Guide to Cleaning and Preparing Data in Python

1. Quick Dataset Overview

The first thing to do once you downloaded a dataset is to check the data type of each column (the values of a column might contain digits, but they might not be datetime or int type)

After reading the CSV file, type .dtypes to find the data type of each column.

```
df_netflix_2019 = pd.read_csv('netflix_titles.csv')
```

DataSet Link: Netflix Movies and TV Shows | Kaggle

```
df_netflix_2019.dtypes
```

Once you run that code, you'll get the following output.

```
show_id          int64
type            object
title           object
director        object
cast            object
country         object
date_added      object
release_year     int64
rating          object
duration        object
listed_in       object
description     object
dtype: object
```

This will help you identify whether the columns are numeric or categorical variables, which is important to know before cleaning the data.

Now to find the number of rows and columns, the dataset contains, use the .shape method.

```
In [1]: df_netflix_2019.shape
Out[1]: (6234, 12) #This dataset contains 6234 rows and 12 columns.
```

2. Identify Missing Data

Missing data sometimes occurs when data collection was done improperly, mistakes were made in data entry, or data values were not stored. This happens often, and we should know how to identify it.

Create a percentage list with .isnull()

A simple approach to identifying missing data is to use the `.isnull()` and `.sum()` methods

`df_netflix_2019.isnull().sum()`

This shows us a number of "NaN" values in each column. If the data contains many columns, you can use `.sort_values(ascending=False)` to place the columns with the highest number of missing values on top.

```
show_id            0
type               0
title              0
director        1969
cast             570
country          476
date_added        11
release_year       0
rating            10
duration           0
listed_in          0
description        0
dtype: int64
```

That being said, I usually represent the missing values in percentages, so I have a clearer picture of the missing data. The following code shows the above output in `%`

```
# % of rows missing in each column
for column in df_netflix_2019.columns:
    percentage = df_netflix_2019[column].isnull().mean()
    print(f'{column}: {round(percentage*100, 2)}%')
```

Now it's more evident that a good number of directors were omitted in the dataset.

```
show_id: 0.0%
type: 0.0%
title: 0.0%
director: 31.58%
cast: 9.14%
country: 7.64%
date_added: 0.18%
release_year: 0.0%
rating: 0.16%
duration: 0.0%
listed_in: 0.0%
description: 0.0%
```

Now that we identified the missing data, we have to manage it.

3. Dealing with Missing Data

There are different ways of dealing with missing data. The correct approach to handling missing data will be highly influenced by the data and goals your project has. That being said, the following cover 3 simple ways of dealing with missing data.

Remove a column or row with .drop, .dropna or .isnull

If you consider it's necessary to remove a column because it has too many empty rows, you can use .drop() and add axis=1 as a parameter to indicate that what you want to drop is a column.

```
#drop column
df_netflix_2019.drop('director', axis=1)
```

However, most of the time is just enough to remove the rows containing those empty values. There are different ways to do so.

```
#drop row
no_director = df_netflix_2019[df_netflix_2019['director'].isnull()].index
df_netflix_2019.drop(no_director, axis=0)
#~ + .isnull()
df_netflix_2019[~df_netflix_2019['director'].isnull()]
#dropna()
df_netflix_2019.dropna(subset=['director'])
```

The first solution uses .drop with axis=0 to drop a row. The second identifies the empty values and takes the non-empty values by using the negation operator ~ while the third solution uses .dropna to drop empty rows within a column.

If you want to save the output after dropping, use inplace=True as a parameter. In this simple example, we'll not drop any column or row.

Replace it by the mean, median or mode

Another common approach is to use the mean, median or mode to replace the empty values. The mean and median are used to replace numeric data, while the mode replaces categorical data.

As we've seen before, the rating column contains 0.16% of missing data. We could easily complete that tiny portion of data with the mode since the rating is a categorical value.

```
mode = ''.join(df_netflix_2019['rating'].mode())
df_netflix_2019['rating'].fillna(mode, inplace=True)
```

First, we calculated the mode (TV-MA), and then we filled all the empty values with .fillna.

Replace it by an arbitrary number with .fillna()

If the data is numeric, we can also set an arbitrary number to prevent removing any row without affecting our model's results.

