5 pandas functions I found to be useful for specific operations

José Fernando Costa

(source)

pandas is a wonderful library to work with data in Python. If you're accustomed to tabular data, then you will feel right at home with this pandas, better yet, while writing Python code. I've started working with this library a couple years ago, but I only started using it seriously last year. In this period, I've come across many useful functions and so today I will briefly show-off five that have stood out to me for their applications.

Categorical

Sometimes there is a need for a custom sorting order. If you try to use the `sort_values` function in a column with month names, the result will use alphabetical order, not the natural order. In these cases, it is better to use a Categorical column, which is created with the `Categorical` function.

In essence, each value's data type becomes *category*, but the important bit is what you use as the *categories* argument of the function call. If you pass it a list or array with the proper order of the months, then pandas will remember to use that order when sorting the months in the column.

```
months = ["October", "November", "December"]
df["Month"] = pd.Categorical(df["Month"], months)
df.sort_values("Month")
```

	Day	Month
0	1	October
2	3	November
1	2	December

Original

	Day	Month
1	2	December
2	3	November
0	1	October

sort_values

	Day	Month
0	1	October
2	3	November
1	2	December

sort_values with Categorical

Various orderings of the "Month" column

If you try to `sort_values` immediately, December will be

the first month even though it should be the last. Try first converting the column using `Categorical` with a list of the ordered months and your problem will be solved.

reset_index

I find myself coming back to this function time and time again. Even in the example of `Categorical` I would have used it. Basically every time I use a sorting function I also use `reset_index` afterwards to return the index to normal sequence (note how the index order in the last DataFrame of `Categorical`'s example is 0, 2, 1).

```
df.reset_index(drop=True)
```

However, more than just resetting the index to a normal numerical sequence, I have also used `reset_index` to make the current index a normal column of the DataFrame.

```
df.reset_index()
```

This is the default behaviour and not desired if you simply want to have a ordered numeric index again, but you may need to manipulate the index values directly if you're working with a multi-level index.

IntervalIndex.from_tuples

Let's say you need to bin a column of numeric data and the `cut` function's automatic bins are not the ones you need. In that case, you can create your own bins using `IntervalIndex.from_tuples`.

If a column of ages has numbers between 0 and 90, inclusive, and you want ten-number bins, then you could easily create the bin limits as tuples with a list comprehension and pass it to `IntervalIndex.from_tuples` to create an IntervalIndex from it.

```
age_bins = pd.IntervalIndex.from_tuples(
    [(i, i+10) for i in range(0, 91, 10)],
    closed="left"
)
```

The above would create bins closed on the left side and open on the right, such as [0, 10) and [30, 40). Supposing you use the same list comprehension to create the bin labels, but this time as strings

```
bin_labels = [f"[{i}, {i+10})" for i in range(0,
```

Now you'd be ready to bin the column of ages using your custom bins.

```
ages_binned = pd.cut(
    df["age"],
```

```
    age_bins,
    labels=bin_labels,
    precision=0,
    include_lowest=True
)
```

Result of using custom bins

Note that what is shown in the “ages_binned” column are the labels, not the bins themselves. It may seem so because this example uses strings of the bins as labels.

This way you’re able to easily generate the bins and labels with list comprehensions, but also reuse those values for the binning operation.

stack

If you had a DataFrame containing the social media used by different users as shown in the next screenshot:

DataFrame with bad data

How would you analyse which are the most used social media? It’s easy for the last two users with single options, but what about Person A that uses two and the data is recorded in a single cell, having the values separated by a semicolon? The answer would be to split the individual values into multiple columns, i.e., two columns for Person A, one for the other two people, and then *stack* those columns into a single one, repeating the User and/or index to know who that social media pertains to.

Yup, this is why I wanted to mention `stack` in this blog post: to handle multiple values in a single cell that need to be separated.

```
unpivoted_data = df["Social Media"]\
        .apply(lambda series_row: pd.Series(serie
        .stack()
unpivoted_data = unpivoted_data.reset_index()
```

Transformed result

Now each social media is on its own row, while keeping the person ids (“level_0”), and now also a order for each of their choices (“level_1”). Note these automatic column names are default behaviour from pandas, but you can rename them afterwards. If you’d like to see a complete example please refer to a previous blog post of mine (this is the same post I linked for `reset_index`).

query

The last function I want to mention is `query`. I like to think of it as the equivalent of f-strings in terms of usefulness, but for filtering DataFrames.

For example, if you have a DataFrame with heights of people in centimeters and want to filter by the people shorter than 1 meter/100 centimeters, then you could use the common conditional syntax:

```
below_1_meter = df[df["Height"] < 100]
```

Or use `query` and write this condition in a more natural way

```
below_1_meter = df.query("Height < 100")
```

Both return the exact same result, but the `query` option is less verbose and lets you use column names directly as variables.