What is Pandas?

Pandas is one of the most popular data science libraries in Python. Easy to use, it is built on top of **NumPy** and shares many functions and properties.

With Pandas, you can read and extract data from files, transform and analyze it, calculate statistics and correlations, and much more!

To start using pandas, we need to import it first:

```python
import pandas as pd
```

pd is a common short name used when importing the library.

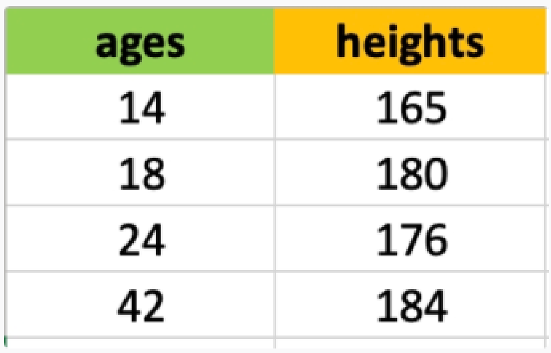
Pandas is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals.

Series & DataFrames

The two primary components of pandas are the **Series** and the **DataFrame**.

A **Series** is essentially a column, and a **DataFrame** is a multi-dimensional table made up of a collection of Series.

For example, the following DataFrame is made of two Series, **ages** and **heights**



ages	heights
14	165
18	180
24	176
42	184

You can think of a **Series** as a one-dimensional array, while a **DataFrame** is a multi-dimensional array.

DataFrames

Before working with real data, let's first create a DataFrame manually to explore its functions.
The easiest way to create a DataFrame is using a **dictionary**:

```python
data = {
   'ages': [14, 18, 24, 42],
   'heights': [165, 180, 176, 184]
}
```

Each key is a column, while the value is an array representing the data for that column.

Now, we can pass this dictionary to the DataFrame constructor:

```python
df = pd.DataFrame(data)
```

Try it Yourself

Run the code to see the resulting DataFrame.

DataFrames

The DataFrame automatically creates a numeric **index** for each row.
We can specify a custom index, when creating the DataFrame:

```
df = pd.DataFrame(data, index=['James', 'Bob', 'Amy',
'Dave'])
```

PY

Try it Yourself

Now we can access a row using its index and the **loc[]** function:

```
print(df.loc["Bob"])
```

PY

Try it Yourself

This will output the row that corresponds to the index "Bob".

! Note, that **loc** uses square brackets to specify the index.

Indexing

We can select a single column by specifying its name in square brackets:

```python
print(df["ages"])
```

PY

Try it Yourself

The result is a **Series** object.

If we want to select multiple columns, we can specify a list of column names:

```python
print(df[["ages", "heights"]])
```

PY

Try it Yourself

This time, the result is a DataFrame, as it includes multiple columns.

! This is useful, when we need to select only a part of the columns from the dataset.

Slicing

Pandas uses the **iloc** function to select data based on its numeric index.
It works the same way indexing lists does in Python.

For example:

```python
# third row
print(df.iloc[2])

#first 3 rows
print(df.iloc[:3])

# rows 2 to 3
print(df.iloc[1:3])
```

PY

Try it Yourself

! **iloc** follows the same rules as slicing does with Python lists.

Conditions

We can also select the data based on a condition.
For example, let's select all rows where **age** is greater than 18 and **height** is greater than 180:

```
df[(df['ages']>18) & (df['heights']>180)]
```

PY

Try it Yourself

Similarly, the or **|** operator can be used to combine conditions.

Reading Data

It is quite common for data to come in a file format. One of the most popular formats is the **CSV** (comma-separated values).
Pandas supports reading data from a CSV file directly into a DataFrame.

For our examples, we will use a CSV file that contains the COVID-19 infection data in California for the year 2020, called '**ca-covid.csv**'.

The **read_csv**() function reads the data of a CSV file into a DataFrame:

```
df = pd.read_csv("ca-covid.csv")
```

PY

We need to provide the file path to the **read_csv()** function.

Pandas also supports reading from JSON files, as well as SQL databases.

Reading Data

Once we have the data in a DataFrame, we can start exploring it.
We can get the first rows of the data using the head() function of the DataFrame:

```
print(df.head())
```

PY

Try it Yourself

```
       date       state  cases  deaths
0  25.01.20  California      1       0
1  26.01.20  California      1       0
2  27.01.20  California      0       0
3  28.01.20  California      0       0
4  29.01.20  California      0       0
```

By default it returns the first 5 rows. You can instruct it to return the number of rows you would like as an argument (for example, **df.head(10)** will return the first 10 rows).

