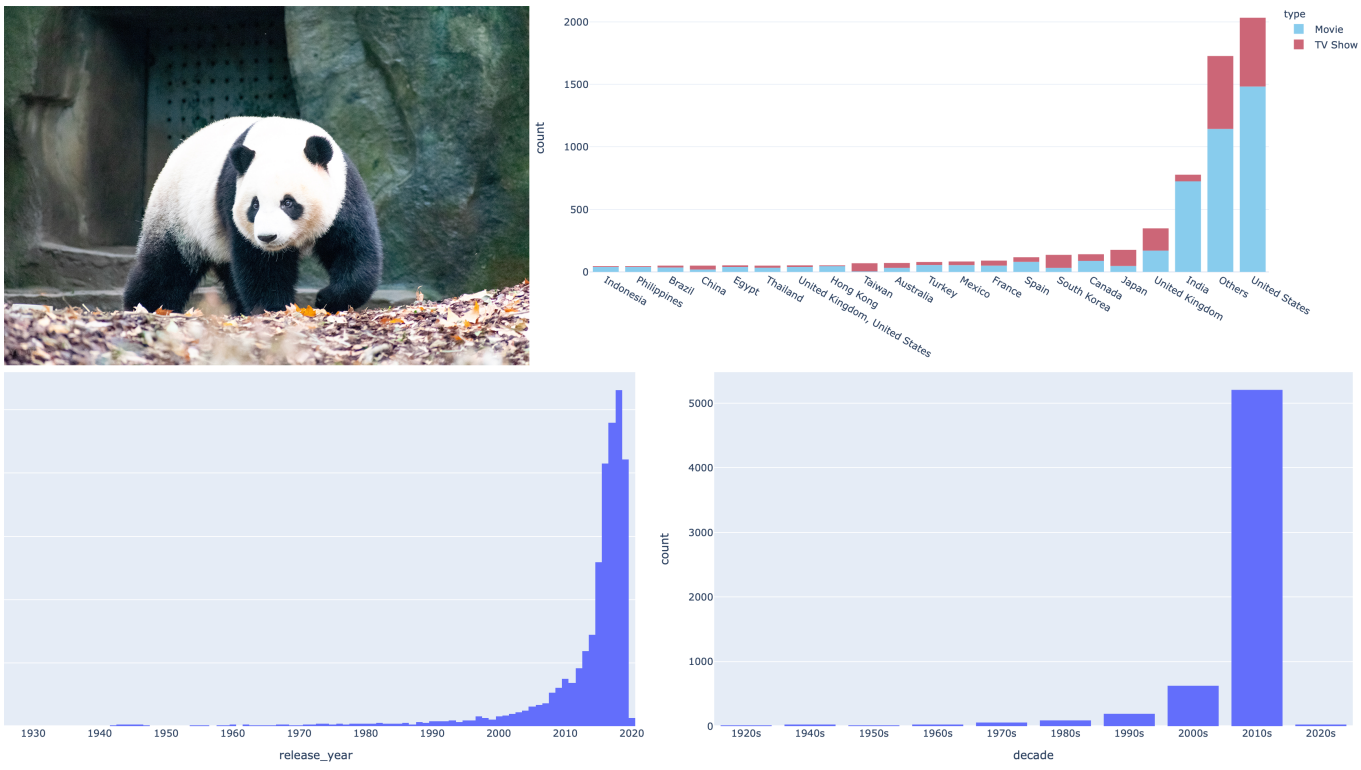
Two Pandas functions you must know for easy data manipulation in Python

Master these pandas functions (and methods) to shorten your code, improve performance and avoid headaches.

JP Hwang Follow
Mar 18 · 6 min read ★



Pandas — one of these are not like the others (Top right photo by Stan W. on Unsplash)

I do a lot of data work — some of it data science, some of it data visualisation, some of it data adjacent — like messing around with sports analytics. All of it is done in Python, and almost always Pandas is involved for data manipulation.

I like to think that in all that time I learned a thing or two about using Pandas. The way I use Pandas is very different from how I used to when I first started dabbling in it a few

years ago. The more I use Pandas, the less I find myself stepping outside of it for data manipulation. My impression is that my code has become more succinct and I've saved a lot of time as a result.

So in this article, I'd like to share with you all a few functions that I now can't do without — I am sure that you'll find them as useful as I did.

It's the time to stay indoors, so today I use the Netflix movie dataset from Kaggle. If nothing else, you might have discovered something to watch :).

Before we get started

`Packages`

I assume you're familiar with python. Even if you're relatively new, this tutorial shouldn't be too tricky, though.

You'll need `pandas` and `plotly`. Install each (in your virtual environment) with a simple `pip install [PACKAGE_NAME]`.

