Pandas

All of us can do data analysis using pen and paper on small data sets. We require specialized tools and techniques to analyze and derive meaningful information from massive datasets. Pandas Python is one of those libraries for data analysis that contains high-level data structures and tools to manipulate data in a simple way. Providing an effortless yet effective way to analyze data requires the ability to index, retrieve, split, join, restructure, and various other analyses on both multi and single-dimensional data.

Key Features of Pandas

Pandas data analysis library has some unique features that provide these capabilities-

I. The Series and DataFrame Objects
   These two are high-performance array and table structures for representing the heterogeneous and homogeneous data sets in Pandas Python.

II. Restructuring of Data Sets
   Pandas python provides the flexibility for reshaping the data structures to be inserted in both rows and columns of tabular data.

III. Labelling

   To allow automatic data alignment and indexing, pandas provide labeling on series and tabular data.

IV. Multiple Labels for a Data Item

   Heterogeneous indexing of data spread across multiple axes, which helps in creating more than one label on each data item.

V. Grouping

   The functionality to perform split-apply-combine on series as well on tabular data.

VI. Identify and Fix Missing Data

   Programmers can quickly identify and mix missing data floating and non-floating pointing numbers using pandas.

VII. Powerful capabilities to load and save data from various formats such as JSON, CSV, HDF5, etc.

VIII. Conversion from NumPy and Python data structures to pandas objects.

IX. Slicing and sub-setting of datasets, including merging and joining data sets with SQL- like constructs.

Although pandas provide many statistical methods, it is not enough to do data science in Python. Pandas depend upon other python libraries for data science like NumPy, SciPy, Sci-Kit Learn, Matplotlib, ggvis in the Python ecosystem to conclude from large data sets. Thus, making it possible for Pandas applications to take advantage of the robust and extensive Python framework.

Pros of using Pandas

- Pandas allow you to represent data effortlessly and in a simpler manner, improving data analysis and comprehension. For data science projects, such a simple data representation helps glean better insights.

- Pandas is highly efficient as it enables you to perform any task by writing only a few lines of code.
- Pandas provide users with a broad range of commands to analyze data quickly.

Cons of using Pandas

The learning curve for Pandas may appear to be simple at first, but as you start working with it, you may find it challenging to grasp.

One of the most evident flaws of Pandas is that it isn’t suitable for working with 3D matrices.

The topics are ordered as below:

1. **Input data**
2. **Overview of data**
3. **Handling missing values**
4. **Combining DataFrames**
5. **Selecting data on a DataFrame**
6. **Reshaping a DataFrame**
7. **Other pandas functions**

The basic data structure of Pandas is ***DataFrame*** *which represents data in tabular form with labeled rows and columns.*

As always, we start with importing numpy and pandas.

```
import numpy as np
import pandas as pd
```

1. Input Data

Reading from a file

In most cases, we read data from a file and convert to a DataFrame. Pandas provide functions to read data from many different file types. The most commonly used is **read_csv**. Other types are also available such as read_excel, read_json, read_html and so on. Let's go over an example using **read_csv**:

```
df = pd.read_csv("Churn_Modelling.csv")
df.head()
```

We need to specify the location of the file. In the file is in same working directory or folder, you can just write the name of the file. df.head() displays the first 5 rows.

```
df.head()
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	15634602	Hargrave	619	France	Female	42	2	0.00	1	1	1	101348.88	1
1	2	15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	1	112542.58	0
2	3	15619304	Onio	502	France	Female	42	8	159660.80	3	1	0	113931.57	1
3	4	15701354	Boni	699	France	Female	39	1	0.00	2	0	0	93826.63	0
4	5	15737888	Mitchell	850	Spain	Female	43	2	125510.82	1	1	1	79084.10	0

If we only need a part of the DataFrame and would not like to read all of it, we can specify the columns with **usecols** parameter:

```
cols = ['CustomerId','CreditScore','NumOfProducts']
df = pd.read_csv("Churn_Modelling.csv", usecols=cols)
df.head()
```

	CustomerId	CreditScore	NumOfProducts
0	15634602	619	1
1	15647311	608	1
2	15619304	502	3
3	15701354	699	2
4	15737888	850	1

We can also select how many rows to read using **nrows** parameter. It is useful when working with very large files:

```
df.shape
(10000, 3)df = pd.read_csv("Churn_Modelling.csv", usecols=cols, nrows=500)
```

```
df.shape
(500, 3)
```

Original DataFrame has 10000 rows and by setting nrows as 500, we only read the first 500 rows.

There are many other parameters of read_csv function which makes it powerful and handy.

Creating a DataFrame

In real life cases, we mostly read data from a file instead of creating a DataFrame. Pandas provide functions to create a DataFrame by reading data from various file types. For this post, I will use a dictionary to create a sample DataFrame.