We can see that our DataFrame contains the **date**, **state**, number of **cases** and **deaths** for that date.

Similarly, you can get the last rows using the **tail()** function.

Reading Data

The **info()** function is used to get essential information about your dataset, such as number of rows, columns, data types, etc:

```
df.info()
```

Try it Yourself

Run the code to see the result!

From the result, we can see that our dataset contains 342 rows and 4 columns: date, state, cases, deaths.

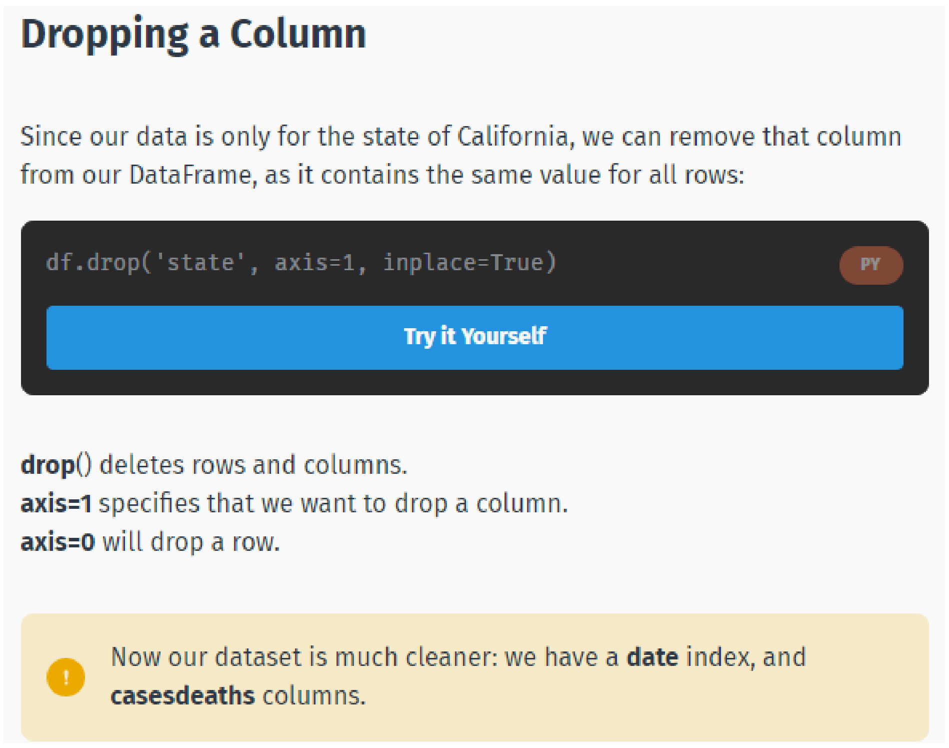
We also see that Pandas has added an auto generated index.
We can set our own index column by using the **set_index**() function:

```
df.set_index("date", inplace=True)
```

Try it Yourself

The **date** column is a good choice for our index, as there is one row for each date.

! The **inplace=True** argument specifies that the change will be applied to our DataFrame, without the need to assign it to a new DataFrame variable.



