

# Essential Guide to Python Pandas

## A Crash Course with Reusable Code Template



```
In [4]: # Create DataFrame from a List of dictionaries
import pandas as pd

list_of_countries = [
    {'country_name': 'China', 'capital_city': 'Beijing', 'population': 1433783686, 'area_km2': 9596961},
    {'country_name': 'New Zealand', 'capital_city': 'Wellington', 'population': 4783063, 'area_km2': 270467},
    {'country_name': 'South Africa', 'capital_city': 'Pretoria', 'population': 58558270, 'area_km2': 1221037},
    {'country_name': 'United Kingdom', 'capital_city': 'London', 'population': 67530172, 'area_km2': 242495},
    {'country_name': 'United States', 'capital_city': 'Washington DC', 'population': 329064917, 'area_km2': 9525067}
]

countries = pd.DataFrame(list_of_countries, index = ['CN', 'NZ', 'ZA', 'GB', 'US'])

countries #.head()
```

Out[4]:

	country_name	capital_city	population	area_km2
CN	China	Beijing	1433783686	9596961
NZ	New Zealand	Wellington	4783063	270467
ZA	South Africa	Pretoria	58558270	1221037
GB	United Kingdom	London	67530172	242495
US	United States	Washington DC	329064917	9525067

Ali Gazala and Janet Zhu

```
In [6]: countries['capital_city'].tolist()
```

```
Out[6]: ['Beijing', 'Wellington', 'Pretoria', 'London', 'Washington DC']
```

```
In [7]: dictionary_of_countries = {'country_name': ['China', 'New Zealand', 'South Africa', 'United Kingdom', 'United States'],
                                   'country_code': ['CN', 'NZ', 'ZA', 'GB', 'US'],
                                   'capital_city': ['Beijing', 'Wellington', 'Pretoria', 'London', 'Washington DC'],
                                   'population': [1433783686, 4783063, 58558270, 67530172, 329064917],
                                   'area_km2': [9596961, 270467, 1221037, 242495, 9525067]}
```

```
countries = pd.DataFrame.from_dict(dictionary_of_countries)
```

# Essential Guide to Python Pandas

A Crash Course with Reusable Code Template in

Jupyter Notebook

---

Ali Gazala and Janet Zhu

<https://naburika.com/>

# Essential Guide to Python Pandas

## A Crash Course with Reusable Code Template in Jupyter Notebook

The Pandas library has emerged as one of the most important data wrangling and processing tools for Python developers and [data professionals](#). It allows users to quickly apply data wrangling tasks such as handling missing data, removing duplicate records, merging multiple datasets, and so on. Therefore, the Pandas library has become a must-have tool in the data science toolkit.

In this series of articles, we provide a crash course to get you started using the Pandas library. The course is designed to be a practical guide with real-life examples of the most common data manipulation tasks.

### Who is this Course for

This course is for aspiring data professionals and Python developers who want to learn how to process data in Pandas. We assume you already have a minimum working knowledge of the Python programming language and are comfortable running data science documents using Jupyter notebook.

For an easy tutorial about how to get up and running with Jupyter notebooks, you can follow [this article](#). Another option is to use [Google Colab](#) to run and save your code within your Google account. For an easy tutorial about how to use the Colab environment, you can watch [this video](#).

To follow the examples in this course, you can copy and paste the code snippets into your Jupyter notebook or Colab environment.

## What Makes Pandas Special

Pandas is an open-source, free (under a [BSD license](#)) Python library originally written by [Wes McKinney](#). It is a high-level data structure and manipulation tool designed to make data analysis and wrangling fast and easy. The library offers a variety of functions and methods to transfer different data sources into a tabular format. In this format, each record is housed on one row and each column contains a unique data type.

Python Developers and Data Professionals can upload data from a variety of data sources such as SQL Databases, API JSON format, CSV files as well as native Python data structures like lists and dictionaries. This flexibility makes Pandas suitable for a wide range of applications. Among them are machine learning modeling, data visualization, and time series forecasting. The versatile structure also makes it easy to integrate pandas with other libraries such as scikit-learn.

Early [releases of Pandas](#) DataFrame dated back to 2011 with Pandas version 1.0 released in January 2020.

## About the Authors

Dr. Ali Gazala is a senior data scientist with a passion for teaching and knowledge sharing. He worked as a lecturer in university to teach many classes related to data science and business intelligence. Ali has more than 10 years of commercial and research experience in demand intelligence, healthcare, and natural language processing. He is a strong advocate of education democratization. Currently, he is focused on developing and delivering online education materials related to artificial intelligence and data science.

Janet Zhu is a senior data analyst with rich experience in linguistics and natural language processing. Her work experience includes many roles in e-commerce and demand intelligence businesses. She has a passion for promoting online education and helping content creators design and develop digital courses.

## Contents

Who is this Course for	2
What Makes Pandas Special	3
<b>How to Import Pandas Library</b>	<b>1</b>
<b>Anatomy of Pandas Data Structures</b>	<b>2</b>
<b>Getting Data into and from Pandas</b>	<b>4</b>
3.1 Python Native Data Structures	5
3.2 Tabular Data Files	8
3.3 API Query and JSON Format	10
3.4 Web Pages Data	11
<b>Describing Information in DataFrames</b>	<b>14</b>
<b>Understanding Data Types</b>	<b>19</b>
<b>Data Cleaning in Pandas</b>	<b>20</b>
6.1 Split Column Values	22
6.2 Replace Strings	23
6.3 Change Column DataType	24
6.4 Drop Rows and Columns	24
6.5 Rename Columns	25
<b>Pandas Merging and Joining Data</b>	<b>28</b>
<b>Data Accessing and Aggregation</b>	<b>38</b>
8.1 Select Data by Row, Column and Index	39
8.2 Filter Data with Conditions	42
8.3 Group and Sort Data	44
<b>Pandas Data Visualization</b>	<b>45</b>
<b>Pandas Analysis Project</b>	<b>50</b>

---

## How to Import Pandas Library

The easiest way to start using Pandas library is to get the Python [Anaconda](#) distribution, a cross-platform distribution for data analysis and scientific computing. The distribution has more than 250 of the most commonly used data science packages and tools such as Pandas, Scikit Learn, Jupyter, and so on. To start using Pandas in your analysis environment, you need to simply run the import Pandas command.

```
# Import Pandas library
import pandas as pd
```

To check your current version of Pandas library, you can run the `__version__` command.

```
# Check Pandas version
pd.__version__
```

Great, you are now ready to start learning Pandas library!

## Anatomy of Pandas Data Structures

The two main Pandas data structure objects are *DataFrames* and *Series*. Pandas DataFrame object is a two-dimensional labeled structure that can hold data in rows and columns, similar to a spreadsheet file or relational database table. Each DataFrame column (also called Pandas *Series* Object) is a one-dimensional labeled structure with a descriptive name and unique data type that applies to all values in that column. *In other words, you can think of a DataFrame as a collection of Series.*

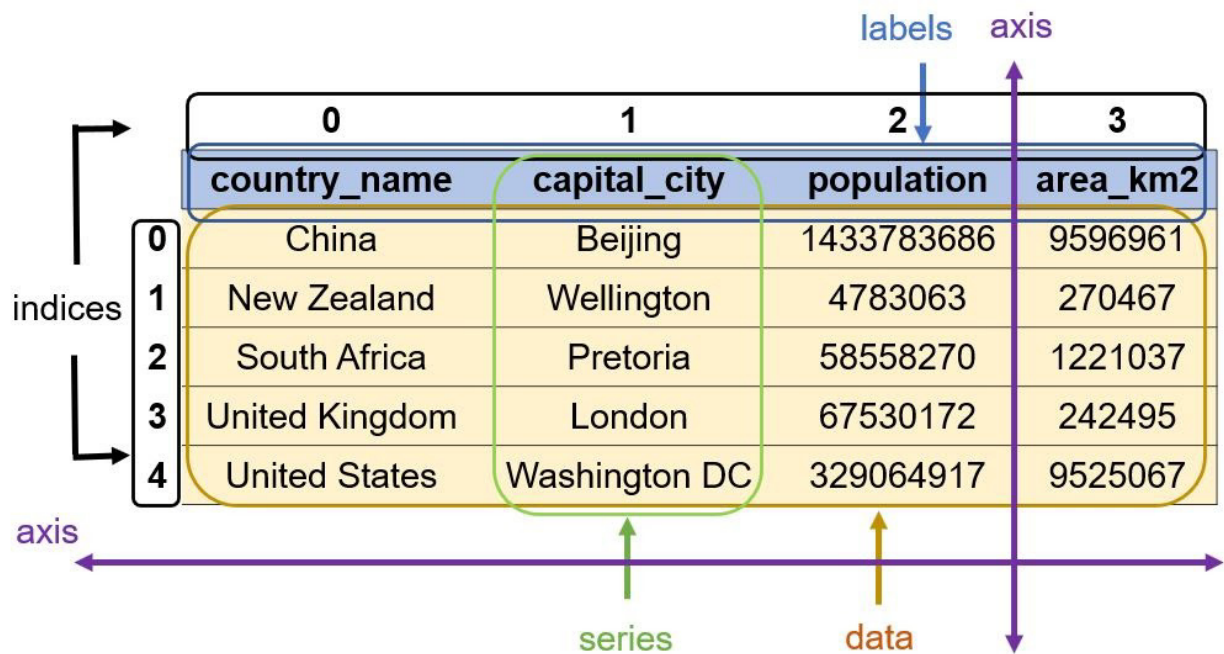
Both DataFrame and Series objects have `index` keys that can be used to reference corresponding values. Index keys are created automatically and can be manipulated by the user to assign specific values as DataFrame or Series index.