```
df = pd.DataFrame({'a':np.random.rand(10),
                   'b':np.random.randint(10, size=10),
                   'c':[True,True,True,False,False,np.nan,np.nan,
                        False,True,True],
                   'b':['London','Paris','New York','Istanbul',
                        'Liverpool','Berlin',np.nan,'Madrid',
                        'Rome',np.nan],
                   'd':[3,4,5,1,5,2,2,np.nan,np.nan,0],
                   'e':[1,4,5,3,3,3,3,8,8,4]})
df
```

	a	b	c	d	e
0	0.093748	London	True	3.0	1
1	0.835929	Paris	True	4.0	4
2	0.166490	New York	True	5.0	5
3	0.439057	Istanbul	False	1.0	3
4	0.077856	Liverpool	False	5.0	3
5	0.669849	Berlin	NaN	2.0	3
6	0.539958	NaN	NaN	2.0	3
7	0.323087	Madrid	False	NaN	8
8	0.367877	Rome	True	NaN	8
9	0.281026	NaN	True	0.0	4

2. Overview of Data

Photo by Markus Winkler on Unsplash

Pandas **describe** function provides summary statistics for numerical (int or float) columns. It counts the number of values and show mean, std, min and max values as well as 25%, 50% and 75% quantiles.

	a	b	c	d	e
0	0.093748	London	True	3.0	1
1	0.835929	Paris	True	4.0	4
2	0.166490	New York	True	5.0	5
3	0.439057	Istanbul	False	1.0	3
4	0.077856	Liverpool	False	5.0	3
5	0.669849	Berlin	NaN	2.0	3
6	0.539958	NaN	NaN	2.0	3
7	0.323087	Madrid	False	NaN	8
8	0.367877	Rome	True	NaN	8
9	0.281026	NaN	True	0.0	4

```
df.describe()
```

	a	d	e
count	10.000000	8.000000	10.000000
mean	0.379488	2.750000	4.200000
std	0.247870	1.832251	2.250926
min	0.077856	0.000000	1.000000
25%	0.195124	1.750000	3.000000
50%	0.345482	2.500000	3.500000
75%	0.514733	4.250000	4.750000
max	0.835929	5.000000	8.000000

Although all of the columns have the same number of rows, count is different for column d because describe function does not count NaN (missing) values.

value_counts() shows the values in a column with number of occurrences:

```
df.c.value_counts()
True     5
False    3
Name: c, dtype: int64
```

value_counts() does not count NaN (missing) values.

We should also check the data types and consider them in our analysis. Some functions can only be performed on certain data types. We can easily check the data types using **dtypes**:

```
df.dtypes
a    float64
b     object
c     object
d    float64
e      int64
dtype: object
```

Both ‘d’ and ‘e’ columns have integers but data type of ‘d’ column is float. The reason is the NaN values in column d. NaN values are considered to be float so integer values in that column are upcasted to float data type.

Pandas 1.0.1 allow using NaN as integer data type. We just need to explicitly indicate dtype as **pd.Int64Dtype():**

```
pd.Series([1, 2, 3, np.nan], dtype=pd.Int64Dtype())
0      1
```

```
1      2
2      3
3    NaN
dtype: Int64
```

If pd.Int64Dtype() is not used, integer values are upcasted to float:

```
pd.Series([1, 2, 3, np.nan])
0    1.0
1    2.0
2    3.0
3    NaN
dtype: float64
```

3. Handling Missing Values

Photo by Fahrul Azmi on Unsplash

Handling missing values is an essential part of data cleaning and preparation process because almost all data in real life comes with some missing values.

Let's create a dataframe with missing values first.

```
df = pd.DataFrame({
'column_a':[1, 2, 4, 4, np.nan, np.nan, 6],
'column_b':[1.2, 1.4, np.nan, 6.2 ,None, 1.1, 4.3],
'column_c':['a', '?', 'c', 'd', '--', np.nan, 'd'],
'column_d':[True, True, np.nan, None, False, True, False]
})df
```

	column_a	column_b	column_c	column_d
0	1.0	1.2	a	True
1	2.0	1.4	?	True
2	4.0	NaN	c	NaN
3	4.0	6.2	d	None
4	NaN	NaN	--	False
5	NaN	1.1	NaN	True
6	6.0	4.3	d	False

np.nan, None and NaT (for datetime64[ns] types) are standard missing value for Pandas.

> ***Note:*** *A new missing data type (<NA>) introduced with Pandas 1.0 which is an integer type missing value representation.*

np.nan is a float so if you use them in a column of integers, they will be upcast to floating-point data type as you can see in "column_a" of the dataframe we created. However, <NA> can be used with integers without causing upcasting. Let's add one more column to the dataframe using <NA> which can be used by explicitly requesting the dtype Int64Dtype().

```
new_column = pd.Series([1, 2, np.nan, 4, np.nan, 5], dtype=pd.Int64Dtype())df['column_e'] =
new_column
df
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1
1	2.0	1.4	?	True	2
2	4.0	NaN	c	NaN	<NA>
3	4.0	6.2	d	None	4
4	NaN	NaN	--	False	<NA>
5	NaN	1.1	NaN	True	5
6	6.0	4.3	d	False	<NA>

Pandas provides **isnull()**, **isna()** functions to detect missing values. Both of them do the same thing.

df.isna() returns the dataframe with boolean values indicating missing values.

```
df.isna()
```

	column_a	column_b	column_c	column_d	column_e
0	False	False	False	False	False
1	False	False	False	False	False
2	False	True	False	True	True
3	False	False	False	True	False
4	True	True	False	False	True
5	True	False	True	False	False
6	False	False	False	False	True

You can also choose to use **notna()** which is just the opposite of isna().

df.isna().any() returns a boolean value for each column. If there is at least one missing value in that column, the result is True.

df.isna().sum() returns the number of missing values in each column.

```
df.isna().any()
```

```
column_a    True
column_b    True
column_c    True
column_d    True
column_e    True
dtype: bool
```

```
df.isna().sum()
```

```
column_a    2
column_b    2
column_c    1
column_d    2
column_e    3
dtype: int64
```

```
df.isna().sum().sum()
```

```
10
```

Not all missing values come in nice and clean np.nan or None format. For example, “?” and “- - “ characters in column_c of our dataframe do not give us any valuable information or insight so

essentially they are missing values. However, these characters cannot be detected as missing value by Pandas.