If the duration column was a numeric value (currently, the format is string e.g. 90 minutes), we could replace the empty values by 0 with the following code.

```
df_netflix_2019['duration'].fillna(0, inplace=True)
```

Also, you can use the ffill , bfill to propagate the last valid observation forward and backward, respectively. This is extremely useful for some datasets but it's not useful in the df_netflix_2019 dataset.

4. Identifying Outliers

An outlier is that data that that differs significantly from other observations. A dataset might contain real outliers or outliers obtained after poor data collection or caused by data entry errors.

Using histograms to identify outliers within numeric data

We're going to use the duration as a reference that will help us identify outliers in the Netflix catalog. The duration column is not considered a numerical value (e.g., 90) in our dataset because it's mixed with strings (e.g., 90 min). Also, the duration of TV shows is in seasons (e.g., 2 seasons) so we need to filter it out.

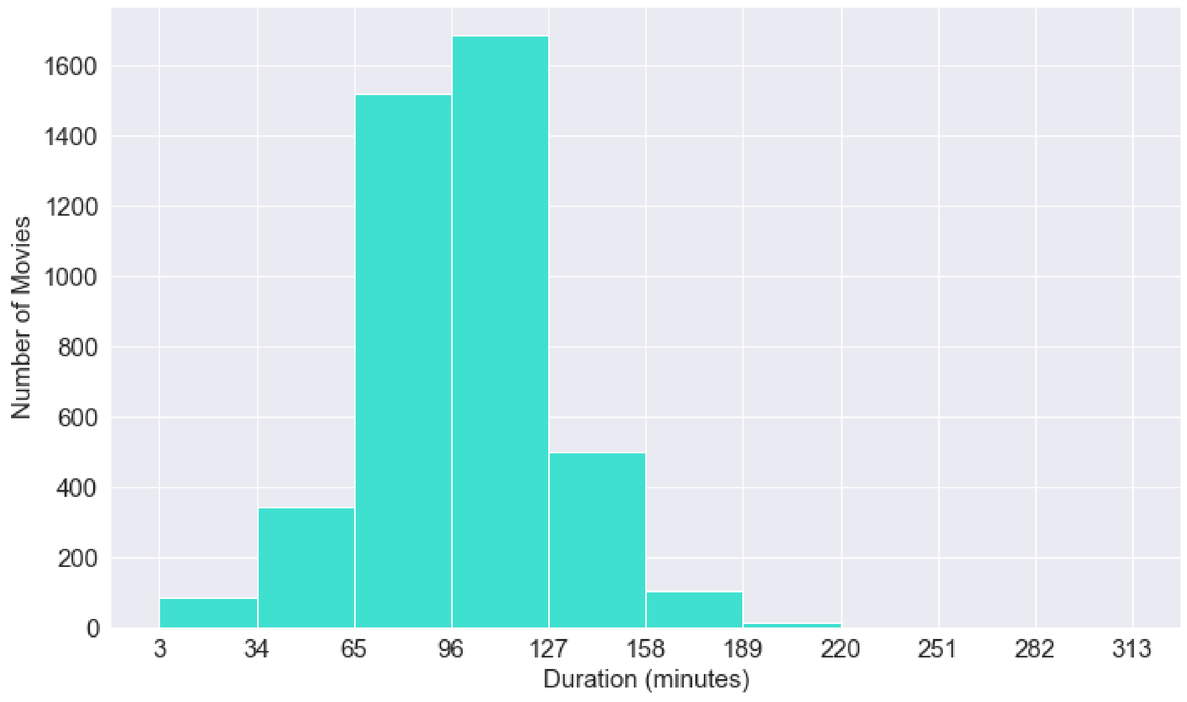
With the following code, we'll take only movies from the dataset and then extract the numeric values from the duration column.

```
#creating column (extract)
df_movie = df_netflix_2019[df_netflix_2019['type']=='Movie']
df_movie = df_movie.assign(minute = df_movie['duration'].str.extract(r'(\d+)', expand=False).asty
```