In the example below, we see a Pandas DataFrame object about `countries`. The DataFrame consists of four different Series objects (`country_name`, `capital_city`, `population`, `area_km2`) with index values representing each country's ISO code. Later on, you will learn how you can use the DataFrame index to select specific data values.



	country_name	capital_city	population	area_km2
CN	China	Beijing	1433783686	9596961
NZ	New Zealand	Wellington	4783063	270467
ZA	South Africa	Pretoria	58558270	1221037
GB	United Kingdom	London	67530172	242495
US	United States	Washington DC	329064917	9525067

Table 1 below demonstrates the structure of Pandas DataFrame and Series objects.



## Getting Data into and from Pandas

Pandas library is designed to access data from a wide variety of sources and formats. Some popular data sources include tabular files, database tables, third-party APIs and even using Python native data structures. This flexibility is what makes Pandas library useful for many user groups such as developers and data professionals.

To upload data into a Pandas DataFrame, you can utilize a set of `reader` functions such as [`pandas.read\_csv\(\)`](#) to get the data into DataFrame objects. The library also has a set of `writer` functions such as [`pandas.DataFrame.to\_csv\(\)`](#) to allow users to export data frames into external dataset files.

In each function, you can use a set of parameters to pass specific information about your dataset. For instance, in the [`pandas.read\_csv\(\)`](#) and [`pandas.read\_table\(\)`](#) functions, you can use the `sep` parameter to identify the delimiter that separates your data values.

Table 2 below shows a list of available readers and writers functions.

Data Description	Reader	Writer
CSV	read_csv	to_csv
JSON	read_json	to_json
HTML	read_html	to_html
Local clipboard	read_clipboard	to_clipboard
MS Excel	read_excel	to_excel
HDF5 Format	read_hdf	to_hdf
Feather Format	read_feather	to_feather
Parquet Format	read_parquet	to_parquet
Msgpack	read_msgpack	to_msgpack
Stata	read_stata	to_stata
SAS	read_sas	
Python Pickle Format	read_pickle	to_pickle
SQL	read_sql	to_sql
Google Big Query	read_gbq	to_gbq

Source: [Pandas IO Tools](#)

In the following sections, we will learn about some most commonly used methods to get data into and from a Pandas DataFrame.

### 3.1 Python Native Data Structures

The Python programming language has a variety of built-in [data structures](#) such as lists, tuples, dictionaries, strings, and sets. These data structures are ideal for storing data during program execution, however, they can not be efficiently used to perform analytical tasks such as exploratory analysis and data visualization. Pandas library can transfer Python data structures into DataFrame objects to allow users to easily perform data manipulation and analytics.

For example, imagine we have a Python dictionary to save country information such as below:

```
{'country_name': 'New Zealand',  
'capital_city': 'Wellington',  
'country_code': 'NZ',  
'population': 4783063,  
'area_km2': 270467}
```

The dictionary key represents the attribute label or title while the dictionary value represents the corresponding information. Pandas library can convert a list of similar dictionaries into a DataFrame object as shown in the example below:

```
list_of_countries = [  
    {'country_name': 'China', 'capital_city': 'Beijing', 'population': 1433783686, 'area_km2': 9596961},  
    {'country_name': 'New Zealand', 'capital_city': 'Wellington', 'population': 4783063, 'area_km2': 270467},  
    {'country_name': 'South Africa', 'capital_city': 'Pretoria', 'population': 58558270, 'area_km2': 1221037},  
    {'country_name': 'United Kingdom', 'capital_city': 'London', 'population': 67530172, 'area_km2': 242495},  
    {'country_name': 'United States', 'capital_city': 'Washington DC', 'population': 329064917, 'area_km2': 9525067}]  
  
import pandas as pd  
countries = pd.DataFrame(list_of_countries, index = ['CN', 'NZ', 'ZA', 'GB', 'US'])  
  
countries.head()
```

In the code above, we notice the variable `list_of_countries` is defined as a Python list with each element representing a Python dictionary of country information. We import the Pandas library and then we use the built-in `DataFrame` function to transfer the list of countries into a Pandas Dataframe called `countries`. The `pd.DataFrame` is a built-in function to construct `DataFrame` objects from scratch or from native Python data structures.

Notice how we used the `index` parameter to pass a list of country codes as our `DataFrame` index values. We can then examine the new `DataFrame` object using the built-in `head()`

function to return the top rows. We will learn later about the different ways to examine any DataFrame content.

We notice how the dictionary keys were assigned as the DataFrame column names while dictionary values are assigned as the cells. A numerical index with values between 0 to 4 was automatically assigned to the DataFrame object. The user can choose to pass specific index values by using the `index` parameter as shown in the example above.

Another approach is to use a Python dictionary where keys represent column names and values represent Python lists. You can then make use of Pandas [from\\_dict](#) function to transfer the dictionary into a DataFrame object as shown in this example:

```
dictionary_of_countries = {'country_name': ['China', 'New Zealand', 'South Africa', 'United Kingdom', 'United States'],
                           'country_code': ['CN', 'NZ', 'ZA', 'GB', 'US'],
                           'capital_city': ['Beijing', 'Wellington', 'Pretoria', 'London', 'Washington DC'],
                           'population': [1433783686, 4783063, 58558270, 67530172, 329064917],
                           'area_km2': [9596961, 270467, 1221037, 242495, 9525067]}

countries = pd.DataFrame.from_dict(dictionary_of_countries)

countries.head()
```

The above examples demonstrated the flexibility of transforming data stored in Python native data structures into Pandas DataFrame objects.

## 3.2 Tabular Data Files

Tabular data is usually structured into rows and columns and presented in various file formats including CSV, tab-delimited files, fixed-width formats, and spreadsheets. Tabular files can be accessed from the local computer or online.

In this section, we will learn about how to access data from CSV files, Excel Sheet files, and SQL tables. First, we access a CSV file for alcohol consumption by country accessed from the fivethirtyeight [GitHub](#) Repository. To do that, we use the `read_csv()` function and pass the file online location on GitHub. If the CSV file is stored on the local machine, we need to pass the file path.

```
import pandas as pd
alcohol_data =
pd.read_csv('https://raw.githubusercontent.com/fivethirtyeight/data/master/alcohol-consumption/
drinks.csv')

alcohol_data.head()
```

Another commonly used tabular data format is spreadsheets. Pandas library provides the `read_excel()` built-in function to access Microsoft Excel spreadsheet files as shown in the example below:

```
# Return a DataFrame
pd.read_excel("path_to_file/myFile.xls", sheet_name="Sheet1")
```

Notice how the `read_excel()` example makes use of the `sheet_name` parameter to tell the system which sheet name contains the needed dataset. For a complete list of all parameters for each built-in function, check the Pandas official documentation by clicking the function name.

Another common scenario is to query relational database tables using SQL language. Obviously, you would need to provide the necessary credentials and metadata to establish a connection with the database server. You can then apply [pandas.read\\_sql\(\)](#) function to pass the SQL query and load the result into a Pandas DataFrame object.

To simulate this scenario, the following code will create a local database using [the Python SQLite](#) engine. We will then use Pandas to access the data using SQL queries.

```
# Import SQLite library
import sqlite3

# Assign the database name
db_path = r'local_db_example.db'

# Create the database file
conn = sqlite3.connect(db_path)

# Establish a connection with the database file
c = conn.cursor()

# Create a database table
c.execute("""CREATE TABLE mytable
            (id, name, position)""")

# Add some data
c.execute("""INSERT INTO mytable (id, name, position)
            values(1, 'James', 'Data Scientist')""")

c.execute("""INSERT INTO mytable (id, name, position)
            values(2, 'Mary', 'Software Developer')""")

c.execute("""INSERT INTO mytable (id, name, position)
            values(3, 'Max', 'Data Engineer')""")

# Commit changes and close the connection
conn.commit()

c.close()
```

The relational database name `local_db_example.db` should appear as an external file in the same location with your notebook. The database file already includes dummy data describing employee details. The following code queries the data into a Pandas DataFrame object.

```

# Identify the database name
database = "local_db_example.db"

# Establish a connection with the database file
conn = sqlite3.connect(database)

# Use Pandas function to pass SQL query and create a DataFrame object
people = pd.read_sql("select * from mytable", con=conn)

# Print the generated DataFrame
print(people)

# Close the connection
conn.close()

```

In the above example, we created a local database file and used the Pandas library to query the data using SQL, and passed the results into a Pandas DataFrame object. In more practical examples, you may need to query data from relational databases that are stored on remote servers or in the cloud.

### 3.3 API Query and JSON Format

When working on daily tasks, data professionals often need to access data from third-party APIs. This approach is common when the data is continually updated like weather forecasting or when you need to select a small subset of data. API response data usually comes in JSON format which you can think of as a collection of Python data structures like dictionaries and lists represented as text.