If we know what kind of characters used as missing values in the dataset, we can handle them while creating the dataframe using **na_values** parameter:

```
missing_values = ['?', '--']
df_test = pd.read_csv('filepath.csv', na_values = missing_values)
```

Another option is to use pandas **replace()** function to handle these values after a dataframe is created:

```
df.replace(['?','--'],np.nan, inplace=True)
df
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1
1	2.0	1.4	NaN	True	2
2	4.0	NaN	c	NaN	<NA>
3	4.0	6.2	d	None	4
4	NaN	NaN	NaN	False	<NA>
5	NaN	1.1	NaN	True	5
6	6.0	4.3	d	False	<NA>

We have replaced non-informative cells with NaN values. **inplace** parameter saves the changes in the dataframe. Default value for inplace is False so if it is set it to True, changes will not be saved.

There is not an optimal way to handle missing values. Depending on the characteristics of the dataset and the task, we can choose to:

- Drop missing values
- Replace missing values

Drop missing values

We can drop a row or column with missing values using **dropna()** function. **how** parameter is used to set condition to drop.

- how='any' : drop if there is any missing value
- how='all' : drop if all values are missing

Furthermore, using **thresh** parameter, we can set a threshold for missing values in order for a row/column to be dropped.

```
df.dropna(axis=0, how='all', inplace=True)
df
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1
1	2.0	1.4	NaN	True	2
2	4.0	NaN	c	NaN	<NA>
3	4.0	6.2	d	None	4
4	NaN	NaN	NaN	False	<NA>
5	NaN	1.1	NaN	True	5
6	6.0	4.3	d	False	<NA>

axis parameter is used to select row (0) or column (1).

Our dataframe do not have a row with full of missing values so setting how='all' did not drop any row. The default value is 'any' so we don't need to specify it if we want to use how='any':

```
df.dropna(axis=0)
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1

> ***Note: inplace*** *parameter is used to save the changes to the original DataFrame. The default value is false so if we do not set it as True, the changes we make are not saved in the original DataFrame.*

```
df.dropna(axis=0, thresh=3)
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1
1	2.0	1.4	NaN	True	2
3	4.0	6.2	d	None	4
5	NaN	1.1	NaN	True	5
6	6.0	4.3	d	False	<NA>

Setting thresh parameter to 3 dropped rows with at least 3 missing values.

Data is a valuable asset so we should not give it up easily. Also, machine learning models almost alwasy tend to perform better with more data. Therefore, depending on the situation, we may prefer replacing missing values instead of dropping.

Replacing missing values

fillna() function of Pandas conveniently handles missing values. Using fillna(), missing values can be replaced by a special value or an aggreate value such as mean, median. Furthermore, missing values can be replaced with the value before or after it which is pretty useful for time-series datasets.

- Replace missing values with a scalar:

```
df.fillna(25)
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1
1	2.0	1.4	25	True	2
2	4.0	25.0	c	25	25
3	4.0	6.2	d	25	4
4	25.0	25.0	25	False	25
5	25.0	1.1	25	True	5
6	6.0	4.3	d	False	25

- fillna() can also be used on a particular column:

```
mean = df['column_a'].mean()
df['column_a'].fillna(mean)
```

```
0    1.0
1    2.0
2    4.0
3    4.0
4    3.4
5    3.4
6    6.0
Name: column_a, dtype: float64
```

- Using **method** parameter, missing values can be replaced with the values before or after them.

```
df.fillna(axis=0, method='ffill')
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1
1	2.0	1.4	a	True	2
2	4.0	1.4	c	True	2
3	4.0	6.2	d	True	4
4	4.0	6.2	d	False	4
5	4.0	1.1	d	True	5
6	6.0	4.3	d	False	5

ffill stands for “forward fill” replaces missing values with the values in the previous row. You can also choose **bfill** which stands for “backward fill”.

If there are many consecutive missing values in a column or row, you may want to **limit** the number of missing values to be forward or backward filled.

```
df.fillna(axis=0, method='ffill', limit=1)
```

	column_a	column_b	column_c	column_d	column_e
0	1.0	1.2	a	True	1
1	2.0	1.4	a	True	2
2	4.0	1.4	c	True	2
3	4.0	6.2	d	None	4
4	4.0	6.2	d	False	4
5	NaN	1.1	NaN	True	5
6	6.0	4.3	d	False	5

Limit parameter is set to 1 so only one missing value is forward filled.

4. Combining DataFrames

Photo by Denys Nevozhai on Unsplash

Data science projects usually require us to gather data from different sources. Hence, as part of data preparation, we may need to combine DataFrames. In this section, I will cover the following topics:

- Concat
- Merge

Concat

Let's first create two DataFrames:

```
df1 = pd.DataFrame({
'column_a':[1,2,3,4],
'column_b':['a','b','c','d'],
'column_c':[True,True,False,True]
})df2 = pd.DataFrame({
'column_a':[1,2,9,10],
'column_b':['a','k','l','m'],
'column_c':[False,False,False,True]
})
```

```
df1
```

	column_a	column_b	column_c
0	1	a	True
1	2	b	True
2	3	c	False
3	4	d	True

```
df2
```

	column_a	column_b	column_c
0	1	a	False
1	2	k	False
2	9	l	False
3	10	m	True

One way to combine or concatenate DataFrames is **concat()** function. It can be used to concatenate DataFrames along rows or columns by changing the **axis** parameter. The default value of the axis parameter is 0, which indicates combining along rows.

```
df = pd.concat([df1,df2])
df
```

	column_a	column_b	column_c
0	1	a	True
1	2	b	True
2	3	c	False
3	4	d	True
0	1	a	False
1	2	k	False
2	9	l	False
3	10	m	True

```
df = pd.concat([df1,df2], axis=1)
df
```

	column_a	column_b	column_c	column_a	column_b	column_c
0	1	a	True	1	a	False
1	2	b	True	2	k	False
2	3	c	False	9	l	False
3	4	d	True	10	m	True