Now the data is ready to be displayed in a histogram. You can make plots with matplotlib, seaborn or pandas in Python. In this case, I'll do it with matplotlib.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(nrows=1, ncols=1)
plt.hist(df_movie['minute'])
fig.tight_layout()
```

The plot below reveals how the duration of movies is distributed. By observing the plot, we can say that movies in the first bar (3'–34') and the last visible bar (>189') are probably outliers. They might be short films or long documentaries that don't fit well in our movie category (again, it still depends on your project goals)



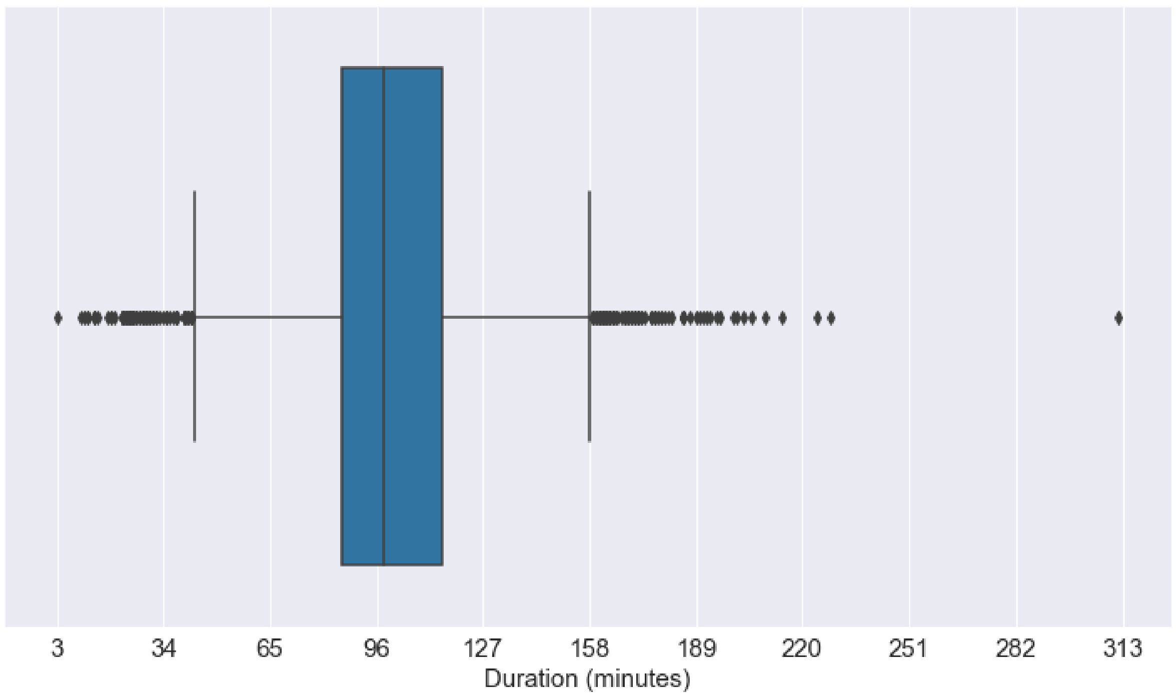
Using boxplots to identify outliers within numeric data

Another option to identify outliers is boxplots. I prefer using boxplots because it leaves outliers out of the box's whiskers. As a result, it's easier to identify the minimum and maximum values without considering the outliers.

We can easily make boxplots with the following code.

```
import seaborn as sns

fig, ax = plt.subplots(nrows=1, ncols=1)
ax = sns.boxplot(x=df_movie['minute'])
fig.tight_layout()
```

The boxplot shows that values below 43' and above 158' are probably outliers.



Also, we can identify some elements of the boxplot like the lower quartile (Q1) and upper quartile (Q3) with the.describe() method.