For example, we will use API data from [open-notify](#) to get information about the International Space Station ISS. The API gives information about the space station location, altitude, and crew members. The following code will make a query about current crew members onboard ISS. In this example, we will make use of the Python request library to establish a connection with the API.



```

# Import requests library to handle API connection
import requests

# Import and initialize Data pretty printer library
import pprint
pp = pprint.PrettyPrinter(indent=4)

# Pass the API query using requests library
response = requests.get("http://api.open-notify.org/astros.json")
# print(response.status_code)

# Convert response data into JSON format
response_data = response.json()

```

Once we have the API response data, we notice the response includes a list of dictionaries about the astronauts currently aboard the ISS. We can convert this part of the response into a Pandas DataFrame object as shown in the example below:

```

# Examine the response data
pp.pprint(response_data)

# Create a DataFrame of astronauts currently aboard the ISS
astronauts = pd.DataFrame(response_data['people'])
print(astronauts)

```

### 3.4 Web Pages Data

Pandas library offers a built-in function to allow users to parse HTML tables from web pages into a list of Pandas DataFrames. This functionality provides users with a fast way to access data tables embedded in web pages' html code. To demonstrate the process, we will use the Pandas function `read_html()` to parse the [list of countries by population table from Wikipedia](https://en.wikipedia.org/wiki/List_of_countries_by_population_(United_Nations)) into a DataFrame object as shown in the example below:

```

web_data =
pd.read_html('https://en.wikipedia.org/wiki/List_of_countries_by_population_(United_Nations)')

type(web_data)

```

When examining the type of `web_data` variable, we notice the `read_html()` function has returned a list of five elements representing the table tags detected in the webpage HTML code. Each table tag was automatically converted into a Pandas DataFrame object.

However, not all tables are useful as they may contain unwanted HTML data. Therefore, we must carefully examine the returned list and identify the useful DataFrame objects.

In this example, we notice the fourth item `web_data[3]` contains the needed `countries` table. Therefore, we can assign the fourth item to a new variable for easier use as shown in the code below:

```
web_countries_table = web_data[3]
web_countries_table.head()
```

	Country/Area	UN Statistical subregion [4]	UN continental region [4]	Population (1 July 2018)	Population (1 July 2019)	Change
0	China [a]	Eastern Asia	Asia	1427647786	1433783686	0.0043
1	India	Southern Asia	Asia	1352642280	1366417754	0.0102
2	United States	Northern America	Americas	327096265	329064917	0.006
3	Indonesia	South-eastern Asia	Asia	267670543	270625568	0.011
4	Pakistan	Southern Asia	Asia	212228286	216565318	0.0204

By examining the first few rows of our DataFrame, we notice the parsed table includes some unwanted strings such as brackets `[]` and parentheses `()` in country names and column titles.

Also, we notice the column `Change` includes both real numbers and mathematical symbols which would transfer the column into a text datatype (we will learn more about Pandas

data types soon). These issues are normal for data accessed from web pages and may not be readily available for data analysis.

Luckily, Pandas library includes some other tools and functions that would help the user to clean up the data for analysis. The use of the [`read\_html\(\)`](#) function would save time from web data using more traditional web scraping libraries such as `Beautiful Soup`. In later examples, we will learn more about how to convert similar tables from web pages into DataFrames ready for data analysis.

In addition to the above scenarios for creating DataFrame objects using `reader` functions, the Pandas library also provides a set of `writer` functions to save DataFrame objects as external datasets.

## Describing Information in DataFrames

In the previous section, we learned how to use Pandas `reader` and `writer` functions to create `DataFrame` objects from different data sources. Once the data is uploaded, users can make use of several built-in attributes designed to examine the states of the `DataFrames`. These attributes can provide users with information about the `DataFrame` size, data types, missing values, in addition to basic summary statistics.

This information is important for users to identify what changes they would need to make to prepare the `DataFrame` object for further analysis.

To demonstrate commands, we will use Pandas `reader` functions to import some publicly available datasets from GitHub.

```
import pandas as pd
alcohol_data =
pd.read_csv('https://raw.githubusercontent.com/fivethirtyeight/data/master/alcohol-consumption/
drinks.csv')
```

One of the first questions to answer when working with a new dataset is to know the size of the data, i.e., how many rows and columns we have.

To answer this question, we can use the `shape` attribute which returns a Python tuple representing the dimensionality of `DataFrame` objects. The first value represents the number of records while the second value counts the number of columns.

```
# Check the dimension of alcohol_data DataFrame
Alcohol_data.shape
```

Another commonly used attribute is `size`, which can be used to identify how many elements we have in a given DataFrame or Series object (including missing values).

```
# How many elements in alcohol_data DataFrame
Alcohol_data.size

# How many elements in alcohol_data['country'] Series
Alcohol_data['country'].size
```

Once we have learned about the size of our DataFrame and Series objects, we can learn more details using the `info()` attribute. It provides a useful summary of the DataFrame columns as shown in the example below:

```
alcohol_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 193 entries, 0 to 192
Data columns (total 5 columns):
 #   Column                                Non-Null Count  Dtype
---  -
 0   country                              193 non-null    object
 1   beer_servings                        193 non-null    int64
 2   spirit_servings                       193 non-null    int64
 3   wine_servings                        193 non-null    int64
 4   total_litres_of_pure_alcohol         193 non-null    float64
dtypes: float64(1), int64(3), object(1)
memory usage: 7.7+ KB
```

The results first highlight the number of records in the DataFrame and the range of the numerical index value automatically assigned to this DataFrame. It shows the total number of columns (5 columns in our dataset). Next, it lists the column names with their respective data types and how many values of that column contain an empty or null value.

In this dataset, it seems we don't have any missing values since the number of records is equal to the number of non-null counts. We notice the data types for the `country` column

is Pandas objects which represent text values, while three servings columns ('beer\_servings', 'spirit\_servings', and 'wine\_servings') have the int64 data type which represents integer numbers, and total litres column assigned float64 data type which allows real numbers.

At this stage, we have an idea about what changes we need to make in order to have the correct data types. For example, numerical data types such as int64 allow us to apply mathematical calculations on the values while object data type allows us to apply text formatting functions. In the next section about data cleaning, we will learn how to change data types.

Finally, the function displays data about how many columns there are for each data type and the memory size of this DataFrame (the memory size info can be useful when working with a large DataFrame and you may wish to optimize the DataFrame size).

The other exploratory attribute `describe()` will return basic statistical analysis of the DataFrame numeric columns as shown in this example below:

```
alcohol_data.describe()
```

	beer_servings	spirit_servings	wine_servings	total_litres_of_pure_alcohol
count	193	193	193	193
mean	106.1606218	80.99481865	49.4507772	4.717098446
std	101.1431025	88.28431211	79.69759846	3.773298164
min	0	0	0	0
25%	20	4	1	1.3
50%	76	56	8	4.2
75%	188	128	59	7.2
max	376	438	370	14.4

From the example above, we notice the `describe()` function was only applied to the numerical columns and the country name column was ignored. This is because descriptive statistics are based on numerical columns only to summarize the central tendency, dispersion, and shape of a dataset's distribution, excluding `NaN` values.

In addition to the numerical statistical summary, you can also explore the features of text values in DataFrames. For this exercise, we will use the [country codes dataset](#) from the [Open Data GitHub repository](#). The data include many details about each country's international codes and geographic regions.

```
countries_data =
pd.read_csv('https://raw.githubusercontent.com/datasets/country-codes/master/data/country-codes
.csv')
countries_data.head()
```

For instance, the column `Region Name` appears to be a text column that holds the geographical region of each country. In order to find the number of individual region values we can apply functions like `unique()`, and `value_counts()` on Pandas series values like below:

```
countries_data['Region Name'].unique()
```

The output would be:

```
array([nan, 'Asia', 'Europe', 'Africa', 'Oceania', 'Americas'], dtype=object)
```

```
countries_data['Region Name'].value_counts()
```

The output would be:

```
Africa      60
Americas    57
Europe      52
Asia        50
Oceania     29
Name: Region Name, dtype: int64
```



## Understanding Data Types

It is important to assign the correct [data type](#) for each DataFrame column in order to avoid any problems for data analysis. Pandas will try to infer the correct data type for each column. For a list of Pandas data type mapping, please refer to [this link](#).

However, sometimes you need to change the data type manually. Selecting the correct data type will allow you to perform further analysis such as mathematical analysis on `numeric` columns and text formatting on `object` data types. The following table presents common Pandas data types:

Pandas dtype	Usage
object	Text or mixed numeric and non-numeric values
int64	Integer numbers
float64	Floating point numbers
bool	True/False values
datetime64	Date and time values
timedelta[ns]	Differences between two datetimes
category	Finite list of text values

We will learn how to change the Pandas data types in the following part of the handbook.

## Data Cleaning in Pandas

Previously, we learned how to use the different Pandas tools and functions to get external datasets into a DataFrame object.

Many real-life datasets come with problems such as missing values, wrong datatype, and bad formatting. Data professionals usually need to spend lots of time correcting these issues before the dataset becomes ready for analysis. Luckily, Pandas library comes with a set of built-in functions to help users fix these issues.

In this section, we will learn how to use Pandas to identify and correct some common data quality issues.