As you can see in the first figure above, indices of individual DataFrames are kept. In order to change it and re-index the combined DataFrame, **ignore_index** parameter is set as True.

```
df = pd.concat([df1,df2], ignore_index=True)
df
```

	column_a	column_b	column_c
0	1	a	True
1	2	b	True
2	3	c	False
3	4	d	True
4	1	a	False
5	2	k	False
6	9	l	False
7	10	m	True

join parameter of concat() function determines how to combine DataFrames. The default value is ‘outer’ returns all indices in both DataFrames. If ‘inner’ option is selected, only the rows with shared indices are returned. I will change the index of df2 so that you can see the difference between ‘inner’ and ‘outer’.

```
df1
```

	column_a	column_b	column_c
0	1	a	True
1	2	b	True
2	3	c	False
3	4	d	True

```
df2
```

	column_a	column_b	column_c
3	1	a	False
4	2	k	False
5	9	l	False
6	10	m	True

```
df = pd.concat([df1,df2], axis=1, join='inner')
df
```

	column_a	column_b	column_c	column_a	column_b	column_c
3	4	d	True	1	a	False

```
df = pd.concat([df1,df2], axis=1, join='outer')
df
```

	column_a	column_b	column_c	column_a	column_b	column_c
0	1.0	a	True	NaN	NaN	NaN
1	2.0	b	True	NaN	NaN	NaN
2	3.0	c	False	NaN	NaN	NaN
3	4.0	d	True	1.0	a	False
4	NaN	NaN	NaN	2.0	k	False
5	NaN	NaN	NaN	9.0	l	False
6	NaN	NaN	NaN	10.0	m	True

Pandas also provides ways to label DataFrames so that we know which part comes from which DataFrame. We just pass the list of combined DataFrames in order using **keys** parameter.

```
df = pd.concat([df1, df2], keys=['df1', 'df2'])
df
```

		column_a	column_b	column_c
df1	0	1	a	True
	1	2	b	True
	2	3	c	False
	3	4	d	True
df2	0	1	a	False
	1	2	k	False
	2	9	l	False
	3	10	m	True

It also makes it easier to access different parts of DataFrames conveniently:

```
df.loc['df1']
```

	column_a	column_b	column_c
0	1	a	True
1	2	b	True
2	3	c	False
3	4	d	True

> *One important note about concat() function is that it makes a copy of the data. To prevent making unnecessary copies, the* ***copy*** *parameter needs to set as False. The default value is True.*

append() function is also used to combine DataFrames. It can be seen as a particular case of concat() function (axis=0 and join='outer') so I will not cover it in detail but will just give an example to show the syntax.

```
df = df1.append(df2)
df
```

	column_a	column_b	column_c
0	1	a	True
1	2	b	True
2	3	c	False
3	4	d	True
0	1	a	False
1	2	k	False
2	9	l	False
3	10	m	True

Merge

Another widely used function to combine DataFrames is **merge()**. Concat() function simply adds DataFrames on top of each other or adds them side-by-side. It is more like appending DataFrames. Merge() combines DataFrames based on values in shared columns. Merge() function offers more flexibility compared to concat() function. It will be clear when you see the examples.

```
df1
```

	column_a	column_b	column_c
0	1	a	True
1	2	b	True
2	3	c	False
3	4	d	True

```
df2
```

	column_a	column_b	column_c
0	1	a	False
1	2	k	False
2	9	l	False
3	10	m	True

The **on** parameter selects which column or index level is used to merge.

```
df_merge = pd.merge(df1, df2, on='column_a')
df_merge
```

	column_a	column_b_x	column_c_x	column_b_y	column_c_y
0	1	a	True	a	False
1	2	b	True	k	False

The column names do not have to be the same. Our focus is the values in columns. Assume two DataFrames have common values in a column that you want to use to merge these DataFrames but the column names are different. In this case, instead of **on** parameter, you can use **left_on** and **right_on** parameters. To show the difference, I will change the column name in df2 and then use merge:

```
df2.rename(columns={'column_a':'new_column_a'}, inplace=True)
df2
```

	new_column_a	column_b	column_c
0	1	a	False
1	2	k	False
2	9	l	False
3	10	m	True

```
df_merge = pd.merge(df1, df2, left_on='column_a', right_on='new_column_a')df_merge
```

	column_a	column_b_x	column_c_x	new_column_a	column_b_y	column_c_y
0	1	a	True	1	a	False
1	2	b	True	2	k	False

Although the returned values are the same in column_a and new_column_a, merged DataFrame includes both columns due to having different names.