```
In  [1]: df_movie['minute'].describe()
Out [1]: count    4265.000000
         mean       99.100821
         std        28.074857
         min         3.000000
         25%        86.000000
         50%        98.000000
         75%       115.000000
         max       312.000000
```

In addition to that, you can easily display all elements of the boxplot and even make it interactive with Plotly.

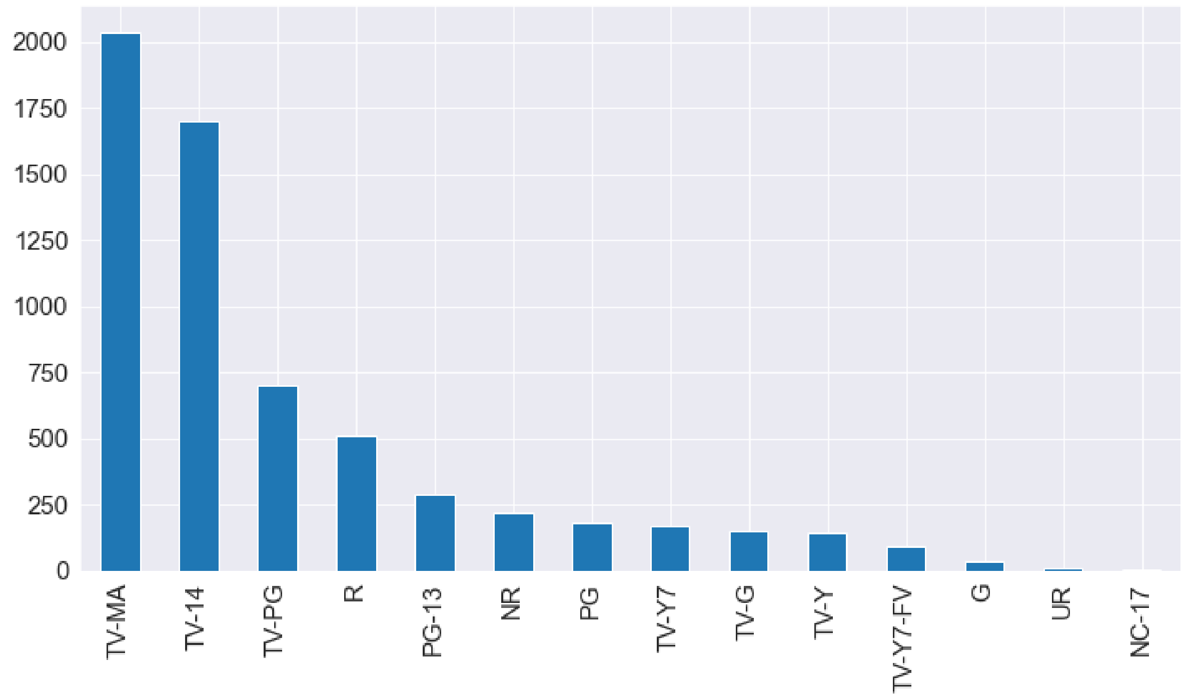
```
import plotly.graph_objects as go
from plotly.offline import iplot, init_notebook_mode

fig = go.Figure()
fig.add_box(x=df_movie['minute'], text=df_movie['minute'])
iplot(fig)
```

Using bars to identify outliers within categorical data

In case the data is categorical, you can identify categories with few observations by plotting bars.

In this case, we'll use the built-in Pandas visualization to make the bar plot.

```
fig=df_netflix_2019['rating'].value_counts().plot.bar().get_figure()
fig.tight_layout()
```



In the plot above, we can see that the mode (the value that appears most often in the column) is 'TV-MA' while 'NC-17' and 'UR' are uncommon.

5. Dealing with Outliers

Once we identified the outliers, we can easily filter them out by using Python's operators.

Using operators & | to filter out outliers

Python operators are simple to memorize. & is the equivalent of and, while| is the equivalent of or.

In this case, we're going to filter out outliers based on the values revealed by the boxplot.

```
#outliers
df_movie[(df_movie['minute']<43) | (df_movie['minute']>158)]
#filtering outliers out
df_movie = df_movie[(df_movie['minute']>43) & (df_movie['minute']
<158)]
```

The df_movie created now contains only movies that last between 43' and 158'.

6. Dealing with Inconsistent Data Before Merging 2 Dataframes

A common task we often come across is merging dataframes to increase the information of an observation. Unfortunately, most of the time, datasets have many inconsistencies because they come from different sources.

From now on, we'll use a second dataset `df_netflix_originals` that contains only Netflix originals, and we'll merge it with the original dataset `df_netflix_2019` to determine original and non-original content.

Dealing with inconsistent column names

A common issue we have to deal with is different column names between tables. Column names can be easily changed with the `.rename` method.

```
df_netflix_originals = pd.read_csv('netflix_originals.csv')
#inconsintent column names
df_netflix_originals.rename(columns={'titles':'title', 'years':'release_year'}, inplace=True)
```

Dealing with inconsistent data type

If you try to merge 2 datasets based on a column that has different data types, Python will throw an error. That's why you have to make sure the type is the same. If the same column have different types, you can use the `.astype` method to normalize it.