To demonstrate the process, we will use a toy DataFrame about `countries`. Each country has different pieces of information such as name, population, size, and independence date as shown in the code below:

```
list_of_countries = [
{'Country Name':'China','ISO Code':'CN','Country Population':1433783686,'Country Area km2
(mi2)':9,596,961 (3,705,407)','Independence Day':'1 October 1949'},
{'Country Name':'New Zealand','ISO Code':'NZ','Country Population':4783063,'Country Area km2
(mi2)':270,467 (104,428)','Independence Day':'26 September 1907'},
{'Country Name':'South Africa','ISO Code':'ZA','Country Population':58558270,'Country Area km2
(mi2)':1,221,037 (471,445)','Independence Day':'31 May 1910'},
{'Country Name':'Australia','ISO Code':'AU','Country Population':25763300,'Country Area km2
(mi2)':7,692,024 (2,969,907)', 'Independence Day':'1 January 1901'},
{'Country Name':'United States','ISO Code':'US','Country Population':329064917,'Country Area
km2 (mi2)':9,525,067 (3,677,649)','Independence Day':'4 July 1776'},
{'Country Name':'New Zealand','ISO Code':'NZ','Country Population':4783063,'Country Area km2
(mi2)':270,467 (104,428)','Independence Day':'26 September 1907'}]

countries = pd.DataFrame(list_of_countries)

countries
```

	Country Name	ISO Code	Country Population	Country Area km2 (mi2)	Independence Day
0	China	CN	1433783686	9,596,961 (3,705,407)	1 October 1949
1	New Zealand	NZ	4783063	270,467 (104,428)	26 September 1907
2	South Africa	ZA	58558270	1,221,037 (471,445)	31 May 1910
3	Australia	AU	25763300	7,692,024 (2,969,907)	1 January 1901
4	United States	US	329064917	9,525,067 (3,677,649)	4 July 1776
5	New Zealand	NZ	4783063	270,467 (104,428)	26 September 1907

Looking at our DataFrame, we notice the information about the country New Zealand was repeated twice in rows 1 and 5. Also, we notice the Country Area has values in both square kilometers and square miles.

```
countries.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 5 columns):
#   Column                Non-Null Count  Dtype
---  -
0   Country Name          6 non-null     object
1   ISO Code              6 non-null     object
2   Country Population     6 non-null     int64
3   Country Area km2 (mi2) 6 non-null     object
4   Independence Day       6 non-null     object
dtypes: int64(1), object(4)
memory usage: 368.0+ bytes
```

Examining the DataFrame using the `info()` attribute shows that both `country area` and `independence day` columns were assigned as text datatypes.

In order to clean up the data for further analysis, we need to perform the following steps:

- Split the Country Area values into two columns for both kilometer values and square miles
- Remove any non-numeric characters from the area values
- Change the country area and independence date columns to the correct data types
- Drop unwanted rows and columns
- Rename all the columns to have lower case letters separated by underscores

## 6.1 Split Column Values


The country area column is represented in both `square kilometers` and `square miles`. The square miles values are included within parentheses () and there is an empty space between the square kilometers and square miles values. Therefore, we can use the Pandas built-in `split()` function to separate these values into new different columns as shown in the following code:

```
countries[['Area km2','Area mi2']] = countries['Country Area km2 (mi2)'].str.split(' ', expand = True)
```

The first part of the code `countries[['Area km2','Area mi2']]` on the left side of the equal “=” sign creates two new DataFrame series. The `split()` function was used on the `Country Area km2 (mi2)` Series to separate the numerical values into square km and mile respectively. Note that we needed to tell the function to use the quotes “” with

space inside as the splitting point and to send each of the generated values into a new column using the `expand` parameter.

The [`split\(\)`](#) is one of the [many built-in string formatting methods](#) that can be applied to the Pandas series using the format `Series.str.<function/property>`. The following figure demonstrates the result of using the [`split\(\)`](#) function in our DataFrame:



Country Area km2 (mi2)
9,596,961 (3,705,407)
270,467 (104,428)
1,221,037 (471,445)
7,692,024 (2,969,907)
9,525,067 (3,677,649)
270,467 (104,428)

Area km2	Area mi2
9,596,961	(3,705,407)
270,467	(104,428)
1,221,037	(471,445)
7,692,024	(2,969,907)
9,525,067	(3,677,649)
270,467	(104,428)

## 6.2 Replace Strings

After we split the country area into two separate columns for `square kilometers` and `square miles`, we need to continue our work to convert these values into numeric format by removing any non-numeric characters such as parentheses and commas from the newly created columns `Area km2` and `Area mi2`.

To do that, we can use the built-in [`replace\(\)`](#) function to replace occurrences of specific patterns in a given series with some other string. The function will take the first parameter as the targeted string or regular expression pattern, and the second parameter as the replacement value. The following code demonstrates how we replace any non-numeric values using the regular expression `(\D+)` within the Pandas [`replace\(\)`](#) function.

```
countries['Area km2'] = countries['Area km2'].str.replace('(\D+)', '')
countries['Area mi2'] = countries['Area mi2'].str.replace('(\D+)', '')
```

Area km2	Area mi2		Area km2	Area mi2
9,596,961	(3,705,407)		9596961	3705407
270,467	(104,428)		270467	104428
1,221,037	(471,445)		1221037	471445
7,692,024	(2,969,907)		7692024	2969907
9,525,067	(3,677,649)		9525067	3677649
270,467	(104,428)		270467	104428

## 6.3 Change Column DataType

Once we separated and cleaned up the `country area` into the proper format, we can move forward to assign the `country area` and `independent day` columns to the correct data types. To do that, we will make use of Pandas [astype\(\)](#) function to pass a Python dictionary representing the name of each column and the corresponding data type as shown in this code below:

```
countries = countries.astype({'Area km2': 'int64', 'Area mi2': 'int64', 'Independence Day': 'datetime64'})
```

## 6.4 Drop Rows and Columns

Any data processing task would often include removing duplicate records or unwanted columns. In our `countries` DataFrame, we notice such cases as repeated rows for `New`

Zealand. As we already split the `country area` into two new columns, there is no need to keep the old country area column.

To remove unnecessary rows, we can make use of the Pandas [drop\(\)](#) function to remove rows or columns by specifying label names and corresponding axes. The axis parameter can take two values:

- 0: to indicate the action will be taken at the row-level or
- 1: to indicate the action will be taken at the column-level

The following code demonstrates how to use the [drop\(\)](#) function to remove unwanted rows and columns:

```
# To remove the old country area column
countries.drop('Country Area km2 (mi2)', axis = 1, inplace = True)

# To remove duplicate row for New Zealand
countries.drop(5, axis = 0, inplace = True)
```

If you have several duplicate rows, we can also make use of the Pandas [drop\\_duplicates\(\)](#) function to keep only the unique rows in the DataFrame. The following code would achieve the same result as the second line of code above.

```
countries.drop_duplicates(inplace = True)
```

## 6.5 Rename Columns

Another common data processing task is to standardize any DataFrame column names by using lower case letters separated by underscores. To achieve this, we can make use of the

Pandas [rename\(\)](#) function by passing a dictionary with current and new column names as shown in the code below:

```
countries.rename(columns={'Country Name': 'country_name', 'ISO Code': 'country_code', 'Country Population': 'country_population', 'Country Date of Independence': 'independence_date', 'Area km2': 'area_km2', 'Area mi2': 'area_mi2'}, inplace=True)
```

Then, we can run the [info\(\)](#) method to print a concise summary of the DataFrame and examine all the changes applied to this tutorial. We can compare the changes below:

```
countries.info()
```

### Before

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 5 columns):
#   Column                Non-Null Count  Dtype
---  ---                ---
0   Country Name          6 non-null     object
1   ISO Code              6 non-null     object
2   Country Population    6 non-null     int64
3   Country Area km2 (mi2) 6 non-null     object
4   Independence Day      6 non-null     object
dtypes: int64(1), object(4)
memory usage: 368.0+ bytes
```

### After

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 0 to 4
Data columns (total 6 columns):
#   Column                Non-Null Count  Dtype
---  ---                ---
0   country_name          5 non-null     object
1   country_code          5 non-null     object
2   country_population    5 non-null     int64
3   Independence Day      5 non-null     datetime64[ns]
4   area_km2              5 non-null     int64
5   area_mi2              5 non-null     int64
dtypes: datetime64[ns](1), int64(3), object(2)
memory usage: 280.0+ bytes
```

The output shows that we added two new columns for country areas and changed the data types to support the needed information in each column. There are only five countries instead of six so no more duplicate rows exist. The final DataFrame looks like this:



	country_name	country_code	country_population	independence_date	area_km2	area_mi2
0	China	CN	1433783686	1949-10-01	9596961	3705407
1	New Zealand	NZ	4783063	1907-09-26	270467	104428
2	South Africa	ZA	58558270	1910-05-31	1221037	471445
3	Australia	AU	25763300	1901-01-01	7692024	2969907
4	United States	US	329064917	1776-07-04	9525067	3677649

## Pandas Merging and Joining Data

Data professionals often need to combine different data sources for analysis projects. For example, if you are working on a data science project to analyze how sports games impact food and beverage sales, you may need to collect several datasets such as game timetables, sports teams' performance, sports venues, and capacity, as well as sales figures for multiple vendors.

If you are using the Pandas library for your project, it's likely that each dataset is stored in a separate DataFrame. Luckily, the library provides a set of tools that allow us to merge and join multiple DataFrames to create large datasets for analysis.

In this section, we will learn the two most common ways to combine DataFrames in the Pandas library:

`pd.concat([DataFrame1, DataFrame2])`: Simple combining two or more Pandas dataframes in a column-wise or row-wise approach.

`pd.merge([DataFrame1, DataFrame2])`: Complex column-wise combining of Pandas dataframes in a SQL-like way.