You can also pass multiple values to **on** parameter. The returned DataFrame only includes rows that have the same values in all the columns passed to **on** parameter.

```
df2.rename(columns={'new_column_a':'column_a'}, inplace=True)df_merge = pd.merge(df1, df2,
on=['column_a','column_b'])
df_merge
```

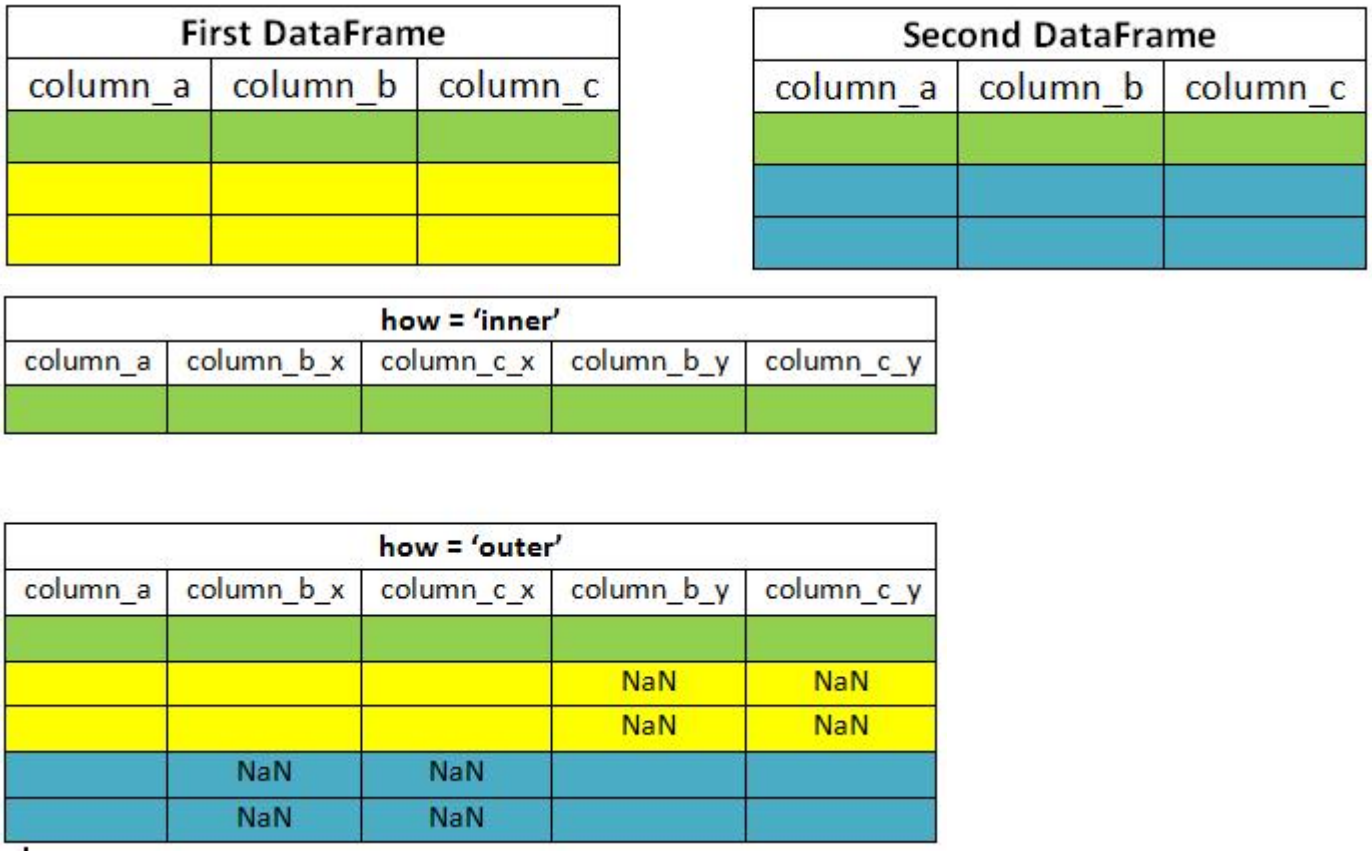
	column_a	column_b	column_c_x	column_c_y
0	1	a	True	False

df1 and df2 are merged based on the common values in column_a. It's time to introduce **how** parameter of merge(). As the name suggests, it indicates how you want to combine. The possible values for how are 'inner', 'outer', 'left', 'right'.

- inner: only rows with same values in the column specified by **on** parameter (default value of **how** parameter)
- outer: all the rows
- left: all rows from left DataFrame
- right: all rows from right DataFrame

The concept of how is more clear in the figure below. If you are familiar with SQL, the logic is same as SQL joins.

The figures below represents the concept of how parameters more clearly.



how = 'left'				
column_a	column_b_x	column_c_x	column_b_y	column_c_y
			NaN	NaN
			NaN	NaN

how = 'right'				
column_a	column_b_x	column_c_x	column_b_y	column_c_y
	NaN	NaN		
	NaN	NaN		

'outer', 'left', and 'right' options include data that is not present in one of the DataFrames. The missing parts are automatically filled with **NaN** values. NaN is the standard representation of missing values in Pandas.

The default value of how is 'inner' so you don't have to explicitly write in the function. 'Inner' only returns the rows with common values in column_a.

When 'outer' is chosen for how parameter, merged DataFrame includes all values of column_a in both DataFrames. However, common values (column_a = 1 and column_a = 2) are not duplicated.

```
df1
```

	column_a	column_b	column_c
0	1	a	True
1	2	b	True
2	3	c	False
3	4	d	True

```
df2
```

	column_a	column_b	column_c
0	1	a	False
1	2	k	False
2	9	l	False
3	10	m	True

```
df_merge = pd.merge(df1, df2, on='column_a')
df_merge
```

	column_a	column_b_x	column_c_x	column_b_y	column_c_y
0	1	a	True	a	False
1	2	b	True	k	False

```
df_merge = pd.merge(df1, df2, how='outer', on='column_a')
df_merge
```

	column_a	column_b_x	column_c_x	column_b_y	column_c_y
0	1	a	True	a	False
1	2	b	True	k	False
2	3	c	False	NaN	NaN
3	4	d	True	NaN	NaN
4	9	NaN	NaN	l	False
5	10	NaN	NaN	m	True