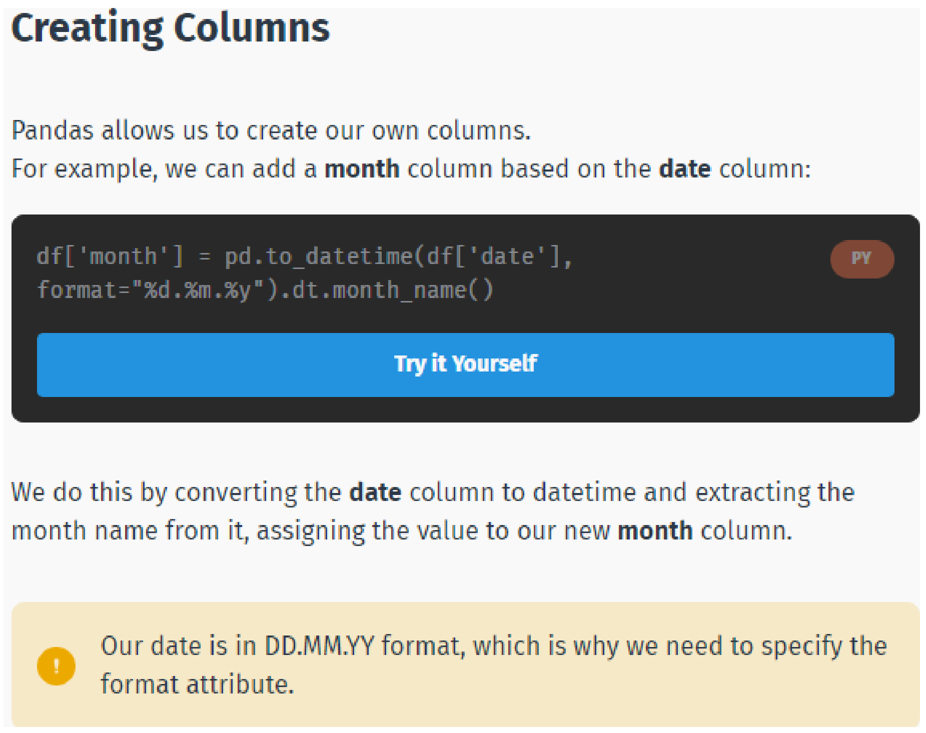
Dropping a Column

Since our data is only for the state of California, we can remove that column from our DataFrame, as it contains the same value for all rows:

```
df.drop('state', axis=1, inplace=True)
```

drop() deletes rows and columns.
axis=1 specifies that we want to drop a column.
axis=0 will drop a row.

Now our dataset is much cleaner: we have a **date** index, and **casesdeaths** columns.



Creating Columns

Pandas allows us to create our own columns.
For example, we can add a **month** column based on the **date** column:

```
df['month'] = pd.to_datetime(df['date'],
format="%d.%m.%y").dt.month_name()
```

We do this by converting the **date** column to datetime and extracting the month name from it, assigning the value to our new **month** column.

Our date is in DD.MM.YY format, which is why we need to specify the format attribute.

Summary Statistics

Now that our dataset is clean and set up, we are ready to look into some stats!
The **describe**() function returns the summary statistics for all the numeric columns:

```python
df.describe()
```

Try it Yourself

This function will show main statistics for the numeric columns, such as **std**, **mean**, **min**, **max** values, etc.

Run the code to see the result!
From the result, we see that the maximum cases that have been recorded in a day is **64987**, while the average daily number of new cases is **6748**.

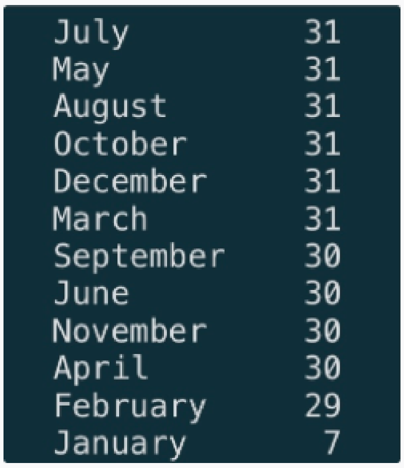
We can also get the summary stats for a single column, for example: **df['cases'].describe()**

Grouping

Since we have a **month** column, we can see how many values each month has, by using the **value_counts**() functions:

```
df['month'].value_counts()
```

Try it Yourself



```
  July          31
  May           31
  August        31
  October       31
  December      31
  March         31
  September     30
  June          30
  November      30
  April         30
  February      29
  January        7
```

We can see that, for example, **January** has only **7** records, while the other months have data for all days.

value_counts() returns how many times a value appears in the dataset, also called the **frequency** of the values.

Grouping

Now we can calculate data insights!
For example, let's determine the number of total infections in each month.
To do this, we need to group our data by the month column and then calculate the sum of the cases column for each month:

```
df.groupby('month')['cases'].sum()
```

PY

Try it Yourself

The **groupby**() function is used to group our dataset by the given column.

We can also calculate the number of total cases in the entire year:

```
df['cases'].sum()
```

PY

Try it Yourself

We can see that California had **2,307,769** infection cases in 2020.

! Similarly, we can use **min**(), **max**(), **mean**(), etc. to find the corresponding values for each group.