The `concat()` function is used to add together one or more DataFrames. To demonstrate how it works, we will use the function to combine multiple toy DataFrames about popular sports tournaments like FIFA Soccer World Cup and Rugby World Cup. Each dataset has different pieces of information such as winning team, host country, attendance size as shown in the code below:

```
# FIFA World Cup Winning Teams
df_fifa_world_cup_winners = pd.DataFrame({'year': [2018,2014,2010,2006,2002,1998],
      'winner': ['France','Germany','Spain','Italy','Brazil','France'],
      'host_country': ['Russia','Brazil','South Africa',
      'Germany','South Korea','Japan']})

df_fifa_world_cup_winners
```

	year	winner	host_country
0	2018	France	Russia
1	2014	Germany	Brazil
2	2010	Spain	South Africa
3	2006	Italy	Germany
4	2002	Brazil	South Korea
5	1998	France	Japan

```
# Rugby World Cup Winning Teams
df_rugby_world_cup_winners = pd.DataFrame({'year': [1999,2003,2007,2011,2015,2019],
      'winner': ['Australia','England','South Africa','New Zealand','New Zealand','South Africa'],
      'host_country': ['Wales','Australia','France','New Zealand','England','Japan'],
      'venue':['Millennium Stadium','Telstra Stadium','Stade de France','Eden
      Park','Twickenham','Nissan Stadium'],
      'attendance':[72500,82957,80430,61079,80125,70103]})

df_rugby_world_cup_winners
```

	year	winner	host_country	venue	attendance
0	1999	Australia	Wales	Millennium Stadium	72500
1	2003	England	Australia	Telstra Stadium	82957
2	2007	South Africa	France	Stade de France	80430
3	2011	New Zealand	New Zealand	Eden Park	61079
4	2015	New Zealand	England	Twickenham	80125
5	2019	South Africa	Japan	Nissan Stadium	70103

We first noticed the DataFrames above have some common information such as the `year` of the event, the `winner` team name, and the `host_country`. However, the Rugby World Cup dataset has two extra columns: `venue` and `attendance`.

Let's try to create a large dataset with all winning FIFA and Rugby world cup teams. The code below demonstrates how to use all the common column names to stack the two DataFrames on top of each other.

```
df_teams = pd.concat([df_fifa_world_cup_winners[['year', 'winner', 'host_country']],
                     df_rugby_world_cup_winners[['year', 'winner', 'host_country']]])

df_teams
```

	year	winner	host_country
0	2018	France	Russia
1	2014	Germany	Brazil
2	2010	Spain	South Africa
3	2006	Italy	Germany
4	2002	Brazil	South Korea
5	1998	France	Japan
0	1999	Australia	Wales
1	2003	England	Australia
2	2007	South Africa	France
3	2011	New Zealand	New Zealand
4	2015	New Zealand	England
5	2019	South Africa	Japan

We created a new DataFrame object called `df_teams` with 12 records from the two parent datasets. However, the resulting DataFrame raises some issues.

First, it becomes impossible to identify if a given team was part of the original Rugby or Soccer datasets. Second, the new DataFrame object inherits the original index values from the parent datasets. This behavior can be controlled by adjusting the `concat()` function parameters. The `keys` parameter can be used to track the data source by adding extra index values to the new DataFrame as shown in the example below. This feature would allow us to query and access specific subsets of the DataFrame using the newly assigned index value.

```
# Add data source index values to the new DataFrame
df_teams = pd.concat([df_fifa_world_cup_winners[['year', 'winner', 'host_country']],
                     df_rugby_world_cup_winners[['year', 'winner', 'host_country']]], keys =
                     ['soccer', 'rugby'])

df_teams
```

		year	winner	host_country
soccer	0	2018	France	Russia
	1	2014	Germany	Brazil
	2	2010	Spain	South Africa
	3	2006	Italy	Germany
	4	2002	Brazil	South Korea
	5	1998	France	Japan
rugby	0	1999	Australia	Wales
	1	2003	England	Australia
	2	2007	South Africa	France
	3	2011	New Zealand	New Zealand
	4	2015	New Zealand	England
	5	2019	South Africa	Japan

In another scenario, we may prefer the new DataFrame to have totally new index values. This option can be achieved by setting the `ignore_index` parameter to `true` as shown in the code below:

```
# Ignore old index values in the new DataFrame
df_teams = pd.concat([df_fifa_world_cup_winners[['year', 'winner', 'host_country']],
                     df_rugby_world_cup_winners[['year', 'winner', 'host_country']]],
                     ignore_index = True)

df_teams
```

	year	winner	host_country
0	2018	France	Russia
1	2014	Germany	Brazil
2	2010	Spain	South Africa
3	2006	Italy	Germany
4	2002	Brazil	South Korea
5	1998	France	Japan
6	1999	Australia	Wales
7	2003	England	Australia
8	2007	South Africa	France
9	2011	New Zealand	New Zealand
10	2015	New Zealand	England
11	2019	South Africa	Japan

The `concat()` function also allows us to combine multiple datasets even with little or no common values among them. The newly generated dataset will include all columns from the original DataFrame, with missing values replaced with `null` or `NaN` as shown in the example below:

```
# The new DataFrame will include all original columns
df_teams = pd.concat([df_fifa_world_cup_winners,
                      df_rugby_world_cup_winners])

df_teams
```

	year	winner	host_country	venue	attendance
0	2018	France	Russia	NaN	NaN
1	2014	Germany	Brazil	NaN	NaN
2	2010	Spain	South Africa	NaN	NaN
3	2006	Italy	Germany	NaN	NaN
4	2002	Brazil	South Korea	NaN	NaN
5	1998	France	Japan	NaN	NaN
0	1999	Australia	Wales	Millennium Stadium	72500
1	2003	England	Australia	Telstra Stadium	82957
2	2007	South Africa	France	Stade de France	80430
3	2011	New Zealand	New Zealand	Eden Park	61079
4	2015	New Zealand	England	Twickenham	80125
5	2019	South Africa	Japan	Nissan Stadium	70103

The examples above demonstrate how the Pandas [concat\(\)](#) function can create new datasets by adding DataFrame objects on top of each other (row axis). The function also provides the possibility to add the DataFrames sideways (column axis). This option is controlled using the `axis` parameter when it's set to 0 or 1 as shown in the example below:

```
# The new DataFrame will include all original columns aligned horizontally
df_teams = pd.concat([df_fifa_world_cup_winners,
                      df_rugby_world_cup_winners], axis = 1)

df_teams
```

	year	winner	host_country	year	winner	host_country	venue	attendance
0	2018	France	Russia	1999	Australia	Wales	Millennium Stadium	72500
1	2014	Germany	Brazil	2003	England	Australia	Telstra Stadium	82957
2	2010	Spain	South Africa	2007	South Africa	France	Stade de France	80430
3	2006	Italy	Germany	2011	New Zealand	New Zealand	Eden Park	61079
4	2002	Brazil	South Korea	2015	New Zealand	England	Twickenham	80125
5	1998	France	Japan	2019	South Africa	Japan	Nissan Stadium	70103

Pandas [merge\(\)](#) provides the functionality to join DataFrame and Series objects in a way similar to relational database operations. Users who are familiar with merging datasets using SQL but new to Pandas might be interested in [this comparison](#). In this set of examples, we will demonstrate the use of the [merge\(\)](#) function and highlight the use of some important parameters.

The below code will create two DataFrame objects with one similar and four different columns, namely, `key`, `column_A`, `column_B`, `column_C`, and `column_D`. The `key` column includes some similar values that appear on both DataFrames as well as uncommon ones.

```
left = pd.DataFrame(
{
    "key": ["K0", "K1", "K2", "K3", "K4", "K5"],
    "column_A": ["A0", "A1", "A2", "A3", "A4", "A5"],
    "column_B": ["B0", "B1", "B2", "B3", "B4", "B5"]})

right = pd.DataFrame(
{
    "key": ["K0", "K1", "K2", "K3", "K6"],
    "column_C": ["C0", "C1", "C2", "C3", "C6"],
    "column_D": ["D0", "D1", "D2", "D3", "D6"]})
```



To join the two DataFrame objects using the `merge()` function, we can use a set of parameters to identify common values, joining type, and source object.

`on`: This parameter can be used if the common column has the same name in both DataFrames

`left on`: Identify the joining column in the left DataFrame

`right on`: identify the joining column in the right DataFrame

`indicator`: Add an extra column to the joined DataFrame to show the source of each column

`how`: Identify the joining type as one of four possible options [inner, left, right, outer]

The following code will merge the two tables using the key column and the default inner joining method. We notice that only four records we selected represent the key values that appear in both DataFrames [k0, k1, k2, k3].

```
# Merge the two DataFrames using the common column
result = pd.merge(left, right, on="key", how = 'inner', indicator = True)

result
```

	key	column_A	column_B	column_C	column_D	_merge
0	K0	A0	B0	C0	D0	both
1	K1	A1	B1	C1	D1	both
2	K2	A2	B2	C2	D2	both
3	K3	A3	B3	C3	D3	both

By changing the `how` parameter to `outer` value, we notice the joined DataFrame includes all records from both original DataFrames as shown in the example below:

```
# Merge the two DataFrames using the common column
```