When ‘left’ is chosen for how parameter, merged DataFrame includes all rows from left DataFrame. The columns that come from right DataFrame are filled with NaN values if the value in column_a (the column that is passed to on parameter) is not present in right DataFrame. ‘right’ option is rarely used because you can just change the order of DataFrames in merge function (instead of (df1,df2) use (df2,df1)).

```
df_merge = pd.merge(df1, df2, how='left', on='column_a')
df_merge
```

	column_a	column_b_x	column_c_x	column_b_y	column_c_y
0	1	a	True	a	False
1	2	b	True	k	False
2	3	c	False	NaN	NaN
3	4	d	True	NaN	NaN

You may have noticed that a suffix is added to column names that are same in both DataFrames. It is useful to distinguish which column comes from which DataFrame. You can specify the suffix to be added using **suffixes** parameter.

```
df_merge = pd.merge(df1, df2, on='column_a', suffixes=['_df1','_df2'])
df_merge
```

	column_a	column_b_df1	column_c_df1	column_b_df2	column_c_df2
0	1	a	True	a	False
1	2	b	True	k	False

5. Selecting Data on a DataFrame

Photo by Bodie Pyndus on Unsplash

iloc and **loc** allows selecting part of a DataFrame.

- iloc: Select by position
- loc: Select by label

Let's go through some examples because, as always, practice makes perfect. I will use the following DataFrame for the examples in this section:

```
df
```

	a	b	c	d	e
0	0.041211	London	True	3.0	1
1	0.571258	Paris	True	4.0	4
2	0.676766	New York	True	5.0	5
3	0.674262	Istanbul	False	1.0	3
4	0.477875	Liverpool	False	5.0	3
5	0.532795	Berlin	NaN	2.0	3
6	0.823467	NaN	NaN	2.0	3
7	0.929395	Madrid	False	NaN	8
8	0.652076	Rome	True	NaN	8
9	0.722104	NaN	True	0.0	4

iloc

Select second row:

```
df.iloc[1]
a    0.571258
b       Paris
c        True
d           4
e           4
Name: 1, dtype: object
```

Select first row, second column (i.e. the second value in the first row):

```
df.iloc[0,1]
'London'
```

All rows, third column (It is same as selecting the second column but I just want to show the use of ':'):

```
df.iloc[:,2]
0     True
1     True
2     True
3    False
4    False
5     True
7    False
8     True
9     True
Name: c, dtype: bool
```

First two rows, second column:

```
df.iloc[:2,1]
0    London
1     Paris
Name: b, dtype: object
```

loc

Rows up to 2, column 'b' :

```
df.loc[:2,'b']
0      London
1       Paris
2    New York
Name: b, dtype: object
```

Rows up to 2 and columns up to 'b' :

```
df.loc[:2, :'b']
```

	a	b
0	0.093748	London
1	0.835929	Paris
2	0.166490	New York

Row ‘2’ and columns up to ‘b’ :

```
df.loc[2, :'b']
a     0.16649
b    New York
Name: 2, dtype: object
```

> *You may wonder why we use same values for rows in both loc and iloc. The reason is the numerical index. Loc selects by position but the position of rows are same as index.*

Let’s create a new DataFrame with a non-numerical index so that we can see the difference:

```
index = ['aa','bb','cc','dd','ee']
df2 = pd.DataFrame({'a':np.random.rand(5),
                    'b':np.random.randint(10, size=5)},
                      index = index)
df2
```

	a	b
aa	0.892290	8
bb	0.174937	1
cc	0.600939	5
dd	0.504597	7
ee	0.942866	6

```
df2.loc['bb','b']
1df2.loc[:'cc','a']
aa    0.892290
bb    0.174937
cc    0.600939
Name: a, dtype: float64
```

6. Reshaping a DataFrame

There are multiple ways to reshape a dataframe. We can choose the one that best fits the task at hand. The functions to reshape a dataframe:

- **Melt**
- **Stack and unstack**

Melt

Melt is used to convert wide dataframes to narrow ones. What I mean by wide is a dataframe with a high number of columns. Some dataframes are structured in a way that consecutive measurements or variables are represented as columns. In some cases, representing these columns as rows may fit better to our task.

Consider the following dataframe:

```
df1 = pd.DataFrame({'city':['A','B','C'],
                    'day1':[22,25,28],
                    'day2':[10,14,13],
                    'day3':[25,22,26],
                    'day4':[18,15,17],
                    'day5':[12,14,18]})
```

```
df1
```

	city	day1	day2	day3	day4	day5
0	A	22	10	25	18	12
1	B	25	14	22	15	14
2	C	28	13	26	17	18

We have three different cities and measurements done on different days. We decide to represent these days as rows in a column. There will also be a column to show the measurements. We can easily accomplish this by using **melt** function:

```
df1.melt(id_vars=['city'])
```

```
df1.melt(id_vars=['city'])
```

	city	variable	value
0	A	day1	22
1	B	day1	25
2	C	day1	28
3	A	day2	10
4	B	day2	14
5	C	day2	13
6	A	day3	25
7	B	day3	22
8	C	day3	26
9	A	day4	18
10	B	day4	15
11	C	day4	17
12	A	day5	12
13	B	day5	14
14	C	day5	18