```
df_netflix_originals = df_netflix_originals.astype({"release_year": int})
```

Dealing with inconsistent names e.g., "New York" vs. "NY"

Usually, the column and data type normalization is enough to merge to datasets; however, sometimes, there are inconsistencies between the data within the same column caused by data entry errors (typos) or disagreements in the way a word is written.

Movies titles don't usually have these problems. They might have a disagreement in punctuation (we'll take care of this later), but movies usually have a standard name, so to explain how to deal with this problem, I'll create a dataset and a list containing states written in different ways.

There are many libraries that can help us solve this issue. In this case, I'll use the fuzzywuzzy library. This will give a score based on the distance between 2 strings. You can choose the scorer that fits your data better. I will set scorer=fuzz.token_sort_ratio in this example.

```python
from fuzzywuzzy import process, fuzz
states = ['New York', 'California', 'Washington', 'Hawaii']
df_states = pd.DataFrame({'states':['NY', 'CA', 'Washington DC', 'Hawai']})
df_states[['match', 'score']] = df_states['states'].apply(lambda x:process.extractOne(x, states, 
df_states
```

As we can see in the output, the scorer does a good job matching strings.

```
states         match       score
CA             California  33
Hawai          Hawaii      91
NY             New York    40
Washington DC  Washington  87
```

However, keep in mind that it can still match wrong names.

7. Text Normalization

Text normalization is necessary for Natural Language Processing. This involves the following techniques to make text uniform.

- Removing whitespace, punctuation and non-alphanumeric characters
- Tokenization, Stemming, Lemmatization, removing stop words

To make things simple, this article will only cover the first point. However, in the article below, I explain how to tokenize text in Python.

Dealing with inconsistent capitalization

Before merging 2 frames, we have to make sure most rows will match, and normalizing capitalization helps with it.

There are many ways to lower case text within a frame. Below you can see two options (.apply or .str.lower)

```
df_netflix_originals['title'] = df_netflix_originals['title'].apply(lambda x:x.lower())
df_netflix_originals['title'] = df_netflix_originals['title'].str.lower()
```

Remove blank spaces with .strip()

Sometimes data has leading or trailing white spaces. We can get rid of them with the .strip method

```
df_netflix_originals['title'] = df_netflix_originals['title'].apply(lambda x:x.strip())
df_netflix_originals['title'] = df_netflix_originals['title'].str.strip()
```

Remove or replace strings with .replace() or .sub()

Texts between two datasets often have disagreements in punctuation. You can remove it with .apply and .sub or by using .replace

It's good to use any of them with regular expressions. For example, the regex[^\w\s] will help you remove characters other than words (a-z, A-Z, 0–9, _) or spaces.

```
# remove punctuation: clean characters other than word or spaces
df_netflix_originals['title'] = df_netflix_originals['title'].apply(lambda x:re.sub('[^\w\s]','',
df_netflix_originals['title'].replace('[^\w\s]', '', regex=True, inplace=True)
```

Regular expressions (regex) might look intimidating, but they're simpler than you think and are vital when extracting information from text data. In the link below, you'll find a simple guide I made to easily learn regular expressions

8. Merging Datasets

Finally, we can merge the dataset df_netflix_originals and df_netflix_2019. With this, we can identify which movies are Netflix originals and which only belong to the catalog. In this case, we do an outer join to give 'Catalog' value to all the rows with empty values in the "Original" column.

```
df_netflix = pd.merge(df_netflix_originals, df_netflix_2019, on=['title', 'type', 'release_year']
                      how='outer')
df_netflix['original'].fillna('Catalog', inplace=True)
```

Remove duplicates with .drop_duplicates()

One of the pitfalls of outer join with 2 key columns is that we'll obtain duplicated rows if we consider a column alone. In this case, we merged based on the title and release_year columns, so most likely there are titles duplicated that have different release_year.

You can drop duplicates within a column with the .drop_duplicates method

```
#drop_duplicates: data duplicated because of disagreement in releaase_year
df_netflix.drop_duplicates(['title'], keep='first', inplace=True)
```

The data grouped by type and origin is distributed like this.

```
In[1]: df_netflix[['original', 'type']].value_counts()

Out[1]:
original  type
Catalog   Movie      3763
          TV Show    1466
Netflix   TV Show    1009
          Movie       504
```

Thank you for reading