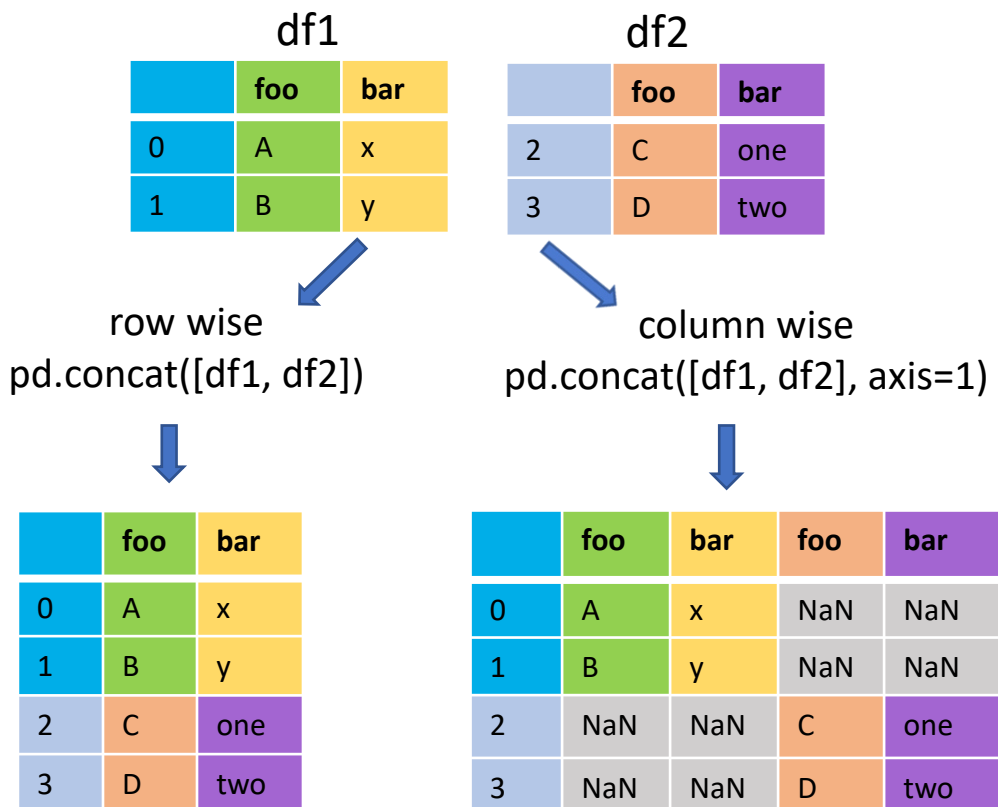
```
result = pd.merge(left, right, on="key", how = 'outer', indicator = True)

result
```

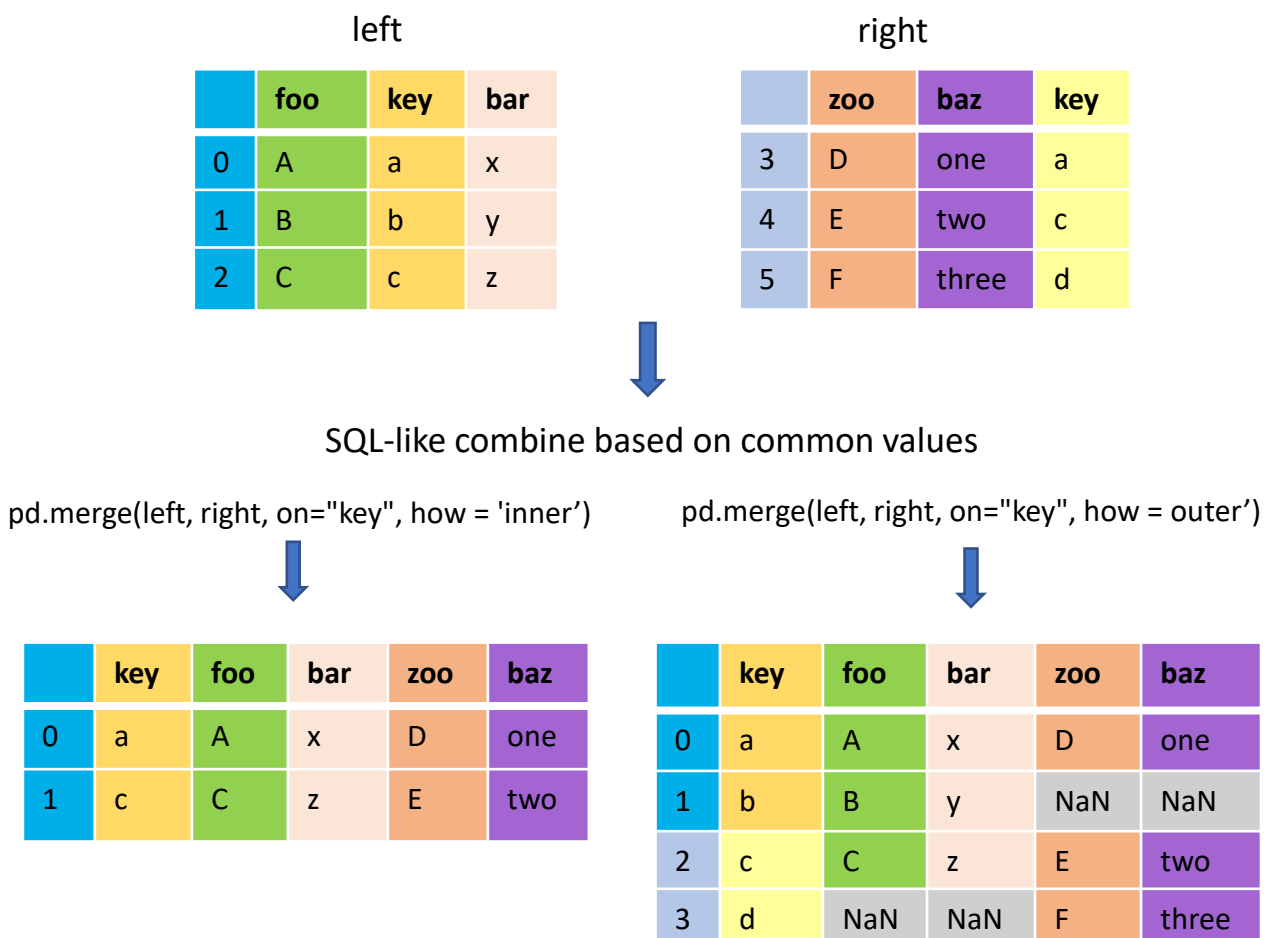
	key	column_A	column_B	column_C	column_D	_merge
0	K0	A0	B0	C0	D0	both
1	K1	A1	B1	C1	D1	both
2	K2	A2	B2	C2	D2	both
3	K3	A3	B3	C3	D3	both
4	K4	A4	B4	NaN	NaN	left_only
5	K5	A5	B5	NaN	NaN	left_only
6	K6	NaN	NaN	C6	D6	right_only

# two common ways to combine DataFrames

## CONCAT



## MERGE



## Data Accessing and Aggregation

So far in this course, you have learned about Pandas different techniques to process and get data ready for analysis. In this section of the course, we will learn about how to explore your data by performing the following tasks:

- Selecting data by row, column and index values
- Filtering data with conditions
- Aggregating and sorting data

To demonstrate these tasks, we will use the following toy DataFrame about different countries' information. The DataFrame has an index value representing each country's ISO code, regions, population size, and most common language in each country.

```
# Countries Information DataFrame
df_countries_info = pd.DataFrame({'country_name': ['Egypt', 'Kenya', 'Morocco', 'Nigeria', 'South Africa', 'Brazil', 'Canada', 'Chile', 'Mexico', 'United States', 'China', 'India', 'Indonesia', 'Japan', 'Vietnam', 'Austria', 'Belgium', 'France', 'Italy', 'United Kingdom', 'Australia', 'Fiji', 'New Zealand', 'Tonga', 'Tuvalu'],

'region': ['Africa', 'Africa', 'Africa', 'Africa', 'Africa', 'South Americas', 'North Americas', 'South Americas', 'North Americas', 'North Americas', 'Asia', 'Asia', 'Asia', 'Asia', 'Asia', 'Europe', 'Europe', 'Europe', 'Europe', 'Oceania', 'Oceania', 'Oceania', 'Oceania'],

'population': [100388073, 52573973, 36471769, 200963599, 58558270, 211049527, 37411047, 18952038, 127575529, 329064917, 1433783686, 1366417754, 270625568, 126860301, 96462106, 8955102, 11539328, 65129728, 60550075, 67530172, 25203198, 889953, 4783063, 110940, 11646],

'main_language': ['Arabic', 'English', 'Arabic', 'English', 'English', 'Portuguese', 'English', 'Spanish', 'Spanish', 'English', 'Mandarin', 'Hindi', 'Indonesian', 'Japanese', 'Vietnamese', 'German', 'Dutch', 'French', 'Italian', 'English', 'English', 'English', 'English', 'English', 'English']},

index = ['EG', 'KE', 'MA', 'NG', 'ZA', 'BR', 'CA', 'CL', 'MX', 'US', 'CN', 'IN', 'ID', 'JP', 'VN', 'AT', 'BE', 'FR', 'IT', 'GB', 'AU', 'FJ', 'NZ', 'TO', 'TV'])

df_countries_info
```

## 8.1 Select Data by Row, Column and Index

Pandas library provides multiple ways to select a group of rows and columns by labels or position values using `loc` and `iloc` functions. `loc` is a label-based selection function where users must specify rows and columns based on the row and column labels; while `iloc` is an integer position-based selection function where users must specify rows and columns by the integer position values (0-based integer position).

