Basic Python 7 - Pandas

December 12, 2020

1 Python Pandas Tutorial

The *pandas* package is the most important tool at the disposal of Data Scientists and Analysts working in Python today.

This tutorial is adapted from Reference.

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

2 Series and DataFrames

The primary two components of pandas are the `Series` and `DataFrame`. A `Series` is essentially a column, and a `DataFrame` is a multi-dimensional table made up of a collection of Series.

[ ]:
```
data = {
    'apples': [3, 2, 0, 1],
    'oranges': [0, 3, 7, 2]
}
```

[ ]:
```
purchases = pd.DataFrame(data)

purchases
```

[ ]:
```
type(purchases)
```

How did that work?

Each *(key, value)* item in `data` corresponds to a *column* in the resulting DataFrame. The **Index** of this DataFrame was given to us on creation as the numbers 0-3, but we could also create our own when we initialize the DataFrame. Let's have customer names as our index:

[ ]:
```
purchases = pd.DataFrame(data, index=['June', 'Robert', 'Lily', 'David'])

purchases
```

So now we could **loc**ate a customer's order by using their name:

[ ]:
```
purchases.loc['June']
```

[ ]:
```
purchases.loc['June':'Lily']
```

```
[ ]: purchases.loc['June':'Lily',['apples']]
```

3 Reading data from files

```
[ ]: df = pd.read_csv('purchases.csv')

     df
```

We can designate the `index_col` when reading:

```
[ ]: df = pd.read_csv('purchases.csv', index_col=0)

     df
```

```
[ ]: df = pd.read_excel('purchases.xlsx', index_col=0)

     df
```

4 Write data to files

```
[ ]: df.to_csv('new_purchases.csv')
```

```
[ ]: df.to_excel('new_purchases.xlsx')
```

5 Important DataFrame operations

Let's load in the IMDB movies dataset to begin:

```
[ ]: movies_df = pd.read_csv("IMDB-Movie-Data.csv", index_col="Title")

     #You can use this link too.
     #url = 'https://drive.google.com/file/d/19LKLek-xyLm48W4fuF03lpcIIcDY2jJ1/view?
      ↪usp=sharing'
     #path = 'https://drive.google.com/uc?export=download&id='+url.split('/')[-2]
     #movie_df = pd.read_csv(path)
```

We're loading this dataset from a CSV and designating the movie titles to be our index.

5.1 Viewing data

The first thing to do when opening a new dataset is print out a few rows to keep as a visual reference. We accomplish this with `.head()`:

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

`.head()` outputs the **first** five rows of your DataFrame by default, but we could also pass a number as well: `movies_df.head(10)` would output the top ten rows, for example.

To see the **last** five rows use `.tail()`. `tail()` also accepts a number, and in this case we printing the bottom two rows.:

[ ]:
```
movies_df.tail(2)
```

Typically when we load in a dataset, we like to view the first five or so rows to see what's under the hood. Here we can see the names of each column, the index, and examples of values in each row.

You'll notice that the index in our DataFrame is the *Title* column, which you can tell by how the word *Title* is slightly lower than the rest of the columns.

5.2 Getting data info

`.info()` should be one of the very first commands you run after loading your data:

[ ]:
```
movies_df.info()
```

`.info()` provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using. string objects.

Another fast and useful attribute is `.shape`, which outputs just a tuple of (rows, columns):

[ ]:
```
movies_df.shape
```

5.3 Column cleanup

Print the column names of our dataset:

[ ]:
```
movies_df.columns
```

We can use the `.rename()` method to rename certain or all columns via a `dict`. We don't want parentheses, so let's rename those:

[ ]:
```
movies_df.rename(columns={
        'Runtime (Minutes)': 'Runtime',
        'Revenue (Millions)': 'Revenue_millions'
    }, inplace=True)

movies_df.columns
```

If we want to lowercase all names? Instead of using `.rename()` we could also set a list of names to the columns like so:

[ ]:
```
movies_df.columns = ['rank', 'genre', 'description', 'director', 'actors',␣
 ↪'year', 'runtime',
                     'rating', 'votes', 'revenue_millions', 'metascore']
```

Better way

[ ]:
```
c = movies_df.columns
cnew = []
for name in c:
    cnew.append(name.upper())

movies_df.columns = cnew

movies_df.columns
```

A shorter code

[ ]:
```
movies_df.columns = [col.lower() for col in movies_df]

movies_df.columns
```

`list` (and `dict`) comprehensions come in handy a lot when working with pandas and data in general.

It's a good idea to lowercase, remove special characters, and replace spaces with underscores if you'll be working with a dataset for some time.

5.4 Understanding your variables

Using `describe()` on an entire DataFrame we can get a summary of the distribution of continuous variables:

[ ]:
```
movies_df.describe()
```

Understanding which numbers are continuous also comes in handy when thinking about the type of plot to use to represent your data visually.

`.describe()` can also be used on a categorical variable to get the count of rows, unique count of categories, top category, and freq of top category:

[ ]:
```
movies_df['genre'].describe()
```

This tells us that the genre column has 207 unique values, the top value is Action/Adventure/Sci-Fi, which shows up 50 times (freq).

`.value_counts()` can tell us the frequency of all values in a column:

[ ]:
```
movies_df['genre'].value_counts().head(10)
```

5.5 DataFrame slicing, selecting, extracting

Below are the other methods of slicing, selecting, and extracting you'll need to use constantly.

It's important to note that, although many methods are the same, DataFrames and Series have different attributes, so you'll need be sure to know which type you are working with or else you will receive attribute errors.

Let's look at working with columns first.

5.5.1 By column

You already saw how to extract a column using square brackets like this:

[ ]:
```
genre_col = movies_df['genre']

type(genre_col)
```

This will return a *Series*. To extract a column as a *DataFrame*, you need to pass a list of column names. In our case that's just a single column:

[ ]:
```
genre_col = movies_df[['genre']]

type(genre_col)
```

Since it's just a list, adding another column name is easy:

[ ]:
```
subset = movies_df[['genre', 'rating']]

subset.head()
```

Now we'll look at getting data by rows.

5.5.2 By rows

For rows, we have two options:

- `.loc` - **loc**ates by name
- `.iloc`- **loc**ates by numerical **i**ndex

Remember that we are still indexed by movie Title, so to use `.loc` we give it the Title of a movie:

[ ]:
```
prom = movies_df.loc["Prometheus"]

prom
```

On the other hand, with `iloc` we give it the numerical index of Prometheus:

[ ]:
```
prom = movies_df.iloc[1]
```

`loc` and `iloc` can be thought of as similar to Python `list` slicing. To show this even further, let's select multiple rows.

How would you do it with a list? In Python, just slice with brackets like `example_list[1:4]`. It's works the same way in pandas:

[ ]:
```
movie_subset = movies_df.loc['Prometheus':'Sing']

movie_subset = movies_df.iloc[1:4]

movie_subset
```

One important distinction between using `.loc` and `.iloc` to select multiple rows is that `.loc` includes the movie *Sing* in the result, but when using `.iloc` we're getting rows 1:4 but the movie at index 4 (*Suicide Squad*) is not included.

Slicing with `.iloc` follows the same rules as slicing with lists, the object at the index at the end is not included.

5.6 Conditional selections

We've gone over how to select columns and rows, but what if we want to make a conditional selection?

For example, what if we want to filter our movies DataFrame to show only films directed by Ridley Scott or films with a rating greater than or equal to 8.0?

To do that, we take a column from the DataFrame and apply a Boolean condition to it. Here's an example of a Boolean condition:

[ ]:
```
condition = (movies_df['director'] == "Ridley Scott")

condition.head()
```

Similar to `isnull()`, this returns a Series of True and False values: True for films directed by Ridley Scott and False for ones not directed by him.

We want to filter out all movies not directed by Ridley Scott, in other words, we don't want the False films. To return the rows where that condition is True we have to pass this operation into the DataFrame:

[ ]:
```
movies_df[movies_df['director'] == "Ridley Scott"].head()
#A safer way is to use loc.
#movies_df.loc[movies_df['director'] == "Ridley Scott"].head()
```

You can get used to looking at these conditionals by reading it like:

> Select movies_df where movies_df director equals Ridley Scott

Let's look at conditional selections using numerical values by filtering the DataFrame by ratings:

[ ]:
```
movies_df[movies_df['rating'] >= 8.6].head(3)
```

We can make some richer conditionals by using logical operators `|` for "or" and `&` for "and".

Let's filter the the DataFrame to show only movies by Christopher Nolan OR Ridley Scott:

[ ]:
```
movies_df[(movies_df['director'] == 'Christopher Nolan') |␣
↪(movies_df['director'] == 'Ridley Scott')].head()
```

We need to make sure to group evaluations with parentheses so Python knows how to evaluate the conditional.

Using the `isin()` method we could make this more concise though:

```
[ ]: movies_df[movies_df['director'].isin(['Christopher Nolan', 'Ridley Scott'])].
     ↪head()
```

Let's say we want all movies that were released between 2005 and 2010, have a rating above 8.0, but made below the 25th percentile in revenue.

Here's how we could do all of that:

```
[ ]: movies_df[
         ((movies_df['year'] >= 2005) & (movies_df['year'] <= 2010))
         & (movies_df['rating'] > 8.0)
         & (movies_df['revenue_millions'] < movies_df['revenue_millions'].quantile(0.
     ↪25))
     ]
```

If you recall up when we used `.describe()` the 25th percentile for revenue was about 17.4, and we can access this value directly by using the `quantile()` method with a float of 0.25.

So here we have only four movies that match that criteria.

5.7 Applying functions

It is possible to iterate over a DataFrame or Series as you would with a list, but doing so — especially on large datasets — is very slow.

An efficient alternative is to `apply()` a function to the dataset. For example, we could use a function to convert movies with an 8.0 or greater to a string value of "good" and the rest to "bad" and use this transformed values to create a new column.

First we would create a function that, when given a rating, determines if it's good or bad:

```
[ ]: def rating_function(x):
         if x >= 8.0:
             return "good"
         else:
             return "bad"
```

Now we want to send the entire rating column through this function, which is what `apply()` does:

```
[ ]: movies_df["rating_category"] = movies_df["rating"].apply(rating_function)

     movies_df.head(2)
```

The `.apply()` method passes every value in the `rating` column through the `rating_function` and then returns a new Series. This Series is then assigned to a new column called `rating_category`.

You can also use anonymous functions as well. This lambda function achieves the same result as `rating_function`:

```
[ ]: movies_df["rating_category"] = movies_df["rating"].apply(lambda x: 'good' if x␣
     ↪>= 8.0 else 'bad')
```

```
movies_df.head(2)
```

Overall, using `apply()` will be much faster than iterating manually over rows because pandas is utilizing vectorization.

> Vectorization: a style of computer programming where operations are applied to whole arrays instead of individual elements —Wikipedia

A good example of high usage of `apply()` is during natural language processing (NLP) work. You'll need to apply all sorts of text cleaning functions to strings to prepare for machine learning.