Content, content everywhere

Load the Netflix `csv` data file into a Pandas DataFrame with:

```
nf_df = pd.read_csv('srcdata/netflix_titles.csv')
```

Let's inspect the DataFrame with the `.head()` method.

Actually, that leads me to my first tip.

Set display options in Pandas

If you're like me, and you *don't* work in Jupyter notebook or its variants, you might have been frustrated by the limited display width of Pandas in Python shells.

This is what your output might look like when running `nf_df.head()`.

```
1  >>> nf_df.head()
```

```
    show_id  ...                                        description
0  81145628  ...  Before planning an awesome wedding for his gra...
1  80117401  ...  Jandino Asporaat riffs on the challenges of ra...
2  70234439  ...  With the help of three human allies, the Autob...
3  80058654  ...  When a prison ship crash unleashes hundreds of...
4  80125979  ...  When nerdy high schooler Dani finally attracts...

[5 rows x 12 columns]
```

pandas_manipulation_head_short.py hosted with ♡ by GitHub view raw

This doesn't tell you that much...

It's not all that useful. But wait! Pandas has a `.set_option` function which you can use to adjust the number of columns to display, and the overall width in which to display it. Set your parameters to something like:

```
pd.set_option('display.max_columns', desired_cols)
pd.set_option('display.width', desired_width)
```

And now your shell will output something far more useful.

```
>>> nf_df.head()
    show_id     type                                  title                 director
0  81145628    Movie  Norm of the North: King Sized Adventure  Richard Finn, Tim Maltby  Alan Mar
1  80117401    Movie                Jandino: Whatever it Takes                      NaN
2  70234439  TV Show                    Transformers Prime                      NaN  Peter Cu
3  80058654  TV Show        Transformers: Robots in Disguise                      NaN  Will Fri
4  80125979    Movie                             #realityhigh          Fernando Lebrija  Nesta Co
```

pandas_manipulation_head hosted with ♡ by GitHub view raw

A much more useful output

There are other options which can be set via the `.set_option` function— take a look at the documentation here if you are interested.

.assign

Have you ever come across this message?

```
<input>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

If this has never happened to you, you are a better person than I — you might have even learned Pandas properly through docs or a book.

While the official explanation from Pandas can be found here; but the tl;dr version is that Pandas is warning you that it can't be certain whether you are operating on a *copy* of the DataFrame (in which case you might not be changing any values).

This is one of the reasons to use `.assign`. Using `.assign`, a *new object* is returned to avoid potential confusion due to a `SettingWithCopyWarning`, which is often raised regardless of whether an actual problem exists.

Following on, a more obvious reason is to create a *new* DataFrame where the existing DataFrame is left alone for whatever reason. As I mentioned, `.assign` returns a new object, so we can avoid any problems like this one:

```python
# Avoid accidentally modifying the source object
nf_df = pd.read_csv('srcdata/netflix_titles.csv')
nf_test1_df = nf_df
nf_test1_df['like'] = True
print(len(nf_df.columns))

nf_df = pd.read_csv('srcdata/netflix_titles.csv')
nf_test2_df = nf_df
nf_test2_df = nf_test2_df.assign(like=True)
print(len(nf_df.columns))
```

pandas_manipulation_assign_eg1.py hosted with ♡ by GitHub view raw

The first example prints 13, as we have accidentally modified the original DataFrame (`nf_df`), while in the second, the original DataFrame remains unchanged with 12 columns.

Now, let's move on to `.apply` — which might be one of the most underrated Pandas functions :).

.apply

If you're like me, you have probably tried to manipulate pandas data in a loop — maybe doing something like:

```
for i in range(len(nf_df)):
    nf_df.iloc[i, 1] = ...some function here
```

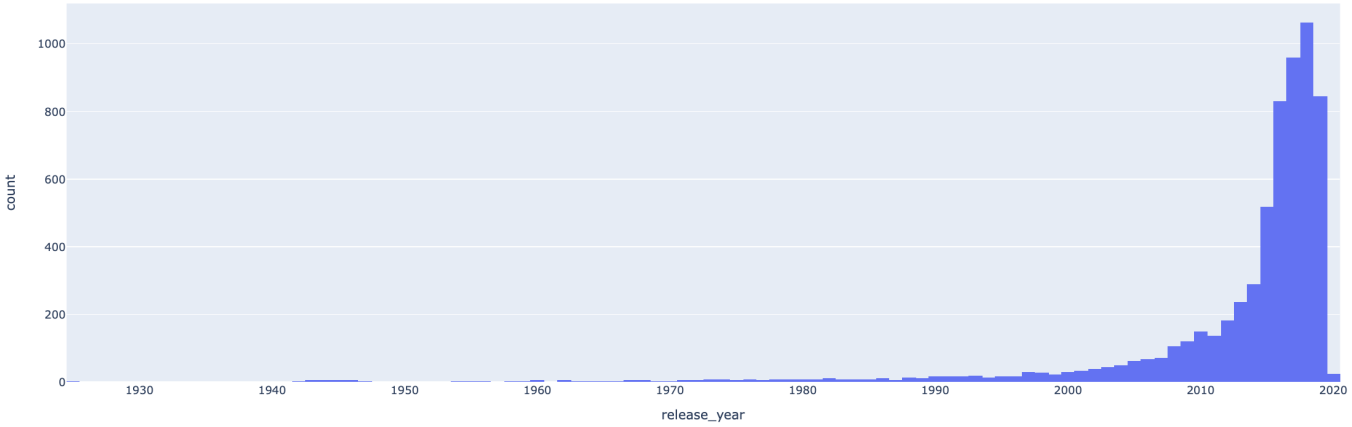
Stop. There's no need for that.

That's what the `.apply` method is there for. It applies a function to every element along an axis of the DataFrame. Let's take a look at some examples.

Example 1 — Content count by release years

Our DataFrame includes a column with the release year (`release_year`). The distribution can be plotted as below, and looks like this:

```
fig = px.histogram(nf_df, x='release_year')
fig.show()
```



Histogram of release data for Netflix titles by year

What if we wanted to look at the data by decade? Simple — create a new column `'decade'` as below:

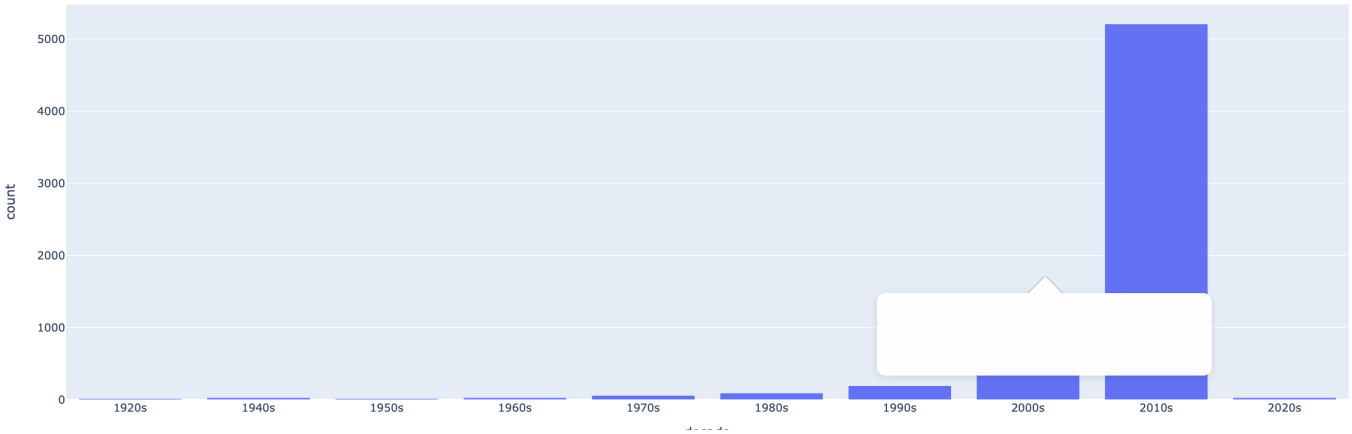
```
nf_df['decade'] = nf_df.release_year.apply(lambda x: str(x)[:3]+'0s')
```

Here, we get the `release_year` column data by `nf_df.release_year`, and apply the lambda function `lambda x: str(x)[:3]+'0s'`.

Lambda functions can look confusing, but it is quite simple once you get used to it. Here, it grabs every element (`x`), and first applies the transformation `str(x)[:3]` to get the first 3 letters of the year string, and add `'0s'` to the end.

So, take a look at a histogram of our new decade column:

```
fig = px.histogram(nf_df, x='decade', category_orders={'decade':
np.sort(nf_df.decade.unique())})
fig.show()
```



Histogram of release data for Netflix titles by decade

See how simple that was?

Also, you might have noticed my using `.unique()` method. It is an extremely useful method that will return an array of unique entities in that column — don't go without it!

Okay. Let's go through one more example of using `.apply`.

Example 2 — Content count by source country

The DataFrame also includes a column (‘`country`’) that (to my best knowledge) includes the source of the content.

A quick look at the data (`nf_df['country'].nunique()`) reveals that there are 555(!) unique entities in the `country` column. What is going on?

As it turns out, many listings include multiple country names (e.g. ‘`United States, India, South Korea, China`’) — so, let’s preserve the top 20 names, and change all the remaining names to ‘Others’.

It’s easy to do with `.apply`. I’ll use two lines of code for clarity, but it’s possible in just one. Ready?

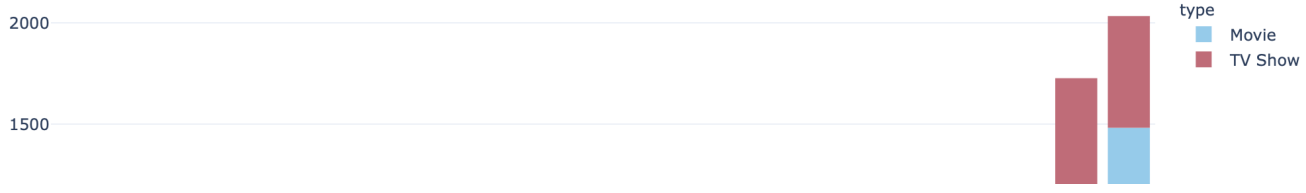
```
top_countries = nf_df.groupby('country')
['title'].count().sort_values().index
nf_df['country'] = nf_df.country.apply(lambda x: 'Others' if (x not
in top_countries[-20:]) else x)
```

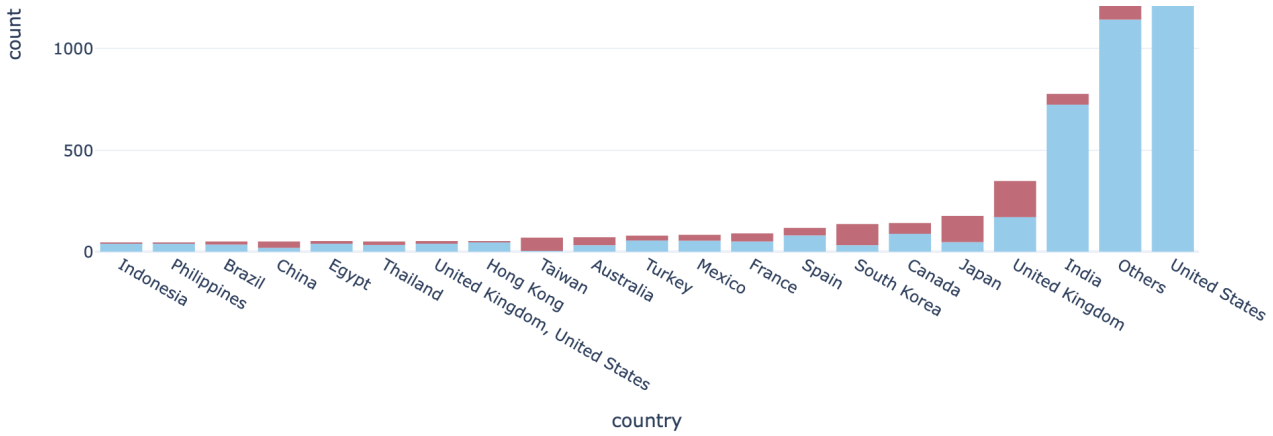
And that’s it! Running `nf_df['country'].nunique()` again, we see that there are only 21 ‘countries’ now.

The lambda function here is a simple if… else statement, checking the ‘`country`’ entity against the list with the highest counts.

Given that each column also includes the content type column, we can visualise breakdowns of content data by colouring in each bar by content type.

```
fig = px.histogram(nf_df, x='country', color='type', category_orders=
{'country': top_countries},
color_discrete_sequence=px.colors.qualitative.Safe,
template='plotly_white')
fig.show()
```





Histogram of Netflix titles by source country and type

Isn't that easy? Imagine doing that with some crazy looped function — finding the right indexer row/column numbers through `.loc` or `.iloc` indexers. No, thank you.

Neat, right?

With just these two short functions, you can do the same things that probably used to take lines and lines of code. Try it out — I'd bet that you'll be pleasantly surprised, and improve your productivity.

. . .

If you liked this, say 👏 / follow on twitter, or follow here for updates. ICYMI: I also wrote this article about building a web data dashboard with Plotly Dash.

Build a web data dashboard in just minutes with Python

Exponentially increase power & accessibility by converting your data visualizations into a web-based dashboard with...

towardsdatascience.com

And also this article about visualising data across time for storytelling:

Effectively visualize data across time to tell better stories

Build clean, easy-to-read time-series data visualizations to support your

narratives with Python and Plotly.

towardsdatascience.com

Programming Data Science Coding Technology Learning

About Help Legal