In the example below, we apply both loc and iloc to select a record based on its index label 'CN' or its 10th position in the DataFrame. We notice both methods would return a series object about China country information.

```
# Select one record
df_countries_info.loc['CN']
```

```
# Select one record
df_countries_info.iloc[10]
```

```
country_name    China
region          Asia
population      1433783686
main_language    Mandarin
Name: CN, dtype: object
```

We can also pass a list of index labels or position values as shown in the example below:

```
# Select a list of records
df_countries_info.loc[['CN','NZ','GB']]
```

```
# Select a list of records
df_countries_info.iloc[[10, 19, 22]]
```

	country_name	region	population	main_language
CN	China	Asia	1433783686	Mandarin
NZ	New Zealand	Oceania	4783063	English
GB	United Kingdom	Europe	67530172	English

We can also pass a range of index labels or position values using the colon sign : as shown in the example below.

```
# Select a range of values
df_countries_info.loc['CN':'NZ']
```

```
# Select a range of values
df_countries_info.iloc[10:22]
```

	country_name	region	population	main_language
CN	China	Asia	1433783686	Mandarin
IN	India	Asia	1366417754	Hindi
ID	Indonesia	Asia	270625568	Indonesian
JP	Japan	Asia	126860301	Japanese
VN	Vietnam	Asia	96462106	Vietnamese
AT	Austria	Europe	8955102	German
BE	Belgium	Europe	11539328	Dutch
FR	France	Europe	65129728	French
IT	Italy	Europe	60550075	Italian
GB	United Kingdom	Europe	67530172	English
AU	Australia	Oceania	25203198	English
FJ	Fiji	Oceania	889953	English
NZ	New Zealand	Oceania	4783063	English

We can also pass different DataFrame labels and positions as shown in the example below:

```
# Select a list of records and columns
df_countries_info.loc[['CN','NZ','GB'],['region','population']]
```

```
# Select a list of records and columns
df_countries_info.iloc[[10, 19, 22],[1,2]]
```

	region	population
CN	Asia	1433783686
NZ	Oceania	4783063
GB	Europe	67530172

## 8.2 Filter Data with Conditions

We can also select rows by adding filter conditions that only match a subset of records. Each individual condition is often surrounded by parentheses () and several conditions can be grouped together using AND and OR conditions represented with & or | symbols respectively.

In the following example, we retrieve all records with the main language being English and the region being Oceania. Note that we left the column selection area empty to indicate that we want all DataFrame columns.

```
df_countries_info.loc[(df_countries_info['main_language']=='English') &
                      (df_countries_info['region']=='Oceania'),]
```

	country_name	region	population	main_language
AU	Australia	Oceania	25203198	English
FJ	Fiji	Oceania	889953	English
NZ	New Zealand	Oceania	4783063	English
TO	Tonga	Oceania	110940	English
TV	Tuvalu	Oceania	11646	English

Sometimes we may need to sort a DataFrame or query output by specific numeric, alphabet, or date values. This process can be achieved by applying [`sort\_values\(\)`](#) function which takes the name of the targeted sorting value as a mandatory parameter and ascending order as default behavior. We can adjust the above example by sorting the result by population size as shown in the code below:

```
df_countries_info.loc[(df_countries_info['main_language']=='English') &
                      (df_countries_info['region']=='Oceania'),].sort_values(by='population')
```



## 8.3 Group and Sort Data

Finally, we will learn about how to query specific numerical values per group of records. In our toy example, we may want to know the summarization of population size per region or main language. This query can be answered by applying the Pandas [groupby\(\)](#) function on the targeted groups and a summarization function on the numerical value. In the examples below, we calculate the total population size per language and geographic region.

```
# Total population size by main language
df_countries_info.groupby('main_language').population.sum()
```

```
# Total population size by region
df_countries_info.groupby('region').population.sum()
```

# Common Data Accessing Methods Using

**LOC**

label-based selection

**&**

**ILOC**

integer position-based selection

DataFrame.loc/iloc[row labels, column labels]

dataframe

column index



row index



		0	1	2
		country	capital	population
0	CN	China	Beijing	1433783686
1	NZ	New Zealand	Wellington	4783063
2	SA	South Africa	Pretoria	58558270
3	UK	United Kingdom	London	67530172

## ACCESSING DATA BY VALUE

e.g., retrieve the record for *China*

dataframe.loc['CN']

dataframe.iloc[0]

## ACCESSING DATA BY LIST

e.g., retrieve *capital* & *population* for *China* & *UK*

dataframe.loc[['CN','UK'],  
['capital', 'population']]

dataframe.iloc[[0,3],[1,2]]

## ACCESSING DATA BY SLICING

e.g., retrieve all records for *China* & *NZ*

dataframe.loc[['CN', 'NZ'], :]

Both 'CN' and 'NZ' are included

dataframe.iloc[0:2,:]

0 is included but 2 is excluded

## Pandas Data Visualization

So far in this course, we have learned about many of the features that make Pandas one of the most popular data analysis and manipulation libraries. An essential part of any data analysis project is to perform a comprehensive exploratory analysis to help understand underlying patterns among the dataset variables.

Although Pandas is not designed for comprehensive data visualization, it is capable of creating basic and practical plots.

In this section, we will learn how to use Python Pandas to create two types of data visualizations:

- **Chart Visualization:** Graphical representation of data using graphs such as histograms, pie charts, box plots, and so on.
- **Table Visualization:** Graphical representation of data using HTML styling option within Pandas DataFrame objects.

Pandas chart visualization can be achieved using the [`plot\(\)`](#) method on `Series` and `DataFrame` objects. The function includes several parameters to identify the needed chart type as well as the targeted data columns. In this series of examples, we will use the [`plot\(\)`](#) function to create some popular chart types.

Time Series is one of the most commonly used charts to represent numerical values that change over a defined period of time. We can see this type of chart in visualizing

financial market data like sales and price movement. In the Pandas library, the `plot()` method can be applied to a Series object with index values representing date values. Note that the line chart is the default type in the `plot()` method. Therefore, we do not need to change the `type` parameter. The following example demonstrates how a time series plot is used for visualization as shown in the example below:

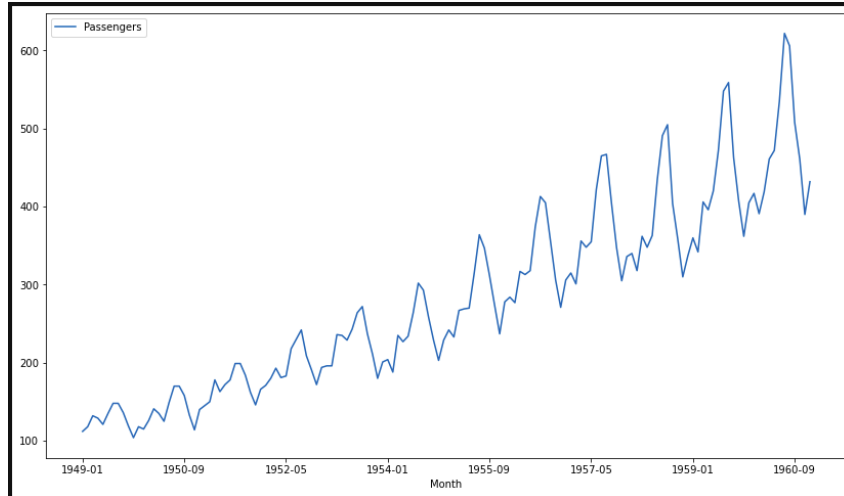
```
# Import the dataset as a CSV file
df_air_passengers =
pd.read_csv('https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv')

df_air_passengers.head()

# Set the Month values as the DataFrame index
df_air_passengers.set_index('Month', inplace = True)

# Plot the DataFrame
df_air_passengers.plot()
```

	Month	Passengers
0	1949-01	112
1	1949-02	118
2	1949-03	132
3	1949-04	129
4	1949-05	121



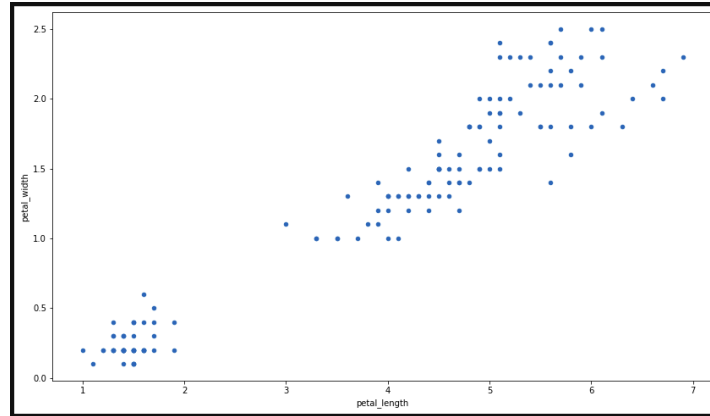
Another commonly used visualization type to examine the relationship between two different numerical variables is a scatter plot. In the example below, we use the popular `iris` dataset to explore the relationship between petal length and width variables. We can use color grouping to add additional analytics dimensions. In this example, we are adding color groups based on each flower species.

```
# Import the dataset as a CSV file
df_iris =
pd.read_csv('https://gist.githubusercontent.com/curran/a08a1080b88344b0c8a7/raw/0e7a9b0a5d22642
a06d3d5b9bcbad9890c8ee534/iris.csv')

df_iris.head()

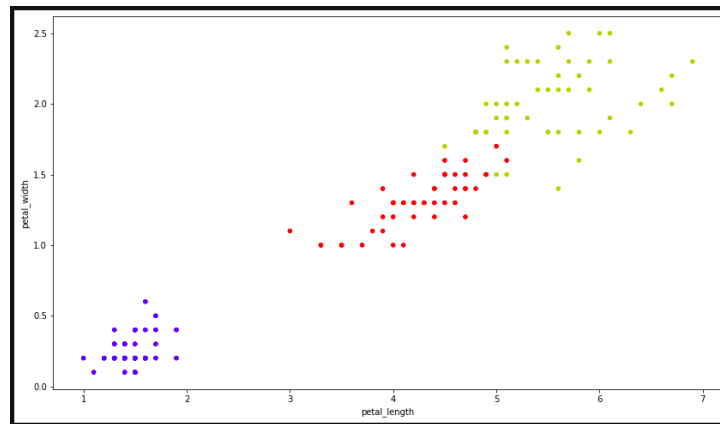
# Apply a scatter plot to examine the relationship between variables
df_iris.plot(kind = 'scatter', x = 'petal_length', y = 'petal_width')
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5	3.6	1.4	0.2	setosa



```
# Map species to different color values
col = df_iris['species'].map({'setosa':'b',
                              'versicolor':'r',
                              'virginica':'y'})

# Apply a scatter plot to examine the relationship between variables
df_iris.plot(kind = 'scatter', x = 'petal_length', y = 'petal_width', c=col, figsize= (14,8))
```



Pandas table visualization can be achieved by adding styling instructions to DataFrame objects in order to be rendered as CSS styles. In the example below, we apply the horizontal bar style to the variable `sepal_width` while the data is sorted in descending order. We notice in the top row, the bar line covers the entire cell and its size countune to get smaller for the rest of the DataFrame.