Variable and value column names are given by default. We can use **var_name** and **value_name** parameters of melt function to assign new column names. It will also look better if we sort the data by city column:

```
df1.melt(id_vars=['city'], var_name = 'date', value_name =
'temperature').sort_values(by='city').reset_index(drop=True)
```

```
df1.melt(id_vars=['city'], var_name = 'date',
         value_name = 'temperature').sort_values(by='city').reset_index(drop=True)
```

	city	date	temperature
0	A	day1	22
1	A	day2	10
2	A	day3	25
3	A	day4	18
4	A	day5	12
5	B	day1	25
6	B	day2	14
7	B	day3	22
8	B	day4	15
9	B	day5	14
10	C	day1	28
11	C	day2	13
12	C	day3	26
13	C	day4	17
14	C	day5	18

Stack and unstack

Stack function kind of increases the index level of the dataframe. What I mean by increasing the level is:

- If dataframe has a simple column index, stack returns a series whose indices consist of row-column pairs of original dataframe.
- If dataframe has multi-level index, stack increases the index level.

It is better explained with examples. Consider the following dataframe:

```
df1
```

	city	day1	day2	day3	day4	day5
0	A	22	10	25	18	12
1	B	25	14	22	15	14
2	C	28	13	26	17	18

df1 has 3 rows and 6 columns with simple integer column index. If stack function is applied to df1, it will return a series with 3 x 6 = 18 rows. The index of the series will be [(0, 'city'), (0, 'day1'), … , (2, 'day5')].

```
df1.stack()
```

```
0  city     A
   day1    22
   day2    10
   day3    25
   day4    18
   day5    12
1  city     B
   day1    25
   day2    14
   day3    22
   day4    15
   day5    14
2  city     C
   day1    28
   day2    13
   day3    26
   day4    17
   day5    18
dtype: object
```

Let's also check the shape and index:

```
df1.shape
(3,6)
df1.stack().shape
(18,)
df1.stack().index[0] #multilevel index
(0, 'city')
```

Stack and unstack functions are more commonly used for dataframes with multi-level indices. Let's create a dataframe with multi-level index:

```
tuples = [('A',1),('A',2),('A',3),('B',1),('A',2)]index = pd.MultiIndex.from_tuples(tuples,
names=['first','second'])df2 = pd.DataFrame(np.random.randint(10, size=(5,2)),
                   index=index, columns=['column_x', 'column_y'])
```

```
df2
```

		column_x	column_y
first	second		
A	1	5	5
	2	7	6
	3	6	8
B	1	8	0
A	2	1	0

If we apply stack function on this dataframe, the level of index will be increased:

```
df_stacked = df2.stack().to_frame()
df_stacked
```

			0
first	second		
A	1	column_x	1
		column_y	1
	2	column_x	0
		column_y	3
	3	column_x	1
		column_y	8
B	1	column_x	8
		column_y	6
	2	column_x	9
		column_y	4

Now the names of the colums (column_x and column_y) are part of multi-level index. So the resulting dataframe has one column and a 3-level multi-index.

```
len(df_stacked.index.levels)
3len(df2.index.levels)
2
```

```
df_stacked.index.levels
```

```
FrozenList([['A', 'B'], [1, 2, 3], ['column_x', 'column_y']])
```

```
df2.index.levels
```

```
FrozenList([['A', 'B'], [1, 2, 3]])
```

Unstack is just the opposite of **stack**. If we apply unstack to the stacked dataframe, we will get back the original dataframe:

```
df_stacked.unstack()
```

		0	
		column_x	column_y
first	second		
A	1	1	1
	2	0	3
	3	1	8
B	1	8	6
	2	9	4

```
df_stacked.unstack().index
MultiIndex(levels=[['A', 'B'], [1, 2, 3]],
           codes=[[0, 0, 0, 1, 1], [0, 1, 2, 0, 1]],
           names=['first', 'second'])df2.index
MultiIndex(levels=[['A', 'B'], [1, 2, 3]],
           codes=[[0, 0, 0, 1, 1], [0, 1, 2, 0, 1]],
           names=['first', 'second'])
```

```
df_stacked.unstack().index == df2.index
```

```
array([ True,  True,  True,  True,  True])
```

7. Other Pandas Functions

The functions covered in this part:

- Explode
- Nunique
- Lookup

- Where
- Infer_objects

Photo by Andre Furtado on Unsplash

Explode

Assume your data set includes multiple entries of a feature on a single observation (row) but you want to analyze them on separate rows.

```
df = pd.DataFrame({'ID':['a','b','c'],
                   'measurement':[4,6,[2,3,8]],
                   'day':1})
df
```

	ID	measurement	day
0	a	4	1
1	b	6	1
2	c	[2, 3, 8]	1

We want to see the measurements of ‘c’ on day ‘1’ on separate rows which easily be done using **explode**:

```
df.explode('measurement').reset_index(drop=True)
```

	ID	measurement	day
0	a	4	1
1	b	6	1
2	c	2	1
2	c	3	1
2	c	8	1

It is better to reset the index as well:

	ID	measurement	day
0	a	4	1
1	b	6	1
2	c	2	1
3	c	3	1
4	c	8	1

We can also use explode on two columns as a chain:

```
df2 = pd.DataFrame({'ID':['a','b','c'],
                    'measurement_1':[4,6,[2,3,8]],
                    'measurement_2':[1,[7,9],1],
                    'day':1})
df2
```

	ID	measurement_1	measurement_2	day
0	a	4	1	1
1	b	6	[7, 9]	1
2	c	[2, 3, 8]	1	1

```
df2.explode('measurement_1').reset_index(drop=True).explode('measurement_2').reset_index(drop=Tr
ue)
```

	ID	measurement_1	measurement_2	day
0	a	4	1	1
1	b	6	7	1
2	b	6	9	1
3	c	2	1	1
4	c	3	1	1
5	c	8	1	1

Make sure to use reset_index after the first explode or you will get an unexpected result as below:

```
df2.explode('measurement_1').explode('measurement_2').reset_index(drop=True)
```

	ID	measurement_1	measurement_2	day
0	a	4	1	1
1	b	6	7	1
2	b	6	9	1
3	c	2	1	1
4	c	2	1	1
5	c	2	1	1
6	c	3	1	1
7	c	3	1	1
8	c	3	1	1
9	c	8	1	1
10	c	8	1	1
11	c	8	1	1

Nunique

Nunique counts the number of unique entries over columns or rows. It is very useful in categorical features especially in cases where we do not know the number of categories beforehand.

Photo by Kyle Glenn on Unsplash

Assume we have the following DataFrame:

	ID	name	measurement1	measurement2	measurement3
0	1	John	4	4	4
1	2	Alex	7	6	2
2	3	Alex	8	9	3
3	4	Alex	9	2	5
4	5	Oscar	2	6	1
5	6	Derek	6	6	4
6	7	Derek	6	5	5
7	8	Will	5	5	9

To find the number of unique values in a column:

```
df.name.nunique()
5
```

We can achieve the same result using **value_counts** with a slightly more complicated syntax:

```
df.name.value_counts().shape[0]
5
```

However, **nunique** allows us to do this operation on all columns or rows at the same time:

```
df.nunique()  #columns
ID              8
name            5
measurement1    7
measurement2    5
measurement3    6dtype: int64df.nunique(axis=1) #rows
0    3
1    4
2    4
3    5
4    5
5    3
6    4
7    4
dtype: int64
```

Lookup

It can be used to look up values in the DataFrame based on the values on other row, column pairs. This function is best explained via an example. Assume we have the following DataFrame:

	Day	Person	John	Alex	Oscar	Derek
0	1	Alex	4	4	4	6
1	2	John	7	6	2	2
2	3	Alex	8	9	3	1
3	4	Derek	9	2	5	8
4	5	Oscar	2	6	1	7
5	6	John	6	6	4	8
6	7	Derek	6	5	5	4
7	8	Oscar	5	5	9	5

For each day, we have measurements of 4 people and a column that includes the names of these 4 people. We want to create a new column that shows the measurement of the person in “Select” column:

```
df['Person_point'] = df.lookup(df.index, df['Person'])
df
```

	Day	Person	John	Alex	Oscar	Derek	Person_point
0	1	Alex	4	4	4	6	4
1	2	John	7	6	2	2	7
2	3	Alex	8	9	3	1	9
3	4	Derek	9	2	5	8	8
4	5	Oscar	2	6	1	7	1
5	6	John	6	6	4	8	6
6	7	Derek	6	5	5	4	4
7	8	Oscar	5	5	9	5	9

We do not have to do this operation on all data points. We can use a specific range as long as the row and column labels have the same size:

```
df.lookup(df.index[:5], df['Person'][:5])
array([4, 7, 9, 8, 1])
```

Where

"Where" is used to replace values in rows or columns based on a condition. You can also specify the value to be put as replacement. The default is NaN. Let's go over an example so that it becomes clear.

	Day	Person	John	Alex	Oscar	Derek	Person_point
0	1	Alex	4	4	4	6	4
1	2	John	7	6	2	2	7
2	3	Alex	8	9	3	1	9
3	4	Derek	9	2	5	8	8
4	5	Oscar	2	6	1	7	1
5	6	John	6	6	4	8	6
6	7	Derek	6	5	5	4	4
7	8	Oscar	5	5	9	5	9

```
df['Person_point'].where(df['Person_point'] > 5)0    NaN
1    7.0
2    9.0
3    8.0
4    NaN
```

```
5    6.0
6    NaN
7    9.0
Name: Person_point, dtype: float64
```

We can specify the replacement value:

```
df['Person_point'].where(df['Person_point'] > 5, 'Not_qualified', inplace=True)df
```

	Day	Person	John	Alex	Oscar	Derek	Person_point
0	1	Alex	4	4	4	6	Not_qualified
1	2	John	7	6	2	2	7
2	3	Alex	8	9	3	1	9
3	4	Derek	9	2	5	8	8
4	5	Oscar	2	6	1	7	Not_qualified
5	6	John	6	6	4	8	6
6	7	Derek	6	5	5	4	Not_qualified
7	8	Oscar	5	5	9	5	9

Infer_objects

Pandas supports a wide range of data types, one of which is **object**. Object covers text or mixed (numeric and non-numeric) values. However, it is not preferred to use object data type if a different option is available. Certain operations is executed faster with more specific data types. For example, we prefer to have integer or float data type for numerical values.

infer_objects attempts to infer better dtypes for object columns. Let's go through an example.

```
df = pd.DataFrame({'A': ['a', 1, 2, 3],
                   'B':['b',2.1,1.5,2],
                   'C':['c',True,False,False],
                   'D':['a','b','c','d']})
df
```

	A	B	C	D
0	a	b	c	a
1	1	2.1	True	b
2	2	1.5	False	c
3	3	2	False	d

```
df = df[1:]df.dtypes
A    object
B    object
C    object
D    object
dtype: object
```

	A	B	C	D
1	1	2.1	True	b
2	2	1.5	False	c
3	3	2	False	d

```
df.infer_objects().dtypesdf.dtypes
A      int64
B    float64
C       bool
D     object
dtype: object
```