe()
```

```
count    129971.000000
mean         88.447138
                ...
75%          91.000000
max         100.000000
Name: points, Length: 8, dtype: float64
```

This method generates a high-level summary of the attributes of the given column.

For string data here's what we get:

```
reviews.taster_name.describe()
```

```
count          103727
unique             19
top        Roger Voss
freq            25514
Name: taster_name, dtype: object
```

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()
```

Maps¶

A **map** is a term 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.

There are **two mapping methods** that you will use often.

⇒ map() is the first, and slightly simpler one.

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

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')
```

Note that map() and apply() return new, transformed Series and DataFrames, respectively. They don't modify the original data they're called on.

universidad de león

Group Data

Often, we want to group our data, and then do something specific to the group the data is in.

```
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.

We can use any of the summary functions we've used before with this data. For example:

```
reviews.groupby('points').price.min()
```

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:

```
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.

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

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

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() 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])
```

		len
country	province	
Argentina	Mendoza Province	3264
	Other	536
...	...	...
Uruguay	San Jose	3
	Uruguay	24

```
mi = countries_reviewed.index
type(mi)
```

```
pandas.core.indexes.multi.MultiIndex
```

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()
```

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')
```

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().**

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

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

Dtypes¶

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

```
reviews.price.dtype
```

Data types tell us something about how pandas is storing the data internally.

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

```
reviews.points.astype('float64')
```

Missing data¶

Entries missing values are given the value NaN, short for "Not a Number".

Pandas provides some methods specific to missing data.

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

Replacing missing values is a common operation. Pandas provides a really handy method for this problem: **fillna()**. 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.

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

⇒ 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.

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'})
```

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().

The simplest combining method is **concat().** This is useful when we have data in different DataFrame or Series objects but having the same fields (columns).

```
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')
```

		video_id_CAN	channel_title_CAN	category_id_CAN	publish_time_CAN
title	trending_date				
!! THIS VIDEO IS NOTHING BUT PAIN !! | Getting Over It - Part 7	18.04.01	PNn8sECd7io	Markiplier	20	2018-01-03T19:33:53.000Z
#1 Fortnite World Rank - 2,323 Solo Wins!	18.09.03	DvPW66IFhMI	AlexRamiGaming	20	2018-03-09T07:15:52.000Z
...	...	...	...	...	...

*The lsuffix and rsuffix parameters are necessary here because the data has the same column names in both British and Canadian datasets.