```
# Add bar style to sepal_width variable
df_iris.sort_values(by='sepal_width',
                    ascending=False).style.bar(subset=['sepal_width'],
                                                color='#00AEE8')
```

	sepal_length	sepal_width	petal_length	petal_width	species
15	5.700000	4.400000	1.500000	0.400000	setosa
33	5.500000	4.200000	1.400000	0.200000	setosa
32	5.200000	4.100000	1.500000	0.100000	setosa
14	5.800000	4.000000	1.200000	0.200000	setosa
16	5.400000	3.900000	1.300000	0.400000	setosa
5	5.400000	3.900000	1.700000	0.400000	setosa
19	5.100000	3.800000	1.500000	0.300000	setosa
44	5.100000	3.800000	1.900000	0.400000	setosa
46	5.100000	3.800000	1.600000	0.200000	setosa
131	7.900000	3.800000	6.400000	2.000000	virginica
117	7.700000	3.800000	6.700000	2.200000	virginica

## Pandas Analysis Project

So far in this crash course, you have learned how to use the Python library Pandas to tackle different data processing and analysis tasks. In this section, you will apply what you have learned to a simple data analysis project and extract useful insights from multiple datasets.

As we all know, the world has been facing a severe health crisis since 2019. Many people around the world have had to change the way they work, study and travel in order to fight the virus. In this project, we will examine different health and human activities datasets to see how COVID19 impacted our daily activities. The following datasets will be used:

[Community Mobility Reports](#): Aggregated datasets generated by google to report movement trends over time by geographical location. The dataset covers different types of locations such as workspaces, parks, residential and so on.

[COVID19 Confirmed Cases](#): A daily aggregated dataset collected and maintained by [Our World in Data](#).

### Project tasks

1. Download the mobility dataset for your country to your local computer. Perform an exploratory analysis to investigate the dataset.



2. Read this COVID19 [CSV](#) directly into Pandas and do an exploratory analysis for this dataset.
3. Both datasets from tasks 1 and 2 contain date values recorded at the daily level. Please use these values to visualize if Covid19 outbreaks impacted your community movement behavior using time series figures.

For example, we can examine how the emergence of Covid19 cases in New Zealand impacted the normal traffic at workspaces and outdoor parks. After setting *date* as index value for both DataFrames, you can use Pandas plotting feature to visualize Covid19 outbreak using the *new\_cases* column.

4. Can you investigate if people's normal work and travel behavior in your country has changed during the same time periods?

You can find the code solution to the previous four questions at the end of this section.

### Extra Challenge

Questions 5 - 8 are considered an extra challenge and can be solved using different data analysis methods you have learned during this course.

5. Use the Covid cases dataset to sort countries based on the vaccination rate as percentage of population. Identify the top five countries.

6. Use the mobility report dataset to investigate if working from home is becoming more popular in your country? Note that the dataset includes variables related to traffic behaviour at residence areas in comparison to before Covid19 outbreak.
7. Please use Pandas DataFrames joining techniques to create new DataFrame objects containing columns from both Mobility reports and Covid cases. The new DataFrame object should include records from your country.
8. The mobility report dataset can also highlight patterns with limited traffic behavior at workplaces at certain times of the year. This can be attributed to special events such as holidays and extreme weather events.

To investigate this, please create a new dataset about public holidays in your country during 2020 - 2021, and examine if holiday events can explain any changes in traffic behavior at workplaces and outdoor parks?

To access public holiday events in your country, refer to this website: [time and date](#)

### **Solution for tasks 1 - 4:**

1. We start by accessing the datasets using Pandas DataFrame objects. To investigate the content of our datasets, we can use `head()` and `info()` builtin functions as shown in the code below:

```
# Access dataset from local computer
df_mobility_data = pd.read_csv('Global_Mobility_Report.csv')
```

```
# Explore the content of the mobility dataset
df_mobility_data.info()

# Show sample records
df_mobility_data.head()
```

The results show the mobility report dataset contains 15 different columns describing information about geographical location and how much has changed in traffic behaviour compared to the time before Covid19 pandemic.

For more information about the values of each column, refer to the [dataset description page](#).

## 2. Read CSV directly to Pandas

```
# Access dataset from GitHub repository
Df_covid_cases =
pd.read_csv('https://raw.githubusercontent.com/owid/covid-19-data/master/public/data/owid-covid-
data.csv')

# Explore the content of the Covid cases dataset
df_covid_cases.info()

# Show sample records
df_covid_cases.head()
```

The Covid cases dataset contain 62 different columns describing infection and vaccination figures in each country in addition to other demographic figures.

To learn more about the different values, refer to the [dataset description page](#).

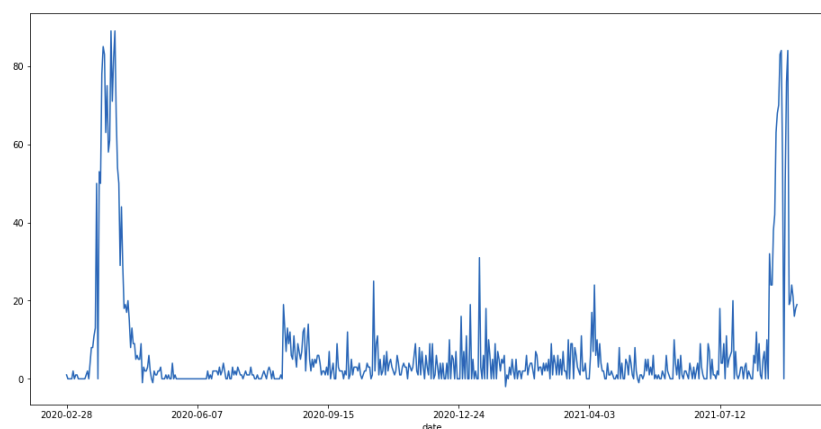
3. We noticed that both datasets contain date values recorded at the daily level. We will use these values to visualize if Covid19 outbreaks impacted your community movement behavior using time series figures.

To demonstrate this process, let's examine how the emergence of Covid19 cases in New Zealand impacted the normal traffic at workspace and outdoor parks. We first need to assign the `date` columns as the index value for both DataFrame objects in order to support our visualization task as shown in the code below:

```
# Set date as index value for both DataFrames
df_mobility_data.set_index('date', inplace = True)
df_covid_cases.set_index('date', inplace = True)
```

Next, we can use Pandas plotting feature to visualize Covid19 cases outbreak using the `new_cases` column as shown in the code below:

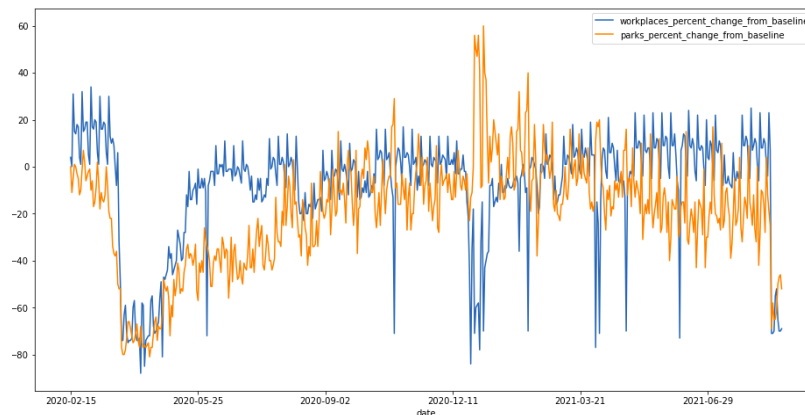
```
df_covid_cases[df_covid_cases['iso_code']=='NZL']['new_cases'].plot(figsize=(14,8))
```



4. From the plot above, we notice the Pandas query above used `iso_code` to filter only records belonging to a specific country code. Also we notice that Pandas visualization function `plot()` was applied to the column `new_cases`. The results demonstrate that the country of New Zealand had two large waves of Covid19 cases during March to April 2020 and August to September 2021.

For other time periods, the outbreak seems to be under control with an average of below 10 cases daily. Next, to investigate if people's normal work and travel behavior in New Zealand have changed during the same time periods, we can use the following code to query the mobility report DataFrame object to filter records for New Zealand and plot workspace and outdoor parks' traffic patterns.

```
df_mobility_data[(df_mobility_data['country_region_code']=='NZ') &
                  (df_mobility_data['sub_region_1'].isnull())][['workplaces_percent_change_from_baseline',
                                                                'parks_percent_change_from_baseline']].plot()
```



The results above show a clear pattern of reduced traffic during the same time periods of Covid19 in New Zealand. The figure also shows other significant time periods with increased and decreased traffic behavior. We clearly notice this pattern during the Christmas period and other